SuperH RISC engine
C/C++ Compiler
Assembler
Optimizing Linkage Editor

User's Manual

HITACHI

Cautions

1. Hitachi neither warrants nor grants licenses of any rights of Hitachi's or any third party's patent, copyright, trademark, or other intellectual property rights for information contained in this document. Hitachi bears no responsibility for problems that may arise with third party's rights, including intellectual property rights, in connection with use of the information contained in this document.
2. Products and product specifications may be subject to change without notice. Confirm that you have received the latest product standards or specifications before final design, purchase or use.
3. Hitachi makes every attempt to ensure that its products are of high quality and reliability. However, contact Hitachi's sales office before using the product in an application that demands especially high quality and reliability or where its failure or malfunction may directly threaten human life or cause risk of bodily injury, such as aerospace, aeronautics, nuclear power, combustion control, transportation, traffic, safety equipment or medical equipment for life support.
4. Design your application so that the product is used within the ranges guaranteed by Hitachi particularly for maximum rating, operating supply voltage range, heat radiation characteristics, installation conditions and other characteristics. Hitachi bears no responsibility for failure or damage when used beyond the guaranteed ranges. Even within the guaranteed ranges, consider normally foreseeable failure rates or failure modes in semiconductor devices and employ systemic measures such as fail-safes, so that the equipment incorporating Hitachi product does not cause bodily injury, fire or other consequential damage due to operation of the Hitachi product.
5. This product is not designed to be radiation resistant.
6. No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without written approval from Hitachi.
7. Contact Hitachi's sales office for any questions regarding this document or Hitachi semiconductor products.

Preface

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

This manual is intended for UNIX*[1], Microsoft® Windows® 95 operating system, Microsoft® Windows® 98 operating system, Microsoft® Windows NT® operating system, and Microsoft® Windows® 2000 operating system*[2] that runs on an IBM PC*[3], and other compatible computers. In this document, the system operating on a UNIX system is referred to as the UNIX version. The system operating on IBM PC*[3] and other compatible computers are referred to as the PC version.

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.

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 Windows NT® are registered trademarks of Microsoft Corporation in the United States and other countries.
3. IBM PC is a registered trademark of International Business Machines Corporation.

Contents

Section 1 Overview

1.1 Procedures for Developing Programs

Figure 1.1 shows the procedures for developing programs. The shaded part shows software provided in the SuperH RISC engine C/C++ compiler package.

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

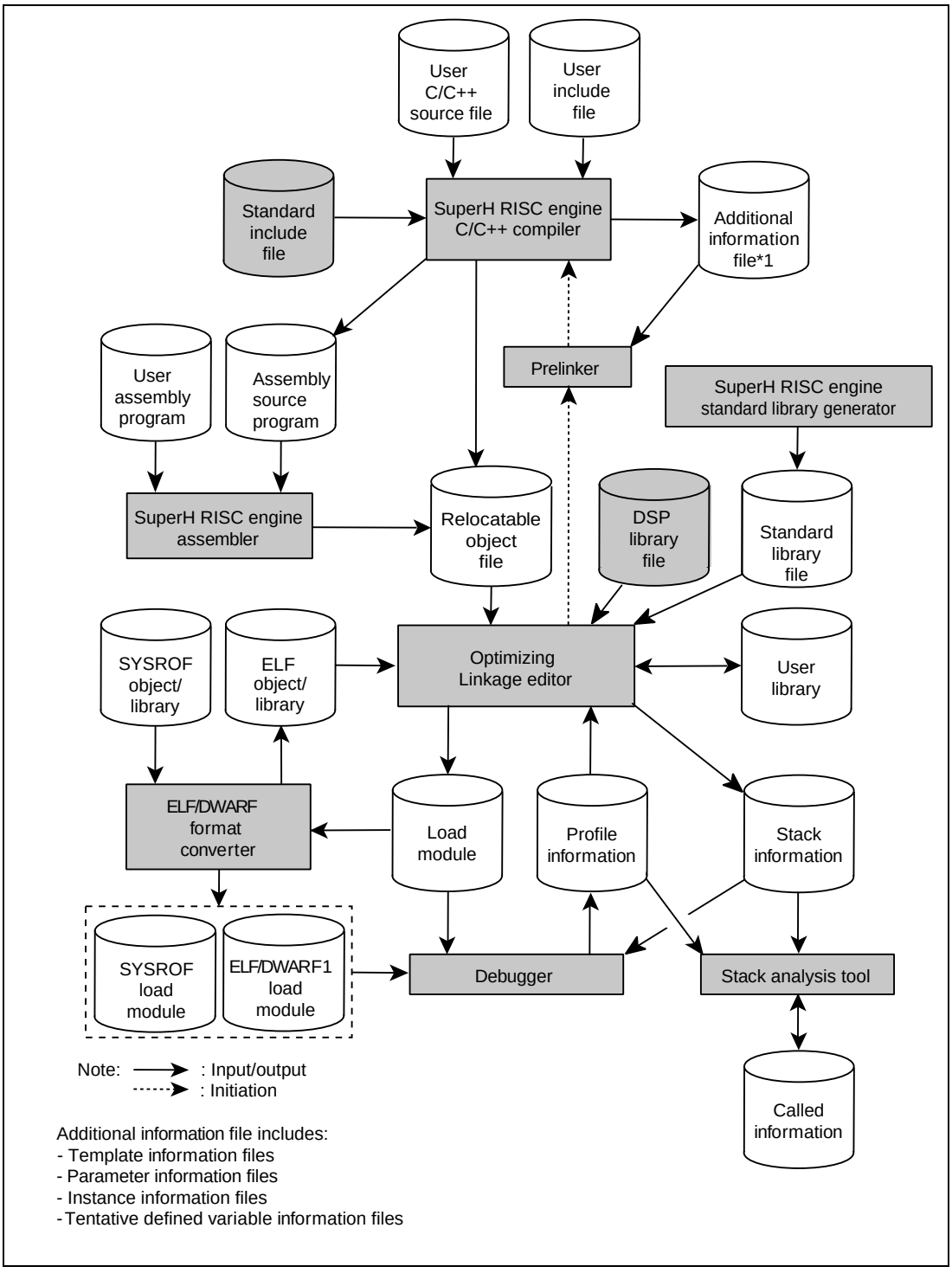


Figure 1.1 Procedures for Developing Programs

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 instructions.

1.2 Compiler

The SuperH RISC engine 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 SuperH RISC engine 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 the C and C++ programming languages .
4. Supports functions that are essential for the programming of embedded programs but are not supported by the C and C++ languages as extended functions. Such functions include interrupt functions and descriptions of system instructions.
5. The output of debugging information to enable C/C++ source-level debugging by the debugger is supported.
6. Either an assembly source program or a relocatable object program can be selected for output.
7. Supports an inter-module optimization information output to execute optimization for the optimizing linkage editor.

1.3 Assembler

The SuperH RISC engine assembler (hereinafter referred to as assembler) takes source programs written in assembly language, and outputs relocatable object programs for SuperH RISC engine 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
2. The mnemonics for execution instruction 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 load modules or library files.

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 is called from the optimizing linkage editor. When a C++ program template or runtime type-detection function is used, the prelinker calls the compiler and instructs it to generate the necessary object files. When neither a C++ program template nor the runtime type-detection function 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 SuperH RISC engine 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 routine 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 the earlier version of the linkage editor.

Section 2 Compiler Options

2.1 Command Line Format

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

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

2.2 List of Options

In the command line format, uppercase letters indicate the abbreviations. Characters underlined indicate the defaults.

The format of the dialog menus that correspond to Hitachi Embedded Workshop is as follows:
Tab name [Item]

Options are described in the order corresponding to tabs in Hitachi Embedded Workshop.

2.2.1 Source Tab Options

Table 2.1 Source Tab Options

Item	Command Line Format	Dialog Menu	Specification
Include file directory	Include = <path name>[,…]	Source [Show entries for:] [Include file directories]	Specifies include-file search path name.
Default include file	PREInclude = <file name>[,...]	Source [Show entries for:] [Preinclude files]	Includes the specified files at the head of compiled files.
Macro name definition	DEFine = <sub>[,...] <sub>: <macro name> [=<string literal>]	Source [Show entries for:] [Defines]	Defines <string literal> as <macro name>.

Include

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 the **include** option specification directory, the environment variable SHC_INC specification directory, and the environment variable SHC_LIB specification directory. User include files are retrieved in the order of the current directory, the **include** option specification directory, the environment variable SHC_INC specification directory, and the environment variable SHC_LIB specification directory.
- Example
  ```
  shc -include=/usr/inc,/usr/SHC test.c
  ```
  Directories /usr/inc and /usr/SHC are retrieved as include file paths.

PREInclude

Source[Show entries for:][Preinclude files]

- Command Line Format
  PREInclude = <file name>[,…]
- Description
  Includes the specified file at the head of the compiled files. Two or more path names can be specified by separating them with a comma (,).
- Example
  ```
  shc -preinclude=a.h test.c
  ```
  — Contents of <test.c>
  ```
  int a;
  main(){...}
  ```
  — Interpretation at compilation
  ```
  #include "a.h"
  int a;
  main(){...}
  ```

DEFine

Source[Show entries for:][Defines]

- Command Line Format

  DEFine = <sub> [,…]

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

- Description

  This option is the same as #define described 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 assumed to be defined. Names or integer constants can be written in **<string literal>**.

2.2.2 Object Tab Options

Table 2.2 Object Tab Options

Item	Command Line Format	Dialog Menu	Specification
Pre-processor expansion	PREProcessor [= <file name>]	Object [Output file type:] [Preprocessed source file]	Outputs source program after preprocessor expansion.
Object type	Code = { Machinecode | Asmcode }	Object [Output file type:] [Machine code] [Assembly source code]	 Outputs machine code program. Outputs assembly-source program.
Debugging information	DEBug NODEBug	Object [Generates debug information]	Output Not output

Section name	SEction = <sub>[,…]
	<sub>:{
	Program=<section name>
	| Const=<section name>
	| Data=<section name>
	| Bss=<section name>
	}
		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 = { Const | Data }	Object [Store string data in:]	Outputs string literal to constant section (C). Outputs string literal to initialized data section (D).
Object file output	OBjectfile = <file name>	Object [Output directory:]	Outputs the object file of the specified file name.

Table 2.2 Object Tab Options (cont)

Item	Command Line Format	Dialog Menu	Specification
Template instance generation	Template={ None	Object [Template:]	Does not generate instances.
	| Static		Generates instances as internal linkage only for referenced templates.
	| Used		Generates instances as external linkage only for referenced templates.
	| ALl		Generates instances for templates declared or referenced.
	| AUto }		Generates instances at linkage.
ABS16 declaration	ABs16={ RUn	Object [Use 16 bit short address]	Assumes all runtime routines to have been declared with #pragma abs16.
	| ALl }		Generates all label addresses in 16 bits.

PREProcessor

Object[Output file type:][Preprocessed source file]

- Command Line Format

  PREProcessor [= <file name>]
- Description

  Outputs 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 from the compiler.
- Remarks

  When **preprocessor** is specified, the following options become invalid:
  code, **debug**, **section**, **string**, **object**, **template**, **abs16**, **show=object**, **statistics**, **optimize**, **speed**, **goptimize**, **nestinline**, **inline**, **case**, **macsave**, **align16**, **rtnext**, **loop**, **fpscr**, **cpu**, **division**, **endian**, **fpu**, **round**, **denormalization**, **pic**, **double=float**, **exception**, **rtti**, and **outcode**.

Code

Object[Output file type:] [Machine code] [Assembly source code]

- Command Line Format
  Code = { Machinecode | Asmcode }
- Description
  Specifies an object program type.
  When **code=machinecode** is specified, a relocatable object program (machine code) is generated.
  When **code=asmcode** is specified, an assembly source program is generated.
  The default of this option is **code=machinecode**.
- Remarks
  When **code=asmcode** is specified, **show=object** or **goptimize** becomes invalid.

DEBug, NODEBug

Object[Generate debug information]

- Command Line Format
  DEBug
  NODEBug
- Description
  Specifies whether to output the debugging information needed for source-level debugging into object files.
  This option is valid whether or not the optimization option is specified.
  The **debug** option outputs the debugging information into object files.
  When **nodebug** option is specified, no debugging information will be output to the object file.
  The default of this option is **nodebug**.

SEction

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

- Command Line Format

  SEction = <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 the program area.

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

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

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

  The <section name> must be alphabetic, numeric, or underscore (_) or $. The first character must not be numeric. The section name must be specified within 8192 characters.

  The default of this option is **section=program=P**, **const=C**, **data=D**, **bss=B**.

- Remarks

  For details on programs and section names, refer to section 9.1, Program Structure.

STring

Object[Store string data in:]

- Command Line Format
  STring = { Const | Data }
- Description
  Specifies the destination where string literals are output.
  When **string=const** is specified, the compiler outputs the string literals in the source program to the constant area.
  When **string=data** is specified, the compiler outputs the string literals in the source program to the initialized data area.
  The string literals output to the initialized data area can be modified at the program execution; however, the initialized data area must be allocated in both ROM and RAM in order to transfer the string literals 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.
  The default of this option is **string=const**.

OBjectfile

Object[Output directory:]

- Command Line Format
  OBjectfile = <object file name>
- Description
  Specifies an object file name to be output.
  If this option is not specified, the object file name body 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 **code**.

Template

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 declared or referenced in the compiling unit are generated.

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

  When an assembly source file is output, **template=static** must be specified.

ABs16

Object[Use 16 bit short address]

- Command Line Format

  ABs16 = { RUn | ALl }
- Description

  abs16=run assumes all the runtime routines to have been declared with **#pragma abs16**.

  abs16=all generates every label address in 16 bits.

2.2.3 List Tab Options

Table 2.3 List Tab Options

Item	Command Line Format	Dialog Menu	Specification
Listing file	Listfile [= <file name>]	List	Output
	NOListfile	[Generate list file]	Not output
Listing contents and format	SHow = <sub> [,…] <sub>: { SOurce | NOSOurce	List [Contents]	With/without source list
	| Object | NOObject		With/without object list
	| STatistics | NOSTatistics		With/without statistics information
	| Include | NOInclude		With/without list after include expansion
	| Expansion | NOExpansion		With/without list after macro expansion
	| Width = <numeric value>		Maximum characters per line: 0 or 80 to 132
	| Length = <numeric value>		Maximum lines per page: 0 or 40 to 255
	}		

Listfile, NOListfile

List[Generate list file]

- Command Line Format
  Listfile [= <file name>]
  NOListfile
- Description
  Specifies whether a listing file is output or not.
  When **listfile** is specified, a file name can be specified.
  When **nolistfile** is specified, a listing file will not be output.
  File names should be specified following section 8.1, Naming Files.
  If no file name is specified, a listing file with the same name as the source file and a standard extension (lis/1st/lpp) is created. The standard extension for UNIX version is lis, that for PC version at C compilation is lst, and that for PC version at C++ compilation is lpp.
  The default of this option is **nolistfile**.

SHow

List[Contents]

- Command Line Format

  SHow= <sub>[,…]

  <sub>: { SOurce | NOSOurce
  | Object | NOObject
  | STatistics | NOSTatistics
  | Include | NOInclude
  | Expansion | NOExpansion
  | Width= <numeric value>
  | Length= <numeric value>
  }

- 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.

  The default of this option is **show=nosource, object, statistics, noinclude, noexpansion, width=0, length=0**.

- Remarks

  Table 2.4 shows a list of suboptions.

Table 2.4 List of Suboptions of show Option

Suboption	Description
source	Outputs a list of source programs
nosource	Outputs no list of source programs
object	Outputs a list of object programs
noobject	Outputs no list of object programs
statistics	Outputs a list of statistics information
nostatistics	Outputs no list of statistics information
include	Outputs a source program list after include file expansion. If the **nosource** suboption and the **include** suboption are specified, the **include** suboption will be invalid, and no source program list will be output to a file.
noinclude	Outputs a source program list before include file expansion. If the **nosource** suboption is and the **noinclude** suboption are specified, the **noinclude** suboption will be invalid, and no source program list will be output to a file.
expansion	Outputs a source program list after macro expansion. If the **nosource** suboption and the **expansion** suboption are specified, the **expansion** suboption will be invalid, and no source program list will be output to a file.
noexpansion	Outputs a source program list before macro expansion. If the **nosource** suboption and the **noexpansion** suboption are specified, the **noexpansion** 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 specified.
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 40 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 specified.

2.2.4 Optimize Tab Options

Table 2.5 Optimize Tab Options

Item	Command Line Format	Dialog Menu	Specification
Optimization	OPtimize = { 0 | 1 }	Optimize [Optimization]	Outputs object without optimization. Outputs object with optimization.
Optimized for speed		Optimize [Speed or size:]	
	SPeed	[Optimize for speed]	Performs optimization for speed.
	SIze	[Optimize for size]	Performs optimization for program size.
	NOSPeed	[Optimize for both speed and size]	Performs balanced optimization between execution speed and program size.
Inter-module optimization information	Goptimize	Optimize [Generate file for inter-module optimization:]	Outputs information for inter-module optimization.
Nested inline expansion	NEstinline = <numeric value>	Optimize [Inline function nesting]	Specifies the depth of nesting in inline function expansion.
Automatic inline expansion	INLine [= <numeric value>]	Optimize [Automatic inline expansion]	Performs inline expansion automatically.
	NOINLine		Does not perform inline expansion automatically.
switch statement expansion method	CAse = { Ifthen | Table }	Optimize [Switch statement:]	Expands by **if_then** method. Expands by jumping to a table.

OPtimize

Optimize[Optimization]

- Command Line Format
  OPtimize = { 0 | 1 }
- Description
  Specifies the level of compiler optimization.
  When **optimize=0** is specified, the compiler does not optimize the object program.
  When **optimize=1** is specified, the compiler optimizes the object program.
  The default of this option is **optimize=1**.

SPeed, SIze, NOSPeed

Optimize[Speed or size:]

- Command Line Format
  SPeed
  SIze
  NOSPeed
- Description
  When **speed** is specified, the compiler performs optimization with priority in execution.
  When **size** is specified, the compiler performs optimization with priority in program size.
  If **nospeed** is specified, the compiler performs balanced optimization between execution speed and program size.
  The default of this option is **nospeed**.

Goptimize

Optimize[Generate file for inter-module optimization]

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

NEstinline

Optimize[Inline function nesting]

- Command Line Format
  NEstinline=<numeric value>
- Description
  Specifies the depth of nesting in inline expansion functions. Up to 16 can be specified. The default of this option is **nestline=1**.
- Example
  Source program

```
#pragma inline(fun1,fun2,fun3)
extern int dat;
void fun1(){a++;}
void fun2(){fun1();}
void fun3(){fun2();}
```

  (1) Inline expansion image for nestinline=1

```
#pragma inline(fun1,fun2,fun3)
extern int dat;
void fun1(){a++;}
void fun2(){a++;}
void fun3(){fun1();}
```

  (2) Inline expansion image for nestinline=2

```
#pragma inline(fun1,fun2,fun3)
extern int dat;
void fun1(){a++;}
void fun2(){a++;}
void fun3(){a++;}
```

INLine, NOINLine

Optimize[Automatic inline expansion]

- Command Line Format
  INLine=<numeric value>
  NOINline
- Description
  Specifies whether to automatically perform inline expansion of functions.
  When **inline** is specified, the compiler automatically performs inline expansion.
  inline=**<numeric value>** specifies the maximum number of nodes in a function (total number of words such as variables and operators except for the declarations) against which inline expansion should be performed.
  When **noinline** is specified, inline expansion is not performed.
  If **speed** is specified, the default of this option is **inline=20**. If the **nospeed**, **size**, or **optimize=0** is specified, the default is **noinline**.

CAse

Optimize[Switch statement:]

- Command Line Format
  CAse = { Ifthen | Table }
- Description
  Specifies a switch statement expansion method.
  When **case=ifthen** is specified, switch statement is expanded using the if_then method, which repeats, for each case label, comparison between the evaluated value of the expression in the switch statement and the case label value. If they match, execution 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 is expanded 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 number of case labels in the switch statement, but the execution speed is always the same.
  If this option is not specified, the compiler automatically selects one of the methods for expansion.

2.2.5 Other Tab Options

Table 2.6 Other Tab Options

Item	Command Line Format	Dialog Menu	Specification
Embedded C++ language	ECpp	Other [Miscellaneous options:] [Check against EC++ language specification]	Checks syntax according to the Embedded C++ language specifications.
Comment nesting	COMment = { Nest | NONest }	Other [Miscellaneous options:] [Allow comment nest]	Permits comment (/* */) nesting. Does not permit comment (/* */) nesting.
Message output control	MEssage NOMEssage	Other [Miscellaneous options:] [Display information level messages]	Outputs information message. Does not output information message.
MAC register	Macsave = { 0 | 1 }	Other [Miscellaneous options:] [Callee saves/restores MACH and MACL registers if used]	Does not guarantee the MAC register contents after a function is called. Guarantees the MAC register contents after a function is called.
16-byte alignment of labels	ALign16 NOALign16	Other [Miscellaneous options:] [Align Labels after unconditional branches 16byte boundaries]	Labels placed immediately after an unconditional branch instruction other than a subroutine call in a program section is aligned on a 16-byte boundary. Does not necessarily place labels on a 16-byte boundary.

Table 2.6 Other Tab Options (cont)

Item	Command Line Format	Dialog Menu	Specification
Extension of return value	RTnext	Other [Miscellaneous options:] [Expand return value to 4 byte]	Creates a sign-extension or zero-extension of the return value
	NORTnext		Creates no sign-extension or zero-extension of the return value
Loop unroll	LOop	Other [Miscellaneous options:] [Loop unrolling]	Performs loop unrolling.
	NOLOop		Does not perform loop unrolling.
FPSCR register switch	FPScr = { Safe	Other [Miscellaneous options:] [Change FPSCR register if double data used]	Switches the FPSCR register whenever a double operation is generated.
	| Aggressive		Inhibits switching FPSCR as much as possible.

ECpp

Other [Miscellaneous options:] [Check against EC++ language specification]

- Command Line Format

  ECpp
- Description

  The compiler checks the syntax of the C++ source program according to the Embedded C++ language specifications. The Embedded C++ specifications do not support such keywords as 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.
- 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 in the source program, the compiler will display warning message "C5882 (W) Embedded C++ does not support multiple or virtual inheritance" at compilation. A compiler generating object program at the output of the warning message C5882 is the same as that without the **ECpp** specified.

COMment

Other [Miscellaneous options:] [Allow comment nest]

- Command Line Format
  COMment={Nest | NONest}
- Description
  Allows nested comments to be written in the source program.
  When this option is not specified, and if nested comments are written, an error will occur.
- Example

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

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

MEssage, NOMEssage

Other[Miscellaneous options:] [Display information level messages]

- Command Line Format
  MEssage
  NOMEssage
- Description
  Specifies whether to output information-level messages.
  If **message** is specified, the compiler outputs information-level messages.
  If **nomessage** is specified, the compiler does not output information-level messages.
  The default of this option is **nomessage**.
- Example

```
shc –message test.c
```

  Information-level messages will be displayed.

Macsave

Other [Miscellaneous options:]
[Callee saves/restores MACH and MACL registers if used]

- Command Line Format

  Macsave = { 0 | 1 }
- Description

  Specifies whether or not to the contents of the MACH and MACL registers before a function call are guaranteed or after the function call.

  When **macsave=0** is specified, the contents of the MACH and MACL registers before a function call are not guaranteed or after the function call.

  If **macsave=1** is specified, the contents of the MACH and MACL registers before a function call are guaranteed or after a function call.

  Functions compiled under **macsave=0** cannot be called from functions compiled under **macsave=1**. On the contrary, functions compiled under macsave=1 can be called from functions compiled under **macsave=0**.

  The default of this option is **macsave=1**.

ALign16, NOALign16

Other [Miscellaneous options:] [Align Labels after unconditional branches 16byte boundaries]

- Command Line Format

  ALign16

  NOALign16
- Description

  When **align16** is specified, every label within the program section that is placed immediately after an unconditional branch instruction except for subroutine call is aligned on a 16-byte boundary.

  When **noalign16** is specified, 16-byte alignment of labels that are placed immediately after an unconditional branch instructions is not performed.

  The default of this option is **noalign16**.

RTnext, NORTnext

Other [Miscellaneous options:] [Expand return value to 4 byte]

- Command Line Format

  RTnext

  NORTnext
- Description

  Specifies whether to perform sign/zero extension of a function return value in register R0 against a return statement that returns an (unsigned) char type or (unsigned) short type value. This option need not be specified if function prototype is declared.

  When **rtnext** is specified, sign/zero extension of the function return value is performed.

  When **nortnext** is specified, sign/zero extension of the function return value is not performed.

  The default of this option is **nortnext**.

LOop, NOLOop

Other [Miscellaneous options:] [Loop unrolling]

- Command Line Format

  LOop

  NOLOop
- Description

  Specifies whether to perform loop unrolling.

  When **loop** is specified, priority is given to execution speed in compiling loop statements (for, while, and do-while).

  When **noloop** is specified, priority is not given to execution speed in compiling loop statements.

  The default of this option is **noloop**.

FPScr

Other [Miscellaneous options:] [Change FPSCR register if double data used]

- Command Line Format

  FPScr = { Safe

  | Aggressive }
- Description

  Specifies single or double precision mode for the FPSCR register when executing float or double operation in SH-4.

  When **fpscr=safe** is specified, the compiler switches the FPSCR register to double precision mode every time the SH-4 performs double operation. After the SH-4 completes double precision operation, the compiler switches the register to single precision mode. In this case, the compiler always switches the FPSCR register to single precision mode after returning from a function call.

  When **fpscr=aggressive** is specified, the compiler tries not to change the FPSCR register very often. In this case, the contents of the FPSCR register are not guaranteed in the single precision mode after returning from a function call.

  This option is valid without **fpu=single** or **fpu=double** when **cpu=sh4**.

  The default of this option is **fpscr=aggressive**.

2.2.6 CPU Tab Options

Table 2.7 CPU Tab Options

Item	Command Line Format	Dialog Menu	Specification
CPU operating mode	CPu =	CPU	
	{ sh1	[CPU:]	Generates SH-1 object.
	| sh2		Generates SH-2 object.
	| sh2e		Generates SH-2E object.
	| sh		Generates SH-3 object.
	| sh3e		Generates SH-3E object.
	| sh4 }		Generates SH-4 object.
Division operation [SH-2]	Division =	CPU	
	{ Cpu	[Division:]	Uses CPU's division instruction.
	| Peripheral		Uses a divider (with masking interrupt).
	| Nomask }		Uses a divider (without masking interrupt).
Byte order [SH-3 to SH-4]	ENdian = { Big	CPU	Specifies big endian.
	| Little }	[Endian:]	Specifies little endian.
FPU [SH-4]	FPu=	CPU	
	{ Single	[FPU:]	Processes floating-point operation in single precision.
	| Double }		Processes floating-point operation in double precision.
Rounding direction [SH-4]	Round= { Zero	CPU	Rounds to zero.
	| Nearest }	[Round to:]	Rounds to nearest.
Denormalized numbers [SH-4]	DENormalization= { OFF	CPU [Denormalized number allower as a result]	Processes denormalized numbers as zeros.
	| ON }		Processes denormalized numbers as they are.

Table 2.7 CPU Tab Options (cont)

Item	Command Line Format	Dialog Menu	Specification
Program section position independent [SH-2 to SH-4]	Pic= { 0 | 1 }	CPU [Position independent code (PIC)]	Generates no position independent codes for the program section. Generates position independent codes for the program section.
double to float conversion [SH-1 to SH-3E]	DOuble=Float	CPU [Treat double as float]	Handles a double-type variable as a float-type variable.
Exception processing	EXception NOEXception	Other [Miscellaneous options:] [Use try, throw and catch of C++]	Enables exception processing function Disables exception processing function.
Runtime type information	RTTI= {ON | OFF }	CPU [Enable/disable runtime information]	Enables dynamic_cast and typeid. Disables dynamic_cast and typeid.

CPu

CPU[CPU]

- Command Line Format

  CPu = { sh1
  | sh2
  | sh2e
  | sh3
  | sh3e
  | sh4
  }

- Description

  Specifies the CPU type for the object program to be generated. Suboptions are listed in table 2.8.

  The default of this option is cpu=sh1.

Table 2.8 Suboptions for cpu Option

Suboption	Description
sh1	Generates SH-1 object.
sh2	Generates SH-2 object.
sh2e	Generates SH-2E object.
sh3	Generates SH-3 object.
sh3e	Generates SH-3E object.
sh4	Generates SH-4 object.

Division

CPU[Division:]

- Command Line Format

  Division = { Cpu
  | Peripheral
  | Nomask
  }

- Description

  Selects runtime routines for integer type division and residue.

  When **division=cpu** is specified, the runtime routine by the DIV1 instruction is selected.

  When **division=peripheral** is specified, the runtime routine that uses the divider is selected (Sets interrupt mask level to 15.) Executable only if cpu type is SH-2 (SH7604).

  When **division=nomask** is specified, the runtime routine that uses the divider is selected. (No change in interrupt mask level.) Executable only if cpu type is SH-2 (SH7604).

  When specifying **peripheral** or **nomask**, note the following:

  1. Division by 0 is not checked and errno is not set up.
  2. When **nomask** is specified, if an interrupt occurs during operation of the divider, and if the divider is used in the interrupt process routine, the result is not guaranteed.
  3. Overflow interrupt is not supported.
  4. Results of division by zero and overflow depend on specifications of the divider, and may differ from the results obtained when **cpu** is specified.

  The default of this option is **division=cpu**.

ENdian

CPU[Endian:]

- Command Line Format

  ENdian = { Big | Little }

- Description

  When **endian=big** is specified, data bytes are arranged in the Big Endian order.

  When **endian=little** is specified, data bytes are arranged in the Little Endian order.

  Little endian object programs do not run on SH-1, SH-2, and SH-2E.

  The default of this option is **endian=big**.

FPu

CPU[FPU:]

- Command Line Format
  FPu = { Single | Double }
- Description
  When **fpu=single** is specified, all floating point calculations are carried out at single precision.
  When **fpu=double** is specified, all floating point calculations are carried out at double precision.
  Specify **fpu=single** if floating point calculations are not used in the program.
  This option is valid when **cpu=sh4**.

Round

CPU[Round to:]

- Command Line Format
  Round = { Zero | Nearest }
- Description
  When **round=zero** is specified, values are rounded to zero.
  When **round=nearest** is specified, values are rounded to nearest.
  This option is valid when **cpu=sh4**.
  The default of this option is **round=zero**.

DENormalization

CPU[Denormalization number allower as a result]

- Command Line Format
  DENormalization = { OFF | ON }
- Description
  When **denormalization=off** is specified, denormalized numbers are treated as zeros.
  When **denormalization=on** is specified denormalized numbers as treated as they are.
  This option is valid when **cpu=sh4**.
  The default of this option is **denormalization=off**.

Pic

CPU[Position independent code (PIC)]

- Command Line Format
  Pic = { 0 | 1 }
- Description
  When **pic=1** is specified, a program section after linking can be allocated to any address and executed. A data section can only be allocated to an address specified at linking. When using this option as a position independent code, a function address cannot be specified as an initial value. At C++ compilation, a pointer to a virtual function or function member requires a function address as the initial value. Therefore, C++ programs containing virtual functions and pointers to member functions cannot be executed as position independent codes.
- Examples
  Example 1

```
extern int f ();
int (*fp)() = f;        <-- Cannot be specified
```

  Example 2

```
struct A {virtual void f();}; <-- Cannot be specified
void (A::*ap)() = &A::f;    <-- Cannot be specified
```

  Note that if **cpu=sh1** is specified, **pic=1** is ignored.
  The default of this option is **pic=0**.

DOuble=Float

CPU[Treat double as float]

- Command Line Format
  DOuble=Float
- Description
  Generates an object with converting double-type (double-precision floating-point) values to float-type (single-precision floating-point) values.
- Remarks
  This option is invalid when **cpu=sh4** has been specified, and assumes that **fpu=single** is specified.

EXception, NOEXception

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

- Command Line Format
  EXception
  NOEXception
- Description
  Enables the C++ exception processing (try, catch, throw).
  When this option is specified, the code performance may be reduced.
  The default of this option is **noexception**.

RTTI

[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.
  The default of this option is **rtti=off**.
- 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. Symbol double definition errors or symbol undefined error will occur.

2.2.7 Options Other than Above

Table 2.9 Options Other than Above

Item	Command Line Format	Dialog Menu	Specification
Selecting C or C++ language	LAng = { C	—	Compiled as C source program.
	| CPp }	(Determined by an extension)	Compiled as C++ source program.
Disable of Copyright output	LOGO	—	Outputs Copyright.
	NOLOGO	(**nologo** is always valid)	Disables to output Copyright.
Character code select in string literals	EUc	—	Selects euc code.
	SJis		Selects sjis code.
Japanese character conversion within object code	OUtcode = { EUc	—	Selects euc code.
	| SJis }		Selects sjis code.
Subcommand file	SUbcommand = <file name>	—	Command option is fetched from the file specified with <file name>.

LAng

None (Always determined by an extension)

- Command Line Format

  LAng = { C | CPp }
- Description

  Specifies the language of the source program.

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

  When **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 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

`shc test.c`	Compiled as a C source program.
`shc test.cpp`	Compiled as a C++ source program.
`shc -lang=cpp test.c`	Compiled as a C++ source program.
`shc test`	Assumed to be test.c and thus be compiled as a C source program.

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

LOGO, NOLOGO

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

Euc, Sjis

None

- Command Line Format
  Euc
  Sjis
- Description
  Use this option to specify the Japanese character code written in a string literal, a character constant, or a comment.

Table 2.10 shows character code in the string literals for three types of host computers.

Table 2.10 Relationship between the Host Computer and Character Code in String Literals

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

OUtcode

None

- Command Line Format
  OUtcode = {EUc | SJis}
- Description
  Specifies the Japanese character code to be output to the object program when Japanese is written in string literals and character constants.
  When the **outcode=euc** is specified, the compiler outputs the Japanese character code in the **euc** code.
  When the **outcode=sjis** is specified, the compiler outputs the Japanese character code in the **sjis** code.
  Option **euc** or **sjis** can be specified for the Japanese character code in a source program.

SUbcommand

None

- Format
  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`
  Command line specification: `shc -cpu=sh4 -subcommand=opt.sub test.c`
  Interpretation at compilation: `shc -cpu=sh4 -show=object –debug test.c`

Section 3 Assembler Options

3.1 Command Line Format

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

```
asmsh [Δ<option> …] [Δ<file name> [,…]] [Δ<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 Hitachi Embedded Workshop is as follows:

Tab name [Item]

Options are described in the order of tabs in Hitachi Embedded Workshop option dialog box.

3.2.1 Source Tab Options

Table 3.1 Source Tab Options

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

Include

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: `asmsh aaa.src –include=C:\common,C:\local`

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

  The current directory, C:\common,C:\local 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 specified by the **include** option is added before that specified by **.INCLUDE**.

DEFine

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

Relationship with Assembler Directives

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

ASsignA

Source[Show entries for:][Preprocessor variables]

- Command Line Format

  ASsignA = <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: `asmsh aaa.src -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.

ASsignC

Source [Show entries for:][Preprocessor variables]

- Command Line Format

  ASsignC = <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 (bytes) can be specified for a string literal.

  Example: `asmsh aaa.src -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 string literals before and after the following characters, enclose the string literals 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 Tab Options

Table 3.2 Object Tab Options

Item	Command Line Format	Dialog Menu	Specification
Debugging information	Debug NODebug	Object [Debug information:]	Controls output of debugging information.
Pre-processor expansion result	EXPand [= <output file name>]	Object [Generate assembly source file after preprocess]	Outputs preprocessor expansion result.
Literal pool output point	LITERAL = <point> [, …] <point>: {Pool | Branch | Jump | Return}	Object [Generate literal pool after:]	Specifies the point to output literal pool.
Object module output	Object [= <output file name>] NOObject	Object [Output file directory:]	Controls object module output.

Debug, NODebug

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

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, 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.

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

LITERAL

Object [Generate assembly source file after preprocess]

- Command Line Format

  LITERAL = <point>[,…]

  <point>:{Pool|Branch|Jump|Return}
- Description

  The **literal** option specifies the point where the literal pool that was created by the automatic literal pool creation function is placed.

  — pool: The literal pool is output at the location of the **.POOL** directive.

  — branch: The literal pool is output after the **BRA/BRAF** instruction.

  — jump: The literal pool is output after the **JMP** instruction.

  — return: The literal pool is output after the **RTS/RTE** instruction.

  When this option is omitted, the assembler assumes **literal** = **pool**, **branch**, **jump**, **return** is specified.

Object, NOObject

Object [Output file directory:]

- Command Line Format

  Object [= <object output file>]

  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 Tab Options

Table 3.3 List Tab Options

Item	Command Line Format	Dialog Menu	Specification
Assemble listing output control	LISt [= <output file name>] NOLISt	List [Generate list file]	Controls output of assemble listing
Source program listing output control*	SOurce NOSOurce	List [Contents:] [Source program:]	Controls output of source program listing.
Part of source program listing output control*	SHow [= <item>[, …]] NOSHow [= <item>[, …]] <item>: {CONditionals | Definitions | CAlls | Expansions | CODe}	List [Contents:] [Conditions:] [Definitions:] [Calls:] [Expansions:] [Code:]	Controls output of parts of source program listing.
Cross-Reference Listing Output Control*	CRoss_reference NOCRoss_reference	List [Contents:] [Cross reference:]	Controls output of cross-reference listing.
Section Information Listing Output Control*	SEction NOSEction	List [Contents:] [Section:]	Controls output of section information listing.

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

LISt, NOLISt

List [Generate list file]

- Command Line Format

  LISt [= <listing output file>]

  NOLISt
- Description

  The **list** option specifies output of an assemble listing.

  The **nolist** option specifies no output of an assemble listing.

  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 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 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.

SOurce, NOSOurce

List [Contents:] [Source program:]

- Command Line Format

  SOurce

  NOSOurce
- Description

  The **source** option specifies output of a source program listing to the assemble listing.

  The **nosource** option specifies no output of 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

List [Contents:] [Conditions:], [Definitions:], [Calls:], [Expansions:], [Code:]

- Command Line Format

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

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

  <output type>: {CONditionals | Definitions | CAlls | Expansions | 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 .ASSSIGNC directive statements
calls	Call	Macro call statements, .AIF, .AIFDEF, and .AENDI directive statements
expansions	Expansion	Macro expansion statements .AREPEAT and .AWHILE 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

List [Contents:] [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.

SEction, NOSEction

List [Contents:] [Section:]

- Command Line Format

  SEction

  NOSEction
- 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 Other Tab Option

Table 3.4 Other Tab Option

Item	Command Line Format	Dialog Menu	Specification
Size mode specification for automatic literal pool generation	AUTO_literal	Other [Miscellaneous options:] [Automatically generate literal pool for immediate value]	Specifies size mode for automatic literal pool generation.

AUTO_literal

Other [Miscellaneous options:] [Automatically generate literal pool for immediate value]

- Command Line Format

  AUTO_literal
- Description

  The **auto_literal** option specifies the size mode for automatic literal pool generation.

  When this option is specified, automatic literal pool generation is performed in size selection mode, and the assembler checks the **imm** value in the data transfer instruction without operation size specification (MOV #imm,Rn) and automatically generates a literal pool if necessary.

When this option is not specified, automatic literal pool generation is performed in size specification mode, and the data transfer instruction without size specification is handled as a 1-byte data transfer instruction.

In the size selection mode, the **imm** value in the data transfer instruction without operation size specification is handled as a signed value. Therefore, a value within the range from H'00000080 to H'000000FF (128 to 255) is regarded as word-size data.

	Selected Size or Error	
imm Value Range	**Size Selection Mode**	**Size Specification Mode**
H'80000000 to H'FFFF7FFF (–2,147,483,648 to –32,769)	Long word	Warning 835
H'FFFF8000 to H'FFFFFF7F (–32,768 to –129)	Word	Warning 835
H'FFFFFF80 to H'0000007F (–128 to 127)	Byte	Byte
H'00000080 to H'000000FF (128 to 255)	Word	Byte
H'00000100 to H'00007FFF (256 to 32,767)	Word	Warning 835
H'00008000 to H'7FFFFFFF (32,768 to 2,147,483,647)	Long word	Warning 835

Note: The value in parentheses ( ) is in decimal.

3.2.5 CPU Tab Options

Table 3.5 CPU Tab Options

Item	Command Line Format	Dialog Menu	Specification
Target CPU specification	CPU = <target CPU>	CPU [CPU:]	Specifies target CPU.
Endian type specification	ENdian = {Big | Little}	CPU [Endian:]	Selects big endian or little endian.
Rounding direction of floating-point data	Round = {Nearest | Zero}	CPU [Round to:]	Specifies the rounding mode for floating-point data.
Handling denormalized numbers in floating-point data	DENormalize = {ON | OFF}	CPU [Denormalize:]	Specifies how to handle denormalized numbers in floating-point data.

CPU

CPU [CPU:]

- Command Line Format
  CPU = <target CPU>
- Description
  The **cpu** option specifies the target CPU for the source program to be assembled.
  The following CPUs can be specified.
  — SH1 (for SH-1)
  — SH2 (for SH-2)
  — SH2E (for SH-2E)
  — SH3 (for SH-3)
  — SH3E (for SH-3E)
  — SH4 (for SH-4)
  — SHDSP (for SH2-DSP)
  — SH3DSP (for SH3-DSP)

Relationship with Assembler Directives

Option	Assembler Directive	SHCPU Environment Variable	Result
cpu= <target CPU>	(regardless of any specification)	(regardless of any specification)	Target CPU specified by cpu
(no specification)	.CPU <target CPU>	(regardless of any specification)	Target CPU specified by .CPU
	(no specification)	SHCPU = <target CPU>	Target CPU specified by SHCPU environment variable
		(no specification)	SH1

ENdian

CPU [Endian:]

- Command Line Format
  ENdian = {Big | Little}
- Description
  The **endian** option selects big endian or little endian for the target CPU.
  The default is big endian.

Relationship with Assembler Directives

The assembler gives priority to specifications made by options.

Option	Assembler Directive	Result
endian=big	(regardless of any specification)	Assembles in big endian
endian=little	(regardless of any specification)	Assembles in little endian
(no specification)	.ENDIAN BIG	Assembles in big endian
	.ENDIAN LITTLE	Assembles in little endian
	(no specification)	Assembles in big endian

Round

CPU [Round to:]

- Command Line Format
  Round = {Nearest | Zero}
- Description
  The **round** option specifies the rounding mode used when converting constants in floating-point data assembler directives into object codes.
  The following two rounding modes can be selected.
  — round to NEAREST even (**nearset**)
  — round to ZERO (**zero**)
  When the **round** option is omitted, the rounding mode depends on the target CPU as follows:

Target CPU	Rounding Mode
SH-1	round to NEAREST even
SH-2	round to NEAREST even
SH-2E	round to ZERO
SH-3	round to NEAREST even
SH-3E	round to ZERO
SH-4	round to ZERO
SH2-DSP	round to NEAREST even
SH3-DSP	round to NEAREST even

Note: When the target CPU is SH-2E or SH-3E and **round to NEAREST even** is selected as the rounding mode, warning 818 occurs at the first floating-point data assembler directive in the source program, and object code is output in the selected "round to NEAREST even" rounding mode.

DENormalize

CPU [Denormalize:]

- Command Line Format

  DENormalize = {ON | OFF}

- Description

  The **denormalize** option specifies whether to handle the denormalized numbers in floating-point data assembler directives as valid values.

  The object code differs when denormalized numbers are specified as valid values (ON) and invalid values (OFF).

  — Valid: Warning 842 occurs and the object code is output.

  — Invalid: Warning 841 occurs and zero is output for the object code.

  When the **denormalize** option is omitted, whether the denormalized numbers are valid depends on the target CPU as follows:

Target CPU	Denormalized Numbers
SH-1	Valid (ON)
SH-2	Valid (ON)
SH-2E	Invalid (OFF)
SH-3	Valid (ON)
SH-3E	Invalid (OFF)
SH-4	Invalid (OFF)
SH2-DSP	Valid (ON)
SH3-DSP	Valid (ON)

Note: When the target CPU is SH-2E or SH-3E and denormalized numbers are specified as valid, warning 818 occurs at the first floating-point data assembler directive in the source program, and object code is output with the denormalized numbers handled as valid values as specified.

3.2.6 Options Other than Above

Table 3.6 Options Other than Above

Item	Command Line Format	Dialog Menu	Specification
Change of error level at which the assembler is abnormally terminated	ABort = {Warning | Error}	Other [User defined options:]	Changes the error level at which the assembler is abnormally terminated.
Western code character enabled	LATIN1	Other [User defined options:]	Enables the use of Western code characters in source file.
Interpretation of Japanese character as Shift JIS code	SJIS	Other [User defined options:]	Interprets Japanese character in source file as shift JIS code.
Interpretation of Japanese character as EUC code	EUC	Other [User defined options:]	Interprets Japanese character in source file as EUC code.
Specification of Japanese character	OUtcode = {SJIS | EUC}	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>	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>	Other [User defined options:]	Specifies the number of digits in assemble listing.
Copyright	LOGO NOLOGO	- (nologo is always valid)	Output Not output
Specification of subcommand	SUBcommand = <file name>	-	Inputs command line from a file.

ABort

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

Other [User defined options:]

- Command Line Format
  LATIN1
- Description
  The **latin1** option enables the use of Western code characters in string literals and in comments.
  Do not specify this option together with the **sjis**, **euc**, or **outcode** option.

SJIS

Other [User defined options:]

- Command Line Format
  SJIS
- Description
  When the **sjis** option is specified, Japanese characters in string literals and comments are interpreted as shift **JIS** code.
  When the **sjis** option is omitted, Japanese characters in string literals 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

Other [User defined options:]

- Command Line Format
  EUC
- Description
  When the **euc** option is specified, Japanese characters in string literals and comments are interpreted as **EUC** code.
  When the **euc** option is omitted, Japanese characters in string literals 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

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:

	Japanese Character in Source File		
outcode Option	**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
SPARC station	EUC code
HP9000/700 series	Shift JIS code
PC	Shift JIS code

LINes

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

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

Other [User defined options:]

- Command Line Format

  SUBcommand = <file name>
- Description

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

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

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

  This option must not be specified in a subcommand file.

Example:

```
asmsh aaa.src -subcommand=aaa.sub
```

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

aaa.sub contents

```
bbb.src
-list
-noobj
```

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

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

Note

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>]
```

For dctails, refer to section 4.2.7, Subcommand File.

4.2 List of Options

Table 4.1, 4.2, 4.6, 4.10, 4.11, 4.12, and 4.13 show Linkage Editor option formats, abbreviations, and defaults. In the command line format, uppercase letters indicate abbreviations. Underlined characters indicate the default settings.

The format of the dialog menus that correspond to the Hitachi Embedded Workshop is as follows: Tab name [Item]

The order of option description corresponds to that of tabs in the Hitachi Embedded Workshop.

4.2.1 Input Tab Options

Table 4.1 Input Tab Options

Item	Command Line Format	Dialog Menu	Specification
Input file	Input = <sub>[{,|Δ}…] <sub>: <file name> [(<module name>[,…])]	Input [Input files:] [Relocatable files and object files]	Specifies input file. (Input file is specified without **input** on the command line.)
Library file	LIBrary = <file name>[,...]	Input [Input files:] [Library files]	Specifies input library file.
Binary file	Binary = <sub> [,...] <sub>: <file name>(<section name> [,<symbol name>])	Input [Input files:] [Binary files]	Specifies input binary file.
Symbol definition	DEFine = <sub>[,…] <sub>: <symbol name> = {<symbol name> |<numerical value>}	Input [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>}	Input [Use entry point:]	Specifies an entry symbol. Specifies an entry address.
Prelinker	NOPRElink	Input [Prelinker control:]	Disables prelinker initiation.

Input

Input[Input files:][Relocatable files and object files]

- Command Line Format

  Input= <suboption>[{, | Δ}…]

  <suboption>:<file name>[(<module name>[,…])]
- Description

  Specifies input files. 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. Upper-case letters are expanded before lower-case 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

Input[Input files:][Library files]

- Command Line Format

LIBrary= <file name>[,…]

- Description

Specifies a 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. Upper-case letters are expanded before lower-case letters.

If **form=library** or **extract** is specified, the library file is input as the object 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.
```

Binary

Input[Input files:][Binary files]

- Command Line Format

  Binary = <suboption>[,…]

  <suboption>:<file name>(<section name>[,<symbol name>])
- Description

  Specifies a 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. The 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.
- Example

  ```
  input=a.obj
  start=p,D*/200
  binary=b.bin(D1bin),c.bin(D2bin,_datab)
  ```

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

  Allocates **c.bin** after **D1bin** as the **D2bin** section.

  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

Input[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 numerical value.

  The numerical value 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 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. For a **C++** function name (except for the **main** function), enclose the definition name with double quotation marks in the program including parameter strings.

- 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** or **library** is specified, this option is unavailable.

ENTry

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 symbol 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. However, the **void** argument is specified with '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.

NOPRElink

Input[Prelinker control:]

- Command Line Format
  NOPRElink
- Description
  Disables the prelinker initiation.
  The prelinker supports the function to generate the C++ template instance automatically. When the C++ template function is 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 Tab Options

Table 4.2 Output Tab Options

Item	Command Line Format	Dialog Menu	Specification
Output format	FOrm ={ Absolute | Relocate | Object | Library [= {S|U}] | Hexadecimal | Stype | Binary }	Output [Type of output file:]	Absolute format Relocatable format Object format Library format HEX format S-type format Binary format
Debug information	DEBug SDebug NODEBug	Output [Debug information:]	Output (in output file) Debug information file output Not output
Record size unification	REcord={ H16 | H20 | H32 | S1 | S2 | S3 }	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>	Output [Show entries:] [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>[:…]}	Output [Show entries:] [Divided output files:]	Specifies output file. (range specification and divided output are enabled)
Information message	Message NOMessage [= <sub>[,…]] <sub>:<error code> [-<error code>]	Output [Show entries:] [Messages:]	Output Not output (error number specification and range specification are enabled)
List file	LISt [= <file name>]	Output [Show entries:] [List file:]	Specifies list file output
List contents	Show [=<sub>[,…]] <sub>:{ SYmbol | Reference | SEction }	Output [Show entries:] [List file:]	 Symbol information Number of references Section information

FOrm

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 **extract** option.
library	Outputs a library file. When **library=s** is specified, a system library is output. When **library=u** is specified, a user library is output. Default is **library=u**.
hexadecimal	Outputs a HEX file. For details of the HEX format, refer to appendix D.2, HEX Format.
stype	Outputs an S-type file. For details of the S-type format, refer to appendix D.1, S-Type 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

Output Format	Specified Option	Enabled File Format	Specifiable Option*[1]
Absolute	strip specified	Absolute file	input, output, show=symbol, reference
	other than above	Object file Relocatable file Binary file Library file	input, library binary, debug/nodebug, sdebug, compress, cpu, start, rom, entry, output, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, rename, delete, define, fsymbol, stack, noprelink, show=symbol, reference
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, rename, delete, noprelink, show=symbol, reference
Object	extract specified	Library file	library, output, show=symbol, reference
Relocate Stype Binary		Object file Relocatable file Binary file Library file	input, library binary, cpu, start, rom, entry, output, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, rename, delete, define, fsymbol, stack, noprelink, record, s9*[2], show=symbol, reference
		Absolute file	input, output, record, s9*[2], show=symbol, reference
Library	strip specified	Library file	library, output, show=symbol, section
	extract specified	Library file	library, output, show=symbol, section
	other than above	Object file Relocatable file	input, library, output, rename, delete, replace, noprelink, show=symbol, section

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

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

DEBug, SDebug, NODEBug

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, this option is interpreted as **debug**.

  If **debug** is specified when two or more files are specified to be output with **output**, this option is 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**, or **extract** or **strip** is specified, this option is unavailable.

REcord

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.

ROm

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 initialization 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, the size of the RAM section of which is 0.
- 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[Show entries for:][Divided output files]

- Command Line Format

  OUtput = <suboption>[,…]

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

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

  Specifies output file name. When **form=absolute**, **hexadecimal**, **stype** or **binary** is specified, two or more files can be specified. An address is specified in 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.

  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**.

Message, NOMessage

Output[Show entries for:][Messages]

- Command Line Format

  Message

  NOMessage [=<suboption>[,…]]

  <suboption> : <error code>[-<error code>]
- 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 because **change_message** is assumed to change 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.

LISt

Output[Show entries for:][List file]

- Command Line Format

  LISt [=<file name>]
- Description

  Specifies a list file output or list file name.

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

SHow

Output[Show entries for:][List file]

- Command Line Format

  SHow [=<suboption>[,…]]

  <suboption>:{ SYmbol | Reference | SEction }
- Description

  Specifies output contents of a list.

  Table 4.5 lists the suboptions.

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

Table 4.5 Suboptions of show Option

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

- Remarks

  When **form=object** or **relocate** is specified, the **show=reference** option is unavailable.

4.2.3 Optimize Tab Options

Table 4.6 Optimize Tab Options

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

OPtimize

Optimize[Optimize:]

- Command Line Format

  Optimize [= <suboption>[,…]]

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

- 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 4.7 shows the suboptions

Table 4.7 Suboptions of Optimize Option

Suboption	Description	Program to be Optimized*			
		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: • Variables defined as const in C/C++ program • Initial value of character string data • Literal constant	O	X	O	X
symbol_delete	Deletes variables/functions that are not referenced. The **entry** 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 **cpu** option should be specified.	X	X	O	O
register	Investigates function calls, relocates registers and deletes redundant register save or restore codes. The **entry** option should be specified.	O	X	O	X
same_code	Creates a subroutine for the same instruction sequence.	O	X	O	X
function_call	Allocates addresses of frequently accessed functions to the range 0 to 0xFF if there is a space. The **cpu** 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, 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, 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.

SAMesize

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

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 (HDI) Ver5.0 or later.
  When profile information file is specified, inter-module optimization according to dynamic information can be performed.
  Table 4.8 shows optimizations influenced by a profile information input.

Table 4.8 Relations Between Profile Information and Optimization

Suboption	Description	Program to be Optimized [*1]			
		SHC	SHA	H8C	H8A
variable_access	Allocates variables that are dynamically accessed frequently first.	X	X	O	O
function_call	Lowers the optimized order of functions 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 **cachesize** option.	O	Δ [*2]	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
  If the **optimize** option is not specified, this option is unavailable.

CAchesize

Optimize[Cache size:]

- Command Line Format
  CAchesize = 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 a size in kbytes and specify a line size in bytes in 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

Optimize[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 size in 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. However, the **void** argument is specified with 'function name ()'.

  Table 4.9 shows the suboptions.

Table 4.9 Suboptions of Show Option

Suboption	Parameter	Description
symbol_forbid	Function name | variable name	Disables optimization with unreferenced symbol deletion
samecode_forbid	Function name	Disables optimization with same-code unification
variable_forbid	Variable name	Disables optimization with using short absolute addressing mode
function_forbid	Function name	Disables optimization with using indirect addressing mode
absolute_forbid	Address [+ size]	Disables optimization with 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.

4.2.4 Section Tab Options

Table 4.10 Section Tab Options

Item	Command Line Format	Dialog Menu	Specification
Section address	STARt = <sub>[,…] <sub>: <section name> [{ : | , }<section name>[,…]] [/<address>]	Section [Relocatable section start address:]	Specifies a section start address.
Symbol address file	FSymbol = <section name>[,…]	Section [Generate external symbol file:]	Outputs externally defined symbol addresses to a definition file.

STARt

Section[Relocatable section start address:]

- Command Line Format

  STARt = <suboption> [,…]

  <suboption> : <section name> [{ : | , } <section name> [,…]] [/ <address>]

- Description

  Specifies the start address of the section. Specifies an address in 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 specified 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 respectively to the addresses starting from 0x100. Both R1 and R2 are allocated to 0x8000.

```
input=a.obj b.obj ;  a.obj references symbols in d.lib and b.obj references 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) and d(P).
```

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

FSymbol

Section[Generate external symbol file:]

- Command Line Format
  FSymbol = <section name> [,…]
- Description
  Outputs externally defined symbols in the specified section to a file in the assembler instruction 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**]
  ```
  ;HITACHI OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
  ;fsymbol = sct2, sct3

  ;SECTION NAME = sct2
   .export _f
  _f: .equ h’00000000
   .export _q
  _q: .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.

4.2.5 Verify Tab Options

Table 4.11 Verify Tab Options

Item	Command Line Format	Dialog Menu	Specification
Address check	CPu = {<cpu information file name> | {ROM|RAM}= <address range>[,…] <address range>: <start address> -<end address>	Verify [CPU information check:]	Specifies a CPU information file. Specifies a specifiable allocation range for section addresses.

CPu

Verify[CPU information check:]

- Command Line Format

  CPu={<cpu information file name>
  | {ROm | RAm} = <address range> [,…]}

  <address range> : <start address> - <end address>

- Description

  Checks section allocation addresses.

  Specify an address range in which a section can be allocated in hexadecimal notation. The ROM/RAM 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.

4.2.6 Other Tab Options

Table 4.12 Other Tab Options

Item	Command Line Format	Dialog Menu	Specification
End code	S9	Other [Miscellaneous options:] [Always output S9 record at the end]	Always outputs the S9 record.
Stack information file	STACk	Other [Miscellaneous options:] [Stack information output]	Outputs a stack use information file.
Symbol name modification	RENname = <sub>[,…] <sub>: {[<file name>] (<name>=<name>[,…]) | [<module name>] (<name>=<name>[,…]) }	Other [User defined options:]	Modifies a symbol name or section name.
Symbol name deletion	DELete = <sub>[,…] <sub>: {<module name> | [<file name>] (<name>[,…]) }	Other [User defined options:]	Deletes a symbol name or section name.
Module replacement	REPlace = <sub>[,…] <sub>: <file> [(<module>[,…])]	Other [User defined options:]	Replaces modules of the same name in a library file.
Module extraction	EXTract = <module>[,…]	Other [User defined options:]	Extracts the specified module in a library file.
Debug information deletion	STRip	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>] [,…]]	Other [User defined options:]	Modifies message levels.
Copyright	LOgo NOLOgo	-	Output Not output

Table 4.12 Other Tab Options (cont)

Item	Command Line Format	Dialog Menu	Specification
Continuation	END	-	Executes option strings already input, inputs continuing option strings and continues processing.
Termination	EXIt	-	Specifies the termination of option input.

S9

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

Other[Miscellaneous options:][Stack information output]

- Command Line Format
  STACk
- Description
  Outputs a stack use information file.
  The file name is <output file name>.sni.
- Remarks
  When **form=object**, **relocate**, **library** or **strip** is specified, this option is unavailable.

REName

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

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 there are several files or modules of the same name, the file that is input first is applied.
  When a symbol is specified to be deleted using this option, the object is not deleted.

- 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

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 with the module **file1.obj**.

  `replace=lib1.lib(mdl1)` ; Replaces with the module **mdl1** in the input library tile **lib1.lib**.
- Remarks

  When **form=object**, **relocate**, **absolute**, **hexadecimal**, **stype**, **binary**, or **extract** or **strip** is specified, this option is unavailable.

EXTract

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

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 from which debug information is to be deleted are backed up in **file1.abk**, **file2.abk**, and **file3.abk**.
- Remarks
  When **form=object**, **relocate**, **hexadecimal**, **stype** or **binary** is specified, this option is unavailable.

CHange_message

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 information and warning messages to error level messages.
  When a message is output, the execution is aborted.

LOgo, NOLOgo

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**.

END

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

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 unavailable.

4.2.7 Subcommand File Option

Table 4.13 Subcommand Tab Option

Item	Command Line Format	Dialog Menu	Specification
Subcommand file	SUbcommand = <file name>	Subcommand [Subcommand file path:]	Specifies an option with a subcommand file

SUbcommand

Subcommand file[Subcommand file path:]

- Command Line Format
  SUbcommand = <file name>
- Description
  Specifies an option 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 a = or a space.
  For the **input** option, suboptions are separated by a space.
  One option is specified on a 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**.

Section 7 Environment Variables

7.1 Environment Variables List

Environment variables are listed in table 7.1.

Table 7.1 Environment Variables

Environment Variable	Description
path	Specifies a storage directory for the compiler. Specification format: PC version: C> path = <compiler path name>; [<previous path name>;...] UNIX C shell: %set path = (<compiler path name> $path) UNIX Bourne shell: %PATH = :<compiler path name> [:<previous path name>...] %export PATH
SHC_LIB	Specifies a directory at which compiler load module and system include file exist. To enter commands using the DOS prompt of the PC version, or for the UNIX version, this environment variable must be specified. Specification format: PC version: C> set SHC_LIB = <compiler file path name> UNIX C shell: %setenv SHC_LIB = <compiler file path name> UNIX Bourne shell: %SHC_LIB = <compiler file path name> %export SHC_LIB
SHCPU	Specifies the CPU type by the compiler or assembler . **cpu** option using environment variables. The following is specified. <CPU>: SH1, SH2, SH2E, SHDSP, SH3, SH3E, SH3DSP, and SH4 When the specification of CPU by SHCPU environment variable and the **cpu** option differs, a warning message is displayed. **cpu** option has priority over SHCPU specification. When SHDSP and SH3DSP are specified for the compiler, SH2 and SH3 are assumed, respectively. Specification format: PC version: C> set SHCPU = <CPU> UNIX C shell: %setenv SHCPU = <CPU> UNIX Bourne shell: %SHCPU = <CPU> %export SHCPU

Table 7.1 Environment Variables (cont)

Environment Variable	Description
SHC_INC*	Specifies a directory at which a system include file exists. A system include file is searched for at a directory specified by **include** option, SHC_INC-specified directory, and system directory (SHC_LIB) in this order. User include files are searched for at the current directory, a directory specified by **include** option, and SHC_INC-specified directory in this order. When the environment variable is not specified, SHC_LIB is assumed for the UNIX version. The PC version does not have default. Specification format: PC version: C> set SHC_INC = <include path name> [;<include path name>;...] UNIX C shell: %setenv SHC_INC = <include path name> [:<include path name>:...] UNIX Bourne shell: % SHC_INC = <include path name> [:<include path name>:...] %export SHC_INC
SHC_TMP	Specifies a directory for a temporary file generated by the compiler. In the PC version, SHC_TMP must be specified so that the DOS prompt can be used to enter commands. For the UNIX version, the directory indicated by the environment variable TMPDIR is used when this environment variable is not specified. If neither SHC_TMP nor TMPDIR is specified, temporary files are generated in /usr/tmp. Specification format: PC version: C> set SHC_TMP = <temporary file path name> UNIX C shell: %setenv SHC_TMP = <temporary file path name> UNIX Bourne shell: %SHC_TMP = <temporary file path name> %export SHC_TMP

Table 7.1 Environment Variables (cont)

Environment Variable	Description
HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	Specifies a default library name for the optimizing linkage editor. Libraries which are specified by a **library** option are linked first. Then, if there is an unresolved symbol, the default libraries are searched in the order 1, 2, 3. Specification format: PC version: C> set HLNK_LIBRARY1 = <library name 1> C> set HLNK_LIBRARY2 = <library name 2> C> set HLNK_LIBRARY3 = <library name 3> UNIX C shell: %setenv HLNK_LIBRARY1 = <library name 1> %setenv HLNK_LIBRARY2 = <library name 2> %setenv HLNK_LIBRARY3 = <library name 3> UNIX Bourne shell: %HLNK_LIBRARY1 = <library name 1> %export HLNK_LIBRARY1 %HLNK_LIBRARY2 = <library name 2> %export HLNK_LIBRARY2 %HLNK_LIBRARY3 = <library name 3> %export HLNK_LIBRARY3
HLNK_TMP	Specifies a directory in which the optimizing linkage editor creates temporary files. If HLNK_TMP is not specified, the temporary files are created in the current directory. Specification format: PC version: C> set HLNK_TMP = <temporary file path name> UNIX C shell: %setenv HLNK_TMP = <temporary file path name> UNIX Bourne shell: %HLNK_TMP = <temporary file path name> %export HLNK_TMP
HLNK_DIR*	Specifies an input file storage directory for the optimizing linkage editor. The search order for files which are specified by an **input** or a **library** option is the current directory then this directory. However, when a wildcard is used in the file specification, only the current directory is searched. Specification format: PC version: C> set HLNK_DIR = <input file path name> [;<input file path name >;...] UNIX C shell: %setenv HLNK_DIR = <input file path name> [:<input file path name >:...] UNIX Bourne shell: %HLNK_DIR = <input file path name> [:<include path name>:...] %export HLNK_DIR

Note: More than one directory can be specified by dividing directories using semicolons or commas (PC version) or colons (UNIX).

7.2 Compiler Implicit Declaration

The following implicit **#define** declarations are made by the compiler according to the option specification and the version.

Table 7.2 Compiler Implicit Declaration

Option	Implicit Declaration
cpu = sh1	#define _SH1
cpu = sh2	#define _SH2
cpu = sh2e	#define _SH2E
cpu = sh3	#define _SH3
cpu = sh3e	#define _SH3E
cpu = sh4	#define _SH4
pic = 1	#define _PIC
endian = big	#define _BIG
endian = little	#define _LIT
double = float	#define _FLT, #define _ _FLT_ _
fpu = single	#define _FPS
fpu = double	#define _FPD
denormalize = on	#define _DON
round = nearest	#define _RON
—	#define _ _HITACHI_VERSION_ _ *1
—	#define _ _HITACHI_ _ *2

Notes: 1. The value of _ _HITACHI_VERSION_ _ is referenced as follows:
C source program: _ _HITACHI_VERSION_ _==aabb
aa: version
bb: revision
Example definition in the compiler:
#define _ _HITACHI_VERSION_ _ 0x0501 //Version 5.1C
#define _ _HITACHI_VERSION_ _ 0x0600 //Version 6.0

2. 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 omitted. The standard file extensions used by the Hitachi Development Environment are shown in table 8.1.

Table 8.1 Standard File Extensions Used by the Hitachi Development Environment

No.	File Extension	Description
1	c	Source program file written in C
2	cpp, cc, cp	Source program file written in C++
3	h	Include file
4	lis, lst*[1]	C source program listing file
5	lis, lpp*[1]	C++ source program listing file
6	p	C source program preprocessor expansion file
7	pp	C++ source program preprocessor expansion file
8	src	Assembly source program file
9	exp	Assembly program preprocessor expansion file
10	lis	Assembly 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 file including definition that was specified by a file with extension td
25	cal	Information files to be called

Note: The extension is lis for UNIX version, lst or lpp for PC version.

Filenames beginning with rti_ are reserved for the system; do not use those files.

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	Tentative-defined variable information file
2	ti	Template information file
3	pi	Parameter information file
4	ii	Instance information file

For details 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 covers the contents and format of the compiler 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

Creating List	Contents	Option Specification Method*1	Default
Source listing information	Source program listing *2	show=source show=nosource	No output
	Source program listing after include file expansion *3	show=include show=noinclude	No output
	Source program listing after macro expansion *3	show=expansion show=noexpansion	No output
Object information	Machine code used in object programs and the assembly code	show=object show=noobject	Output
Statistics information	Total number of errors, number of source program lines, size of each section, and number of symbols	show=statistics show=nostatistics	Output
Command specification information	Displays file names and options specified by the command		Output

Notes: 1. All options are valid when **listfile** option is specified.
2. Source program listings are included in the object information when **source** and **object** suboptions are specified.
3. The source program listing after include file expansion and macro expansion is only valid when **show=source** is specified.

8.2.2 Source Listing

The source listing may be output in two ways. When **show=noinclude, noexpansion** is specified, the unpreprocessed source program is output. When **show=include, expansion** is specified, the preprocessed source program is output. Figures 8.1 and 8.2 show these output formats, respectively. In addition, figure 8.2 shows the differences between them with bold characters.

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

Seq      File   Line    0----+----1----+----2----+----3----+----4----+----5---
  1   m0260.c    1      #include "header.h"
  4   m0260.c    2
  5   m0260.c    3      int sum2(void)
  6   m0260.c    4     {   int j;
  7   m0260.c    5
  8   m0260.c    6      #ifdef SMALL
  9   m0260.c    7           j=SML_INT;
 10   m0260.c    8      #else
 11   m0260.c    9           j=LRG_INT;
 12   m0260.c   10      #endif
 13   m0260.c   11
 14   m0260.c   12           return j;/* continue12345678901234567890123456
 (1)    (2)     (3)        +2345678901234567890 */
                            (7)
 15   m0260.c   13     }
```

Figure 8.1 Source Listing Output for show = noinclude, noexpansion

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

  Seq     File  Line    0----+----1----+----2----+----3----+----4----+----5---
    1   m0260.c    1      #include "header.h"
    2   header.h   1      #define SML_INT        1
    3   header.h   2      #define LRG_INT      100       (4)
    4   m0260.c    2
    5   m0260.c    3      int sum2(void)
    6   m0260.c    4      {   int j;
    7   m0260.c    5
    8   m0260.c    6      #ifdef SMALL
    9   m0260.c    7 X        j=SML_INT;
   10   m0260.c    8 (5)   #else
   11   m0260.c    9 E        j=100;
   12   m0260.c   10 (6)   #endif
   13   m0260.c   11
   14   m0260.c   12         return j;/* continue12345678901234567890123456
   (1)    (2)     (3)       ±2345678901234564890 */
                         (7)
   15   m0260.c   13     }
```

Figure 8.2 Source Listing Output for show=include, expansion

Description:

(1) Listing line number

(2) Source program file name or include file name

(3) Line number in source program or include file

(4) Source program lines resulting from an include file expansion when **show=include** is specified.

(5) Source program lines that are not to be compiled due to conditional compile directives such as #ifdef and #elif being marked with an X when **show=expansion** is specified.

(6) Source program lines containing a macro expansion #define directives being marked with an E when **show=expansion** is specified.

(7) If a source program line is longer than the maximum listing line, the continuation symbol (±) is used to indicate that the source program line is extended over two or more listing lines.

8.2.3 Object Listing

The object listing can be output in two ways. When **show = source**, **object** is specified, the source program is output. When **show = nosource, object** is specified, the source program is not output.

Figures 8.3 and 8.4 show examples of these listings.

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

  SCT   OFFSET   CODE       C LABEL   INSTRUCTION OPERAND     COMMENT
  (1)     (2)     (3)                        (4)                (5)
            m0251.c    1    extern int multipli(int);
            m0251.c    2
            m0251.c    3    int multipli(int x)
  P  00000000               _multipli:                ;function: multipli
                                                      ;frame size=16 (7)
                                                      ;used runtime library name:
                                                      ;__muli       (8)
     00000000   4F22                 STS.L    PR,R15
     00000002   7FF4                 ADD      #-12,R15
     00000004   1F42                 MOV.L    R4,@(8,R15)
           m0251.c    4    {
           m0251.c    5            int i;
           m0251.c    6            int j;
           m0251.c    7
           m0251.c    8            j=1;
     00000006   E201                 MOV       #1,R2
     00000008   2F22                 MOV.L     R2,@R15
           m0251.c    9            for(i=1;i<=x;i++){
     0000000A   E301                 MOV       #1,R3
     0000000C   1F31                 MOV.L     R3,@(4,R15)
     0000000E   A009                 BRA       L213
     00000010   0009                 NOP
     00000012                 L214:
           m0251.c   10                     j*=i;
     00000012   50F1                 MOV.L     @(4,R15),R0
     00000014   61F2                 MOV       @R15,R1
     00000016   D30A                 MOV.L     L216+2,R3       ;_ _muli
     00000018   430B                 JSR       @R3
              .                          .
              .                          .
```

Figure 8.3 Object Listing Output for show = source, object

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

  SCT   OFFSET   CODE       C LABEL   INSTRUCTION OPERAND     COMMENT
  (1)     (2)     (3)                      (4)                 (5)
  P                        ;File m0251.c   ,Line 3    ;block
     00000000               _multipli:       (6)      ;function: multipli
                                                      ;frame size=16 (7)
                                                      ;used runtime library name:
                                                      ;   muli       (8)
     00000000   4F22                 STS.L    PR,@R15
     00000002   7FF4                 ADD      #-12,R15
     00000004   1F42                 MOV.L    R4,@(8,R15)
                             ;File m0251.c   ,Line 4        ;block
                             ;File m0251.c   ,Line 8        ;expression statement
     00000006   E201                 MOV       #1,R2
     00000008   2F22                 MOV.L     R2,@R15
                             ;File m0251.c   ,Line 9        ;for
     0000000A   E301                 MOV       #1,R3
     0000000C   1F31                 MOV.L     R3,@(4,R15)
     0000000E   A009                 BRA       L213
     00000010   0009                 NOP
     00000012                L214:
                             ;File m0251.c   ,Line 9       ;block
                             ;File m0251.c   ,Line 10      ;expression statement
     00000012   50F1                 MOV.L     @(4,R15),R0
     00000014   61F2                 MOV.L     @R15,R1
     00000016   D30A                 MOV.L     L216+2,R3  ;_ _muli
     00000018   430B                 JSR       @R3
               .                         .
               .                         .
```

Figure 8.4 Object Listing Output for show = nosource, object

Description:

(1) Section name (P, C, D, B, C$INIT, and C$VTBL) of each section

(2) Offset address relative to the beginning of each section

(3) Contents of the offset address of each section

(4) Assembly code corresponding to machine language

(5) Comments corresponding to the program (only output when not optimized; however, labels are always output)

(6) Line information of the program (only output when not optimized)

(7) Stack frame size in bytes

(8) Routine name that is being executed

8.2.4 Statistics Information

Figure 8.5 shows an example of statistics information.

```
******** STATISTICS INFORMATION ********


********** ERROR INFORMATION **********                (1)

NUMBER OF ERRORS:                 0
NUMBER OF WARNINGS:               0
NUMBER OF INFORMATIONS:           0

******** SOURCE LINE INFORMATION ********              (2)

COMPILED SOURCE LINE:             13


******** SECTION SIZE INFORMATION ********             (3)

PROGRAM     SECTION(P):               00000044 Byte(s)
CONSTANT    SECTION(C):               00000000 Byte(s)
DATA        SECTION(D):               00000000 Byte(s)
BSS         SECTION(B):               00000000 Byte(s)

TOTAL PROGRAM SIZE:                  00000044 Byte(s)


********** LABEL INFORMATION **********                (4)

NUMBER OF EXTERNAL REFERENCE SYMBOLS:  1
NUMBER OF EXTERNAL DEFINITION SYMBOLS: 1
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:   6
```

Figure 8.5 Statistics Information

Description:

(1) Total number of messages by the level
(2) Number of compiled lines from the source file
(3) Size of each section and total size of sections
(4) Number of external reference symbols, number of external definition symbols, and total number of internal and external labels

Note: NUMBER OF INFORMATIONS in messages by the level ((1) above) is not output when **message** option is not specified. Section size information (3) and label information (4) are not output if an error-level error or a fatal-level error has occurred or when **noobject** option is specified. In addition, section size information (3) is output (indicated as "1") or not output (indicated as "0") according to its specification when **code=asmcode** option is specified.

8.2.5 Command Line Specification

The file names and options specified on the command line when the compiler is invoked are displayed. Figure 8.6 shows an example of command line specification information.

```
*** COMMAND PARAMETER ***

-listfile test.c
```

Figure 8.6 Command Line Specification

8.3 Assembly Listings

This section covers the contents and format of the assembly listing.

8.3.1 Structure of Assembly Listing

Table 8.4 shows the structure and contents of the assembly listing.

Table 8.4 Structure and Contents of Assembly Listing

Creating List	Contents	Option	Default
Source list information	Specifies the source program information	source	Output
Cross reference list information	Specifies the source-program symbol information	cross_ reference	Output
Section information list	Specifies the source-program section information	section	Output

Note: All list options are enabled when **list** option is specified.

8.3.2 Source List Information

The source list information is output. Figure 8.7 shows an example of the source list information.

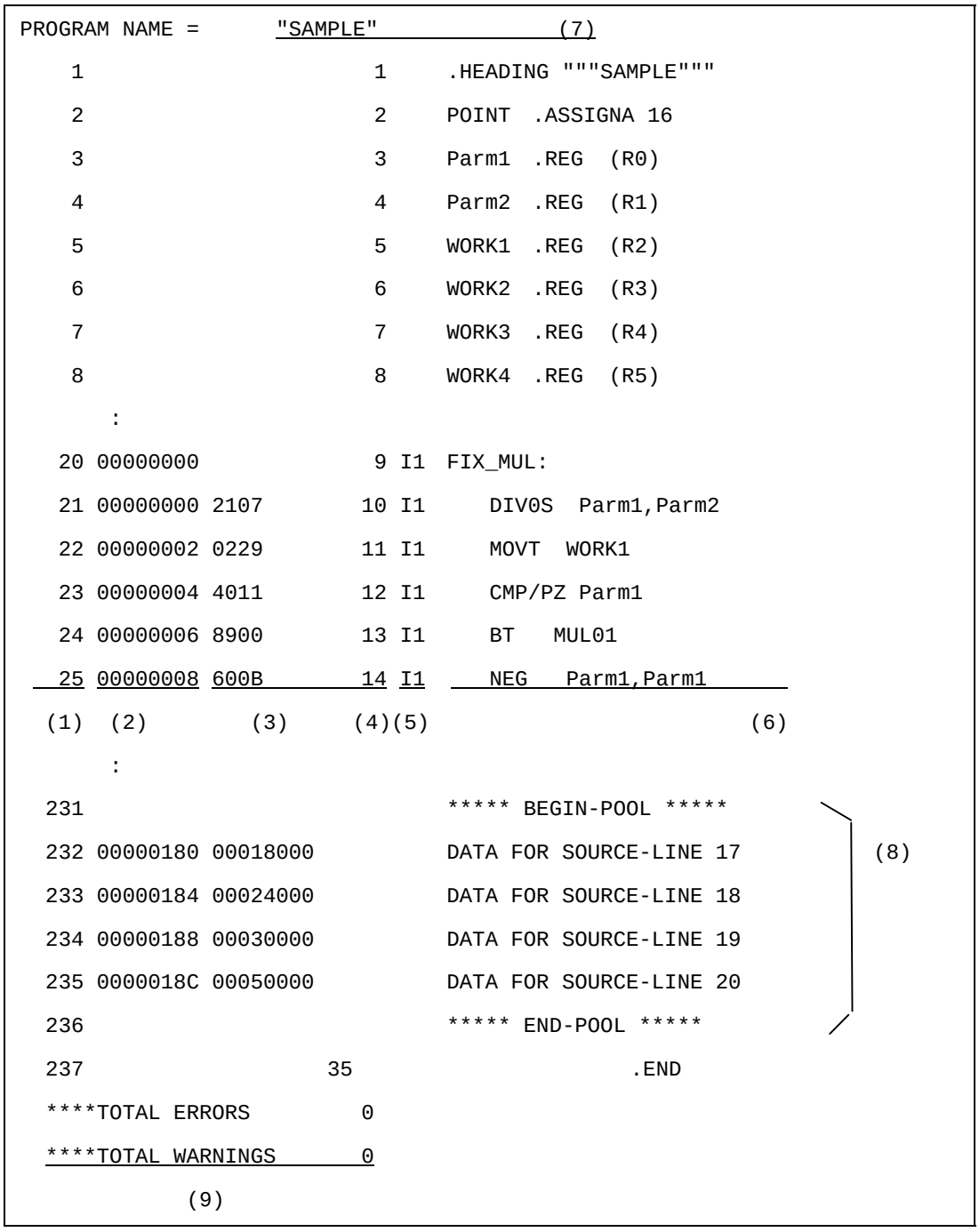


Figure 8.7 Source Program Listing

Description:

(1) Line numbers (in decimal)

(2) The value of the location counter (in hexadecimal)

(3) The object code (in hexadecimal). The size of the reserved area in bytes is listed for areas reserved with the .RES, .SRES, .SRESC, .SRESZ, and .FRES assembler directives.

(4) Source line numbers (in decimal)

(5) Expansion type. Whether the statement is expanded by file inclusion, conditional assembly function, or macro function is listed.

- In: File inclusion (n indicates the nest level).
- C: Satisfied conditional assembly, performed iterated expansion, or satisfied conditional iterated expansion
- M: Macro expansion

(6) The source statements

(7) The header setup with the .HEADING assembler directive.

(8) The literal pool

(9) The total number of errors and warnings. Error messages are listed on the line following the source statement that caused the error.

8.3.3 Cross Reference Listing

The cross reference information is output. Figure 8.8 shows an example of the cross reference information listing.

```
*** CROSS REFERENCE LIST
  NAME       SECTION   ATTR  VALUE              SEQUENCE
  FIX_DIV    SAMPLE          00000088    91*   223
  FIX_MUL    SAMPLE          00000000    19*   218
  MUL01      SAMPLE          0000000A    23    25*
  MUL02      SAMPLE          00000010    26    28*
  MUL03      SAMPLE          00000082    87    89*
  Parm1                 REG               3*   20     22     24     24
                                          28    29     29     31     32
                                          32    35     36     36     38
                                          40    45     49     55     57
                                          59    61     63     65     67
                                          69    71     73     75     77
                                          79    81     83     85     88
                                          88    93     94     99    101
  Parm2                 REG               4*   20     25     27     27
                                          28    31     33     33     35
                                          38    41     43     44     46
                                          48    54     56     58     60
                                          62    64     66     68     70
  (1)        (2)       (3)   (4)                        (5)
```

Figure 8.8 Cross Reference Listing

Description:

(1) The symbol name

(2) The name of the section that includes the symbol (first eight characters)

(3) The symbol attribute

EXPT: Export symbol

IMPT: Import symbol

SCT: Section name

REG: Symbol defined with the .REG assembler directive

FREG: Symbol defined with the .FREG assembler directive

ASGN: Symbol defined with the .ASSIGN assembler directive

EQU: Symbol defined with the .EQU assembler directive

MDEF: Symbol defined two or more times

UDEF: Undefined symbol

No symbol attribute (blank): A symbol other than those listed above

(4) The value of symbol (in hexadecimal)

(5) The list line numbers (in decimal) 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 is output. Figure 8.9 shows an example of the section information output.

```
*** SECTION DATA LIST
  SECTION             ATTRIBUTE     SIZE                START
  SAMPLE              REL-CODE      000000190           ____________
   (1)                 (2)          (3)                   (4)
```

Figure 8.9 Section Information Listing

Description:

(1) The section name

(2) The section type

REL: Relative address section

ABS: Absolute address section

CODE: Code section

DATA: Data section

STACK: Stack section

DUMMY: Dummy section

(3) The section size (in hexadecimal, byte units)

(4) The start address of absolute 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

Creating List	Contents	Suboption	Default
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/end addresses, size, and type	—	Output
Symbol information	Displays static definition symbol address, size, and type in order based on the address.	show = symbol	Not output
	When the **show=reference** 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

Note: The **show** option is valid only when **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 8.10 when **optlnk -sub=test.sub -list -show** is specified.

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

```
*** Options ***

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

Figure 8.10 Option Information Output Example (Linkage Listing)

Description:

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

(2) Subcommand in the test.sub subcommand file

8.4.3 Error Information

Error information outputs an error message as shown in figure 8.11.

```
*** Error information ***

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

Figure 8.11 Error Information Output Example (Linkage Listing)

Description:

(1) Outputs an error message.

8.4.4 Linkage Map Information

Linkage map information outputs section start/end addresses, size, and type in order of addresses in the format shown in figure 8.12.

```
*** Mapping List ***
SECTION                 START       END         SIZE   ALIGN
  (1)                    (2)        (3)          (4)    (5)

P                       00000000    000004d6    4d6    2
C                       000004d6    00000533     5d    2
D                       00000534    0000053c      8    2
B                       0000053c    00004112    3bd6   2
```

Figure 8.12 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. A symbol information output example is shown in figure 8.13.

```
*** Symbol List ***

SECTION = (1)                START          END     SIZE
FILE = (2)                    (3)           (4)      (5)

 SYMBOL                      ADDR           SIZE    INFO           COUNTS     OPT
   (6)                        (7)            (8)     (9)            (10)      (11)

SECTION = P
FILE = test.obj
                            00000000     00000428     428
  _main
                            00000000            2  func ,g        0
  _malloc
                            00000000           32  func ,1        0
FILE=mvn3
                            00000428     00000490      68
  $MVN#3
                            00000428            0  none ,g        0
```

Figure 8.13 Symbol Information Output Example (Linkage Listing)

Description:

(1) Section name
(2) File name
(3) Start address of a section included in the file indicated by (2) file name above
(4) End address of a section included in the file indicated by (2) file name above
(5) Section size of a section included in the file indicated by (2) file name 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.
* 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 8.14.

```
*** Delete Symbols ***

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

Figure 8.14 Symbol Deletion Information Output (Linkage Listing)

Description:

(1) Delete symbol name

(2) Delete symbol size

(3) Delete 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

This information is not output when the SuperH RISC engine microcomputer is used (figure 8.15).

```
*** Variable Accessible with Abs8 ***

SYMBOL                                  SIZE   COUNTS       OPTIMIZE

*** Variable Accessible with Abs16 ***

SYMBOL                                  SIZE   COUNTS       OPTIMIZE
```

Figure 8.15 Output Example of Variable Access Optimization Symbol Information (Linkage Listing)

8.4.8 Function Access Optimization Symbol Information

This information is not output when the SuperH RISC engine microcomputer is used (figure 8.16).

```
*** Function Call ***

SYMBOL                              COUNTS       OPTIMIZE
```

Figure 8.16 Output Example of Function Access Optimization Symbol Information (Linkage Listing)

8.5 Library Listings

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

8.5.1 Structure of Library Listing

Table 8.6 shows the structure and contents of the library listing.

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 the address of which has been determined; it cannot be relocated by the optimizing linkage editor.
- Initial values

  Specifies whether there are initial values at the start of program execution. Data which has initial values and data which does not have initial values cannot be included in the same section. If there is even one initial value, the area without initial values is initialized to zero.
- Write operations

  Specifies whether write operations are or are not possible on 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 memory areas and sections for C/C++ programs and the standard library is described in table 9.1.

Table 9.1 Summary of Memory Area Types and Their Properties

Name	Section		Format Type	Initial Values		Description
	Name	Attribute		Write Operations	Align-ment	
Program area	P*[1]	code	Relative	Yes	4*[2] bytes	Stores machine code
				No		
Constant area	C*[1]	data	Relative	Yes	4 bytes	Stores const-type data
				No		
Initialized data area	D*[1]	data	Relative	Yes	4 bytes	Stores data with initial values
				Yes		
Uninitialized data area	B*[1]	data	Relative	No	4 bytes	Stores data without initial values
				Yes		
GBR section	$G0	data	Relative	Yes	4 bytes	Stores data with initial values specified by #pragma gbr_base. If data does not have initial values, 0 is stored.
				Yes		
GBR section	$G1	data	Relative	Yes	4 bytes	Stores data with initial values specified by #pragma gbr_base1. If data does not have initial values, 0 is stored.
				Yes		
C++ initial processing/ postprocessing data area	C$INIT	data	Relative	Yes	4 bytes	Stores addresses of constructors and destructors called for global class objects
				No		
C++ virtual function table area	C$VTBL	data	Relative	Yes	4 bytes	Stores data for calling the virtual function when a virtual function exists in the class declaration
				No		
Stack area	—	—	Relative	No	4 bytes	Area necessary for program execution (see section 9.2.1 (2), Dynamic Area Allocation)
				Yes		
Heap area	—	—	Relative	No	—	Area used by library functions malloc, realloc, calloc, new (see section 9.2.1 (2), Dynamic Area Allocation)
				Yes		

Notes 1: Section names can be switched in the **section** option or extension #pragma section.
2. Becomes 16 bytes when the **align16** option is specified.

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

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

C program

	Section name
Program area (main() {...})	P
Constant area (c)	C
Initialized data area (a)	D
Uninitialized data area (b)	B

Areas generated by the compiler and stored data

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

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

C++ program

	Section name
Program area (f() {...})	P
Constant area (c)	C
Initialized data area (d)	D
Uninitialized data areas (a,b)	B
Initial processing/postprocessing data areas (&A::A, &A::~A)	C$INIT

Areas generated by the compiler and stored data

9.1.3 Assembly Program Sections

In assembly programs, the .section directives are used to begin sections and declare attributes and formats. The format for declaration of the .section directives is as follows; for details refer to the references in section 11.4, Assembler Directives.

```
.section <section name>[, <section attribute> [, <format type>]]
```

<format type>: In the case of a relative address section, align = < boundary alignment>
In the case of an absolute address section, locate = <address value>

Example: An example of an assembly program section declaration appears below.

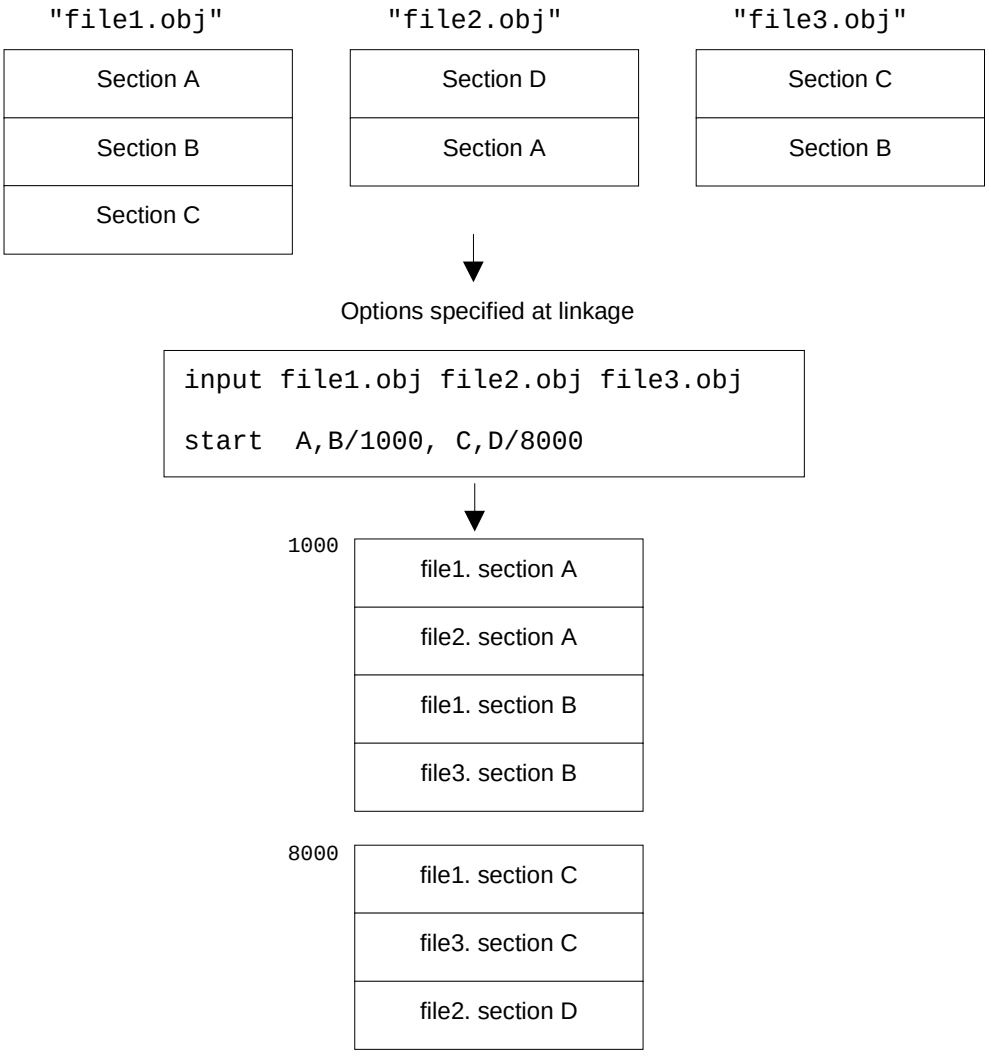
```
            .CPU        SH2
            .OUTPUT     DBG
SIZE:       .EQU        8

            .SECTION    A,CODE,ALIGN=4                  ;(1)
START:
            MOV.L       LITERAL,R0
            MOV.L       LITERAL+4,R1
            MOV.L       #SIZE,R2
LOOP:
            CMP/PL      R2
            BF          EXIT
            MOV.B       @R0+,R3
            MOV.B       R3,@R1
            ADD         #-1,R2
            ADD         #1,R1
            BRA          LOOP
            NOP
EXIT:
            SLEEP
            NOP
LITERAL:
            .DATA.L     CONST
            .DATA.L     DATA
;
            .SECTION    B,DATA,LOCATE=H'00002000        ;(2)
CONST:
            .DATA.B     H'01,H'02,H'03,H'04,H'05,H'06,H'07,H'08
;
            .SECTION    C,STACK,ALIGN=4                 ;(3)
DATA:
            .RES.B      8
            .END
```

(1) Declares a code section with section name A, boundary alignment 4, relative address format.
(2) Declares a data section with section name B, allocated address H'2000, absolute address format.
(3) Declares a stack section with section name C, boundary alignment 4, relative address format.

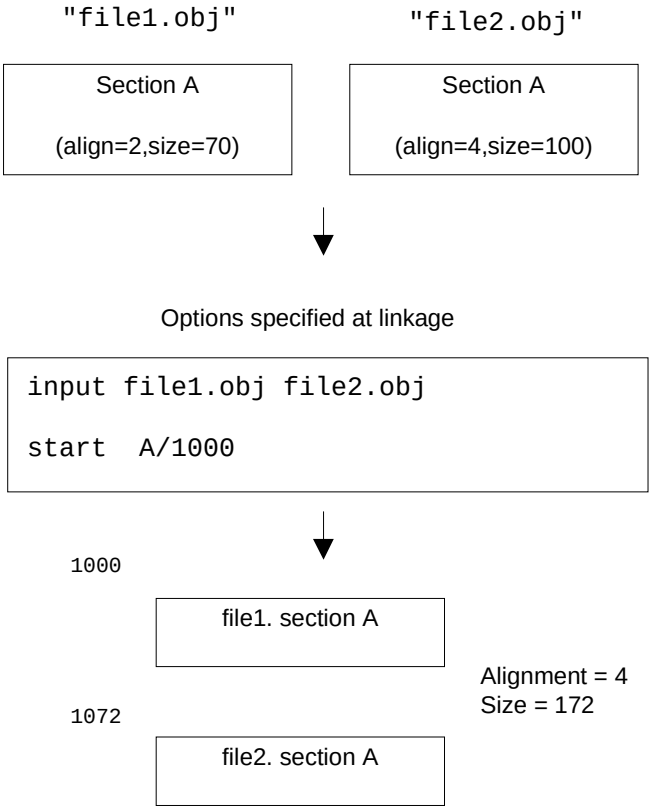
9.1.4 Joining Sections

The optimizing linkage editor joins the same sections within input object programs, and allocates addresses specified by the **start** option.

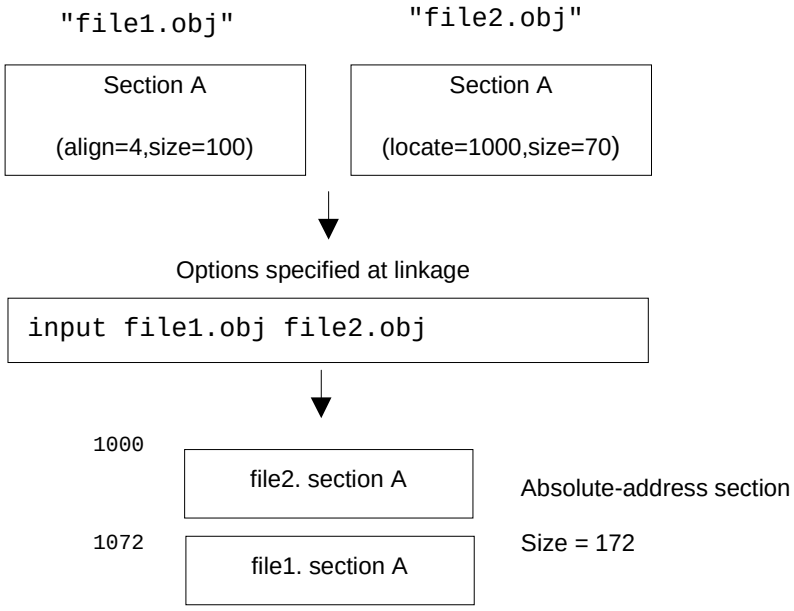
(1) The same section names in different files are allocated contiguously in the order of file input.



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

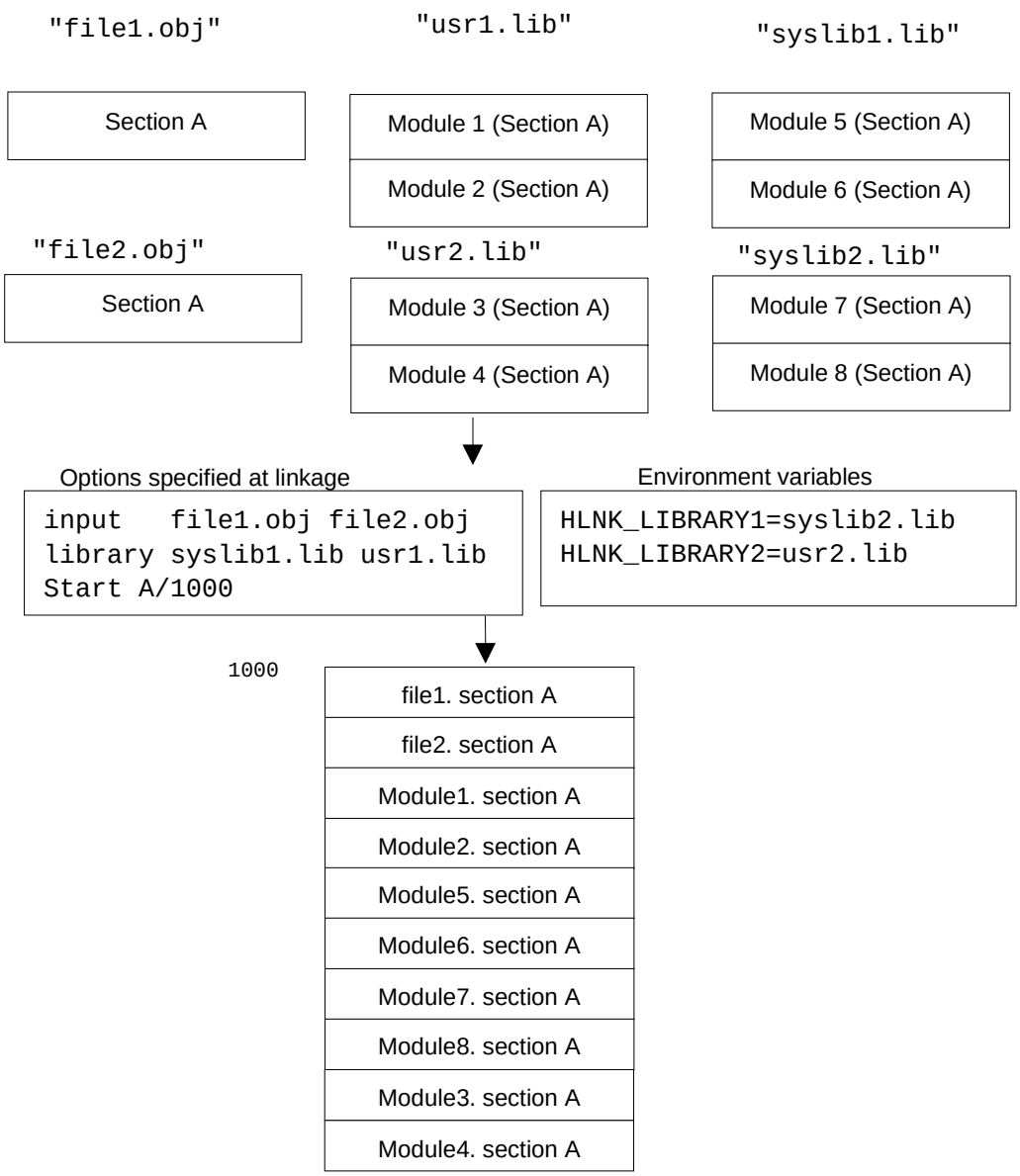


(3) When sections with the same name include both absolute-address and relative-address formats, relative-address objects are joined following absolute-address objects. Even when relocatable file output is specified(**form=relocate** option), the section in question becomes an absolute-address section.



(4) Rules for the order of joining objects within the same section name are as follows.

a. Order specified by the **input** option or input files on the command line
b. Order specified for the user library by the library option and order of input of modules within the library
c. Order specified for the system library by the **library** option and order of input of modules within the library
d. Order specified for libraries by environment variables (HLNK_LIBRARY1 to 3) and order of input of modules within the library


"file1.obj"
"usr1.lib"
"syslib1.lib"
Section A
Module 1 (Section A)
Module 2 (Section A)
Module 5 (Section A)
Module 6 (Section A)
"file2.obj"
"usr2.lib"
"syslib2.lib"
Section A
Module 3 (Section A)
Module 4 (Section A)
Module 7 (Section A)
Module 8 (Section A)
Options specified at linkage
input file1.obj file2.obj
library syslib1.lib usr1.lib
Start A/1000
Environment variables
HLNK_LIBRARY1=syslib2.lib
HLNK_LIBRARY2=usr2.lib
1000
file1. section A
file2. section A
Module1. section A
Module2. section A
Module5. section A
Module6. section A
Module7. section A
Module8. section A
Module3. section A
Module4. section A

9.2 Creation of Initial Setting Programs

Here methods of installing embedded programs for systems employing the SuperH RISC engine microcomputers are explained.

To install an embedded 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 prepared.

When using C library functions for program termination (the exit, atexit, abort functions), these functions must be prepared 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, preparation of low-level interface routines, and examples of preparation of functions for termination processing are also explained.

9.2.1 Memory Allocation

To install an object program generated by the compiler on a system, determine the size of each memory area, and allocate the areas appropriately to the memory addresses.

Some memory areas, such as the area used to store machine code and the area used to store data declared using external definitions or static data members, are allocated statically. Other memory areas, such as the stack area, are allocated dynamically.

This section describes how to allocate each area in memory.

(1) Static Memory Allocation

(a) Contents of static memory

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, C++ initialization 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 library functions used by the C/C++ program.

After linking object programs, the sizes of each section, including libraries, are output to the linkage map information within the linkage listing, and so the size of static memory can be determined.

Figure 9.1 shows an example of linkage map information within the linkage listing.

```
    * * * Mapping list * * *
SECTION                  START              END              SIZE     ALIGN
  (1)                      (2)                (3)              (4)      (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 within the Linkage Listing

Section sizes of compilation units and assembly units are output to the compile list statistical information and assembly list section information. An example of compile list statistical information is shown in figure 9.2, and an example of assembly list section information is shown in figure 9.3.

```
* * * * * * * SECTION SIZE INFORMATION * * * * * * *
PROGRAM SECTION(P)   :0000004A Byte(s)
CONSTANT SECTION(C)  :00000018 Byte(s)
DATA SECTION(D)      :00000004 Byte(s)
BSS SECTION(B)       :00000004 Byte(s)

TOTAL PROGRAM SIZE  :0000006A Byte(s)
```

Figure 9.2 Example of Compile List Statistical 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 section sizes for files is the size of static area.

If the standard library is used, add the memory area used by the library functions to the memory area size of each section. The standard library includes C library functions based on the C language specifications and arithmetic routines (runtime routines) required for C/C++ program execution. Accordingly, the standard library may be necessary even if library functions are not used in the C/C++ source program.

The runtime routines used by the C/C++ programs are output as external reference symbols in the assembly programs generated by the compiler (**code=asmcode** option). The user can see the runtime routine names used in the C/C++ programs through the external reference symbols. The runtime routine names can also be checked by the use of the **listfile** option.

The following shows the examples.

- C/C++ program

```
f( int a, int b)
{
   a /= b;
   return a;
}
```

- Assembly program output by the compiler

```
       .IMPORT       _ _divls       ;(External reference declaration of runtime routine)
       .EXPORT       _f
       .SECTION      P,CODE,ALIGN=4
_f:                                 ;function: f
                                    ;frame size=4
                                    ;used runtime library name:
                                    ;_ _divls
       STS.L  PR,@-R15
       MOV    R5, R0
       MOV.L  L218,R3               ; _ _divls
       JSR    @R3
       MOV    R4, R1
       LDS.L  @R15+, PR
       RTS
       NOP
L218:
       .DATA.L       _ _divls
       .END
```

(c) ROM, RAM allocation

When preparing a program for systems with 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 preparing the sections of a C/C++ program for systems with ROM, sections are allocated to ROM or to RAM as follows.

- Program area (section P) ROM
- Constant areas (sections C, $G0, $G1) ROM
- Uninitialized data areas (sections B, $G0, $G1) RAM
- Initialized data areas (sections D, $G0, $G1) ROM, RAM (see (d) below)
- Initialization processing/postprocessing data area*[1] (section C$INIT) ROM
- Virtual function table area*[2] (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 has a virtual function declaration
3. $G0 and $G1 can be assigned to only one of the above areas.

(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 linkage 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 same memory area both in ROM and in RAM. For an example of this, refer to "(e) Example of memory allocation and address specification at linkage" below. Initial settings for sections to be copied from ROM to RAM are explained in section 9.2.2 (2), Initial Settings.

(e) Example of memory allocation and address specification at linkage

When creating an absolute load module, addresses of allocated area are specified for each section using an optimizing linkage editor option or a subcommand. Below, examples of static memory allocation and of address specification at linkage are explained.

Figure 9.4 shows an example of allocation of static memory areas.

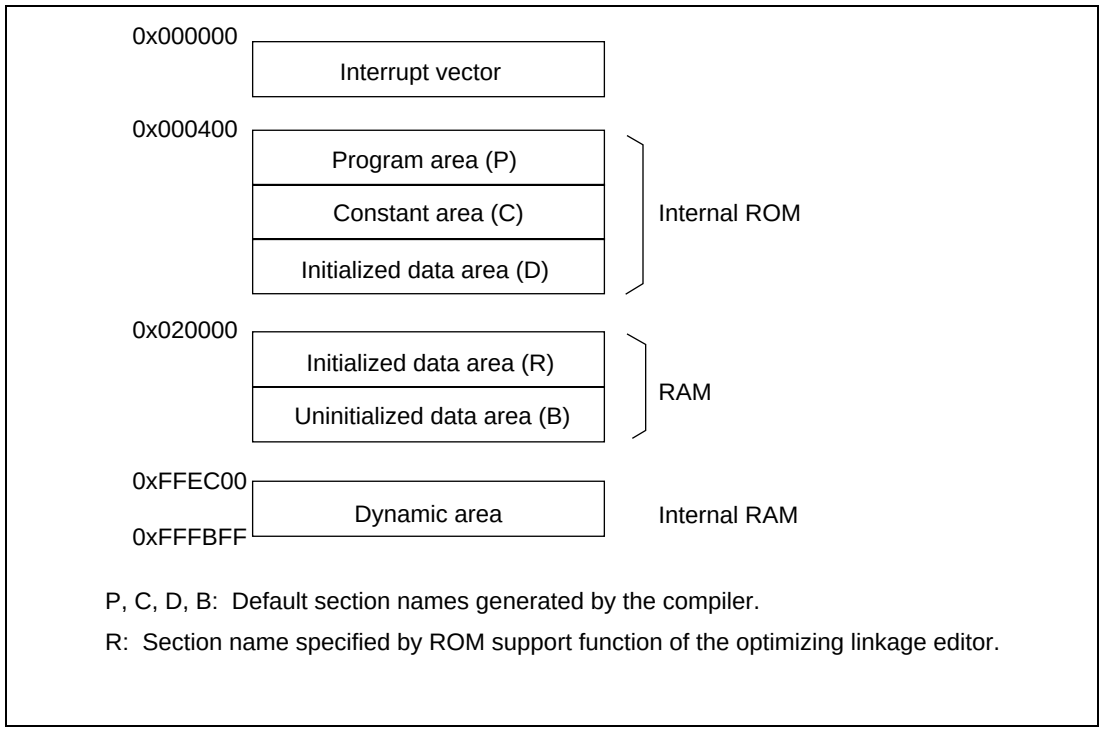


Figure 9.4 Example of Static Memory Allocation

When allocating memory as shown in figure 9.4, the following subcommands are specified at linkage.

```
ROMΔD/R                          ...[1]
STARTΔP,C,D/400,R,B/20000        ...[2]
```

Explanation [1] Space for section R, of size equal to section D, is allocated in the output load module. When symbols allocated to section D are referenced, relocation is performed as if the addresses are in section R. Section D and section R are the names of initialized data sections written to 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 RAM address 0x20000.

(2) Dynamic memory 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)

(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 analysis tool. For details of use of the stack analysis tool, see section 6, Stack Analysis Tool Manipulation.

The stack area used by an assembly program cannot be calculated by the stack analysis tool. 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 analysis tool.

- Stack Usage Calculation of the C/C++ Program

  The stack area used in C/C++ programs is allocated each time a function is called and is deallocated each time a function is returned. The total stack area size is calculated based on the stack size used by each function and the nesting of function calls.

- Stack Area Used by Each Function

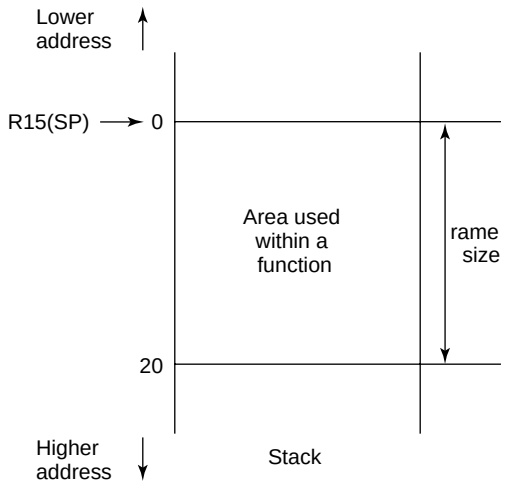
  The object list (frame size) output by the compiler determines the stack size used by each function. The following example shows the object list, stack allocation, and stack size calculation method.

- Example

  The following shows the object list and stack size calculation in a C program.
  The same calculation method is also applicable to C++ programs.

```
extern int h(char, int *, double );
int h(char a, register int *b, double c)
{
    char *d;

    d= &a;
    h(*d,b,c);
    {
        register int i;

        i= *d;
        return i;
    }
}
```

```
************ OBJECT LISTING ************
 FILE NAME: m0251.c

 SCT   OFFSET      CODE    C LABEL   INSTRUCTION OPERAND  COMMENT
 P
       00000000            _h:                            ;function: h
                                                          ;frame size=20
       00000000    2FE6              MOV.L  R14,@-R15
       00000002    4F22              STS.L  PR,@-R15
                   :
```



The size of the stack area used by a function is equal to frame size. Therefore, in the above example, the stack size used by the function h is 20 bytes which is shown as frame size = 20 in COMMENT of the object listing.

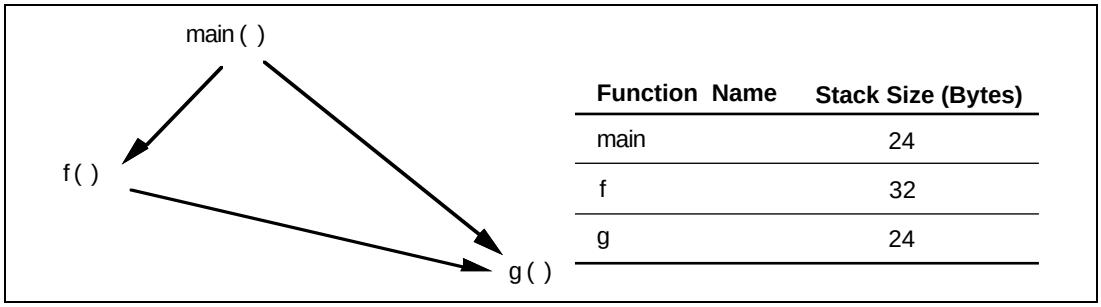
For details on the parameter allocated to the parameter area on the stack, refer to section 9.3.2 (4), Setting and Referencing Parameters and Return Values.

- Stack size calculation

  The following example shows a stack size calculation depending on the function call nesting.

- Example

  Figure 9.5 shows the function call nestings and stack size.



Function Name	Stack Size (Bytes)
main	24
f	32
g	24

Figure 9.5 Nested Function Calls and Stack Size

If function g is called via function f, the stack area size is calculated according to the formula listed in table 9.2.

Table 9.2 Stack Size Calculation Example

Call Route	Sum of Stack Size (Bytes)
main (24) → f (32) → g (24)	80
main (24) → g (24)	48

As can be seen from table 9.2, the maximum size of stack area required for the longest function calling route should be determined (80 bytes in this example) and at least this size of memory should be allocated in RAM.

Note: If recursive calls are used in the C/C++ source program, first determine the stack area required for a recursive call, and then multiply the size with the maximum level of recursive calls.

(c) Heap Area

The total heap area required is equal to the sum of the areas to be allocated by memory management library functions (calloc, malloc, realloc, or new) in the C/C++ program. Four

bytes must be added for one call because a 4-byte management area is used every time a memory management library function allocates an area.

The compiler controls heap area in units of 1024 bytes. Area size allocated for the heap area (HEAPSIZE) is calculated by the following equation.

HEAPSIZE = 1024 × n (n ≥ 1)

(Area size allocated by the memory control library) + control area size ≤ HEAPSIZE

An I/O library function uses memory management library functions for internal processing. The size of the area allocated in an I/O is determined by the following formula: 516 bytes × (maximum number of simultaneously open files)

Note: Areas released by the free or delete function, which is a memory management library function, can be reused. However, since these areas are often fragmented (separated from one another), a request to allocate a new area may be rejected even if the net size of the free areas is sufficient. To prevent this, take note of the following:

1. If possible, allocate the largest area first after program execution is started.
2. If possible, make the data area size to be reused constant.

- Rules for Allocating Dynamic Area

  The dynamic area is allocated to RAM.

  The stack area is determined by specifying the highest address of the stack to the vector table, and refer to it as SP (stack pointer). Since the interrupt operation of the SH-3, SH-3E, and SH-4 differ from that of the SH-1, SH-2, and SH-2E, interrupt handlers are necessary.

  The heap area is determined by the initial settings of the low-level interface routine (sbrk).

  For details on stack and heap areas, refer to section 9.2.2 (1), Vector Table Setting (VEC_TBL), and section 9.2.2 (6), Creating Low-Level Interface Routine, 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 such a program.

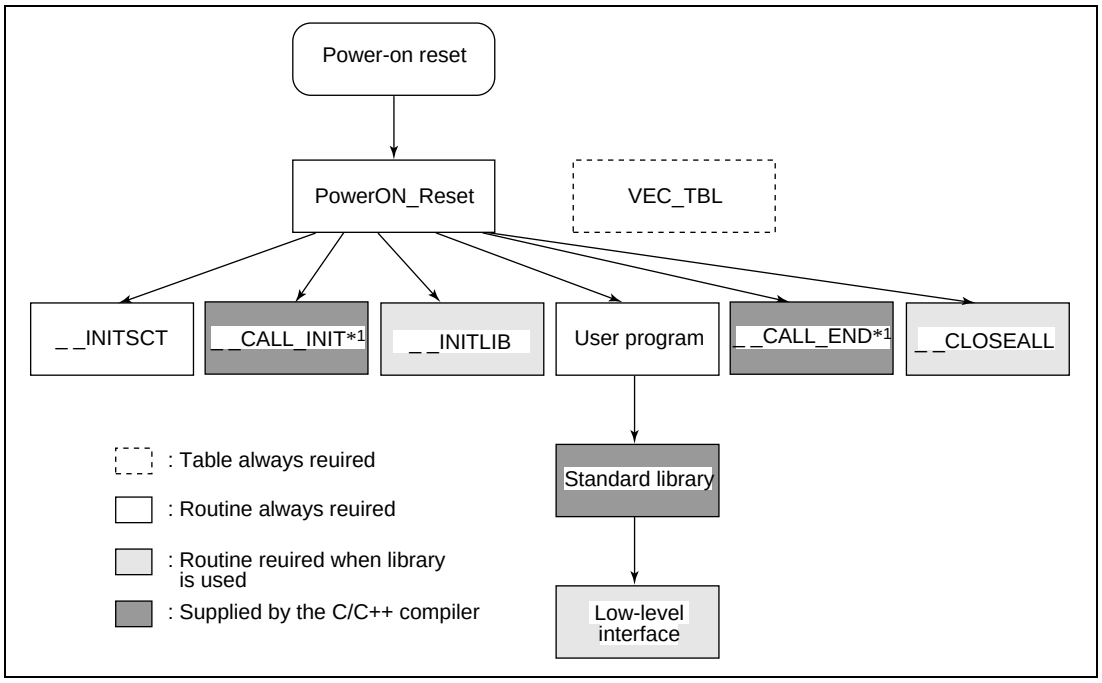


Figure 9.6 Example of Program Structure

Note: Necessary when there is a global class object declaration in the C++ program.

The components are explained below.

- Vector Table Setting (VEC_TBL)

  Sets the vector table to initiate register initialization program (PowerON_Reset) and set the stack pointer (SP) at power-on reset. Since the interrupt of the SH-3, SH-3E, and SH-4 differ from the SH-1, SH-2, and SH-2E, interrupt handlers are necessary.
- Initialization (PowerON_Reset)

  Initializes registers and sequentially calls the initialization routines.
- Initializing Sections (_ _INITSCT)

  Clears non-initialized data area with zeros and copies the initialized data area in ROM to RAM.
- Initializing Library Functions (_ _INITLIB)

  Initializes library functions required to be initialized; especially, prepares standard I/O functions.
- Closing Files (_ _CLOSEALL)

  Closes all files with open status.

- Low-Level Interface Routines

  Interfaces library functions and user system when standard I/O and memory management library functions are used.
- Global Class Object Initial Processing (_ _CALL_INIT)

  Calls a constructor of a class object that is declared as global.
- Global Class Object Post-Processing (_ _CALL_END)

  Calls a destructor of a global class object after the main function is executed.

Implementation of the above routines is described below.

(1) Vector Table Setting (VEC_TBL)

To call register initialization routine PowerON_Reset at power-on reset, specify the start address of function PowerON_Reset at address 0 in the vector table. Also to specify the SP, specify the highest address of the stack to address H'4. Since the interrupt operation of the SH-3, SH-3E, and SH-4 differ from those of the SH-1, SH-2, and SH-2E, interrupt handlers are necessary. When the user system implements interrupt handling, interrupt vector settings are also performed in this component. The coding example of VEC_TBL is shown below.

Example 1 Vector Table for SH-1, SH-2, SH-2E:

```
#pragma interrupt (IRQ0)

extern void PowerON_Reset_PC(void);
extern void PowerON_Reset_SP(void);
extern void Manual_Reset_PC(void);
extern void Manual_Reset_SP(void);

extern void IRQ0(void);

#pragma section VECTTBL    /* Outputs the RESET_Vectors to the CVECTTBL section */
                           /* by #pragma section declaration                    */
                           /* Allocates the CVECTTBL section to address 0x0     */
                           /* by the start option at linkage                    */
void (*const RESET_Vectors[])(void)={
    (void*) PowerON_Reset_PC,
    (void*) PowerON_Reset_SP,
    (void*) Manual_Reset_PC,
    (void*) Manual_Reset_SP
};

#pragma section VECT2      /* Outputs the vec_table2 to the CVECT2 section  */
                           /* by #pragma section declaration */
                           /* Allocates the CVECT2 section to the specified */
                           /* address by the start address at linkage       */

void (*const vec_table2[])(void)={IRQ0};
```

Example 2 Interrupt Handler when Bank 0 is Used in the Program (SH7708)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          env.inc                                          ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

EXPEVT:
        .EQU    H'FFFFFFD4
INTEVT:
        .EQU    H'FFFFFFD8

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                         vect.inc                                          ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SR_Init:
        .EQU    B'00000000000000000000000011110000

;<<VECTOR DATA START (POWER ON RESET)>>
        ;H'000 Power On Reset
        .GLOBAL PowerON_Reset
;<<VECTOR DATA END (POWER ON RESET)>>
;<<VECTOR DATA START (MANUAL RESET)>>
        ;H'020 Manual Reset
        .GLOBAL Manual_Reset
;<<VECTOR DATA END (MANUAL RESET)>>
        ;H'040 TLB miss/invalid (load)
        .GLOBAL INT_TLBMiss_Load
        ;H'060 TLB miss/invalid (store)
        .GLOBAL INT_TLBMiss_Store
        ;H'080 Initial page write
        .GLOBAL INT_TLBInitial_Page
        ;H'0A0 TLB protect (load)
        .GLOBAL INT_TLBProtect_Load
        ;H'0C0 TLB protect (store)
        .GLOBAL INT_TLBProtect_Store
        ;H'0E0 Address error (load)
        .GLOBAL INT_Address_load
        ;H'100 Address error (store)
        .GLOBAL INT_Address_store
        ;H'120 Reserved
        .GLOBAL INT_Reserved1
        ;H'140 Reserved
        .GLOBAL INT_Reserved2
        ;H'160 TRAPA
        .GLOBAL INT_TRAPA
        ;H'180 Illegal code
        .GLOBAL INT_Illegal_code
        ;H'1A0 Illegal slot
        .GLOBAL INT_Illegal_slot
        ;H'1C0 NMI
        .GLOBAL INT_NMI
        ;H'1E0 User breakpoint trap
        .GLOBAL INT_User_Break
        ;H'200 External hardware interrupt
        .GLOBAL INT_Extern_0000
```

```
;H'220 External hardware interrupt
.GLOBAL INT_Extern_0001
;H'240 External hardware interrupt
.GLOBAL     _INT_Extern_0010
;H'260 External hardware interrupt
.GLOBAL     _INT_Extern_0011
;H'280 External hardware interrupt
.GLOBAL     _INT_Extern_0100
;H'2A0 External hardware interrupt
.GLOBAL     _INT_Extern_0101
;H'2C0 External hardware interrupt
.GLOBAL     _INT_Extern_0110
;H'2E0 External hardware interrupt
.GLOBAL     _INT_Extern_0111
;H'300 External hardware interrupt
.GLOBAL     _INT_Extern_1000
;H'320 External hardware interrupt
.GLOBAL     _INT_Extern_1001
;H'340 External hardware interrupt
.GLOBAL     _INT_Extern_1010
;H'360 External hardware interrupt
.GLOBAL     _INT_Extern_1011
;H'380 External hardware interrupt
.GLOBAL     _INT_Extern_1100
;H'3A0 External hardware interrupt
.GLOBAL     _INT_Extern_1101
;H'3C0 External hardware interrupt
.GLOBAL     _INT_Extern_1110
;H'3E0 External hardware interrupt
.GLOBAL     _INT_Extern_1111
;H'400 TMU0 TUNI0
.GLOBAL     _INT_Timer_Under_0
;H'420 TMU1 TUNI1
.GLOBAL     _INT_Timer_Under_1
;H'440 TMU2 TUNI2
.GLOBAL     _INT_Timer_Under_2
;H'460 TMU2 TICPI2
.GLOBAL     _INT_Input_Capture
;H'480 RTC ATI
.GLOBAL     _INT_RTC_ATI
;H'4A0 RTC PRI
.GLOBAL     _INT_RTC_PRI
;H'4C0 RTC CUI
.GLOBAL     _INT_RTC_CUI
;H'4E0 SCI ERI
.GLOBAL     _INT_SCI_ERI
;H'500 SCI RXI
.GLOBAL     _INT_SCI_RXI
;H'520 SCI TXI
.GLOBAL     _INT_SCI_TXI
;H'540 SCI TEI
.GLOBAL     _INT_SCI_TEI
;H'560 WDT ITI
.GLOBAL     _INT_WDT
;H'580 REF RCMI
.GLOBAL     _INT_REF_RCMI
```

```
        ;H'5A0 REF ROVI
        .GLOBAL     _INT_REF_ROVI

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                       vhandler.src                                        ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        .INCLUDE       "env.inc"
        .INCLUDE       "vect.inc"

IMASKclr:
        .EQU    H'FFFFFF0F
RBBLclr:
        .EQU    H'CFFFFFFF
MDRBBLset:
        .EQU    H'70000000

        .IMPORT RESET_Vectors
        .IMPORT INT_Vectors
        .IMPORT INT_MASK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                     macro definition                                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .MACRO  PUSH_EXP_BASE_REG
        STC.L   SSR,@-R15       ; save SSR
        STC.L   SPC,@-R15       ; save SPC
        STS.L   PR,@-R15        ; save CONTEXT REGISTERS
        STC.L   R7_BANK,@-R15
        STC.L   R6_BANK,@-R15
        STC.L   R5_BANK,@-R15
        STC.L   R4_BANK,@-R15
        STC.L   R3_BANK,@-R15
        STC.L   R2_BANK,@-R15
        STC.L   R1_BANK,@-R15
        STC.L   R0_BANK,@-R15
        .ENDM
;
        .MACRO POP_EXP_BASE_REG
        LDC.L   @R15+,R0_BANK   ; RECOVER REGISTERS
        LDC.L   @R15+,R1_BANK
        LDC.L   @R15+,R2_BANK
        LDC.L   @R15+,R3_BANK
        LDC.L   @R15+,R4_BANK
        LDC.L   @R15+,R5_BANK
        LDC.L   @R15+,R6_BANK
        LDC.L   @R15+,R7_BANK
        LDS.L   @R15+,PR
        LDC.L   @R15+,SPC
        LDC.L   @R15+,SSR
        .ENDM
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          reset                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .SECTION    RSTHandler,CODE
_ResetHandler:
        MOV.L   #EXPEVT,R0
        MOV.L   @R0,R0
        SHLR2   R0
        SHLR    R0
        MOV.L   #_RESET_Vectors,r1
        ADD     R1,R0
        MOV.L   @R0,R0
        JMP     @R0
        NOP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    exceptional interrupt                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .SECTION       INTHandler,CODE
        .EXPORT INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L   #EXPEVT,R0              ; set event address
        MOV.L   @R0,R1                  ; set exception code
        MOV.L   #_INT_Vectors,R0        ; set vector table address
        ADD     #-(H'40),R1             ; exception code - H'40
        SHLR2   R1
        SHLR    R1
        MOV.L   @(R0,R1),R3             ; set interrupt function addr
;
        MOV.L   #_INT_MASK,R0           ; interrupt mask table addr
        SHLR2   R1
        MOV.B   @(R0,R1),R1             ; interrupt mask
        EXTU.B  R1,R1
;
        STC     SR,R0                   ; save SR
        MOV.L   #(RBBLclr&IMASKclr),R2
                                        ; RB,BL,mask clear data
        AND     R2,R0                   ; clear mask data
        OR      R1,R0                   ; set interrupt mask
        LDC     R0,SSR                  ; set current status
;
        LDC.L   R3,SPC
        MOV.L   #__int_term,R0          ; set interrupt terminate
        LDS     R0,PR
;
        RTE
        NOP
;
        .POOL
;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    Interrupt terminate                                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .ALIGN  4
__int_term:
        MOV.L   #MDRBBLset,R0  ; set MD,BL,RB
        LDC.L   R0,SR
;
        POP_EXP_BASE_REG
;
        RTE                    ; return
        NOP
;
        .POOL
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    TLB miss interrupt                                       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .ORG    H'300
_TLBmissHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L   #EXPEVT,R0             ; set event address
        MOV.L   @R0,R1                 ; set exception code
        MOV.L   #_INT_Vectors,R0       ; set vector table address
        ADD     #-(H'40),R1            ; exception code - H'40
        SHLR2   R1
        SHLR    R1
        MOV.L   @(R0,R1),R3            ; set interrupt function addr
;
        MOV.L   #_INT_MASK,R0          ; interrupt mask table addr
        SHLR2   R1
        MOV.B   @(R0,R1),R1            ; interrupt mask
        EXTU.B  R1,R1
;
        STC     SR,R0                  ; save SR
        MOV.L   #(RBBLclr&IMASKclr),R2
                                       ; RB,BL,mask clear data
        AND     R2,R0                  ; clear mask data
        OR      R1,R0                  ; set interrupt mask
        LDC     R0,SSR                 ; set current status
;
        LDC.L   R3,SPC
        MOV.L   #__int_term,R0         ; set interrupt terminate
        LDS     R0,PR
;
        RTE
        NOP
;
        .POOL
;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                           IRQ                                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .ORG    H'500
_IRQHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L   #INTEVT,R0              ; set event address
        MOV.L   @R0,R1                  ; set exception code
        MOV.L   #_INT_Vectors,R0        ; set vector table address
        ADD     #-(H'40),R1             ; exception code - H'40
        SHLR2   R1
        SHLR    R1
        MOV.L   @(R0,R1),R3             ; set interrupt function addr
;
        MOV.L   #_INT_MASK,R0           ; interrupt mask table addr
        SHLR2   R1
        MOV.B   @(R0,R1),R1             ; interrupt mask
        EXTU.B  R1,R1
;
        STC     SR,R0                   ; save SR
        MOV.L   #(RBBLclr&IMASKclr),R2
                                        ; RB,BL,mask clear data
        AND     R2,R0                   ; clear mask data
        OR      R1,R0                   ; set interrupt mask
        LDC     R0,SSR                  ; set current status
;
        LDC.L   R3,SPC
        MOV.L   #__int_term,R0          ; set interrupt terminate
        LDS     R0,PR
;
        RTE
        NOP
;
        .POOL
        .END
```

Note: Do not link the function by which #pragma interrupt has been specified.

(2) Initialization (PowerON_Reset)

When library functions are used, sequentially calls the initialization routine _ _INITLIB and file closing routine _ _CLOSEALL. The coding example of PowerON_Reset is shown below. Since the interrupt operation of the SH-3, SH-3E, and SH-4 differ from those of the SH-1, SH-2, and SH-2E, interrupt handlers are necessary.

Example:

```
SR_Init:      .EQU       B'00000000000000000000000011110000
INT_OFFSET    .EQU       H'10
```

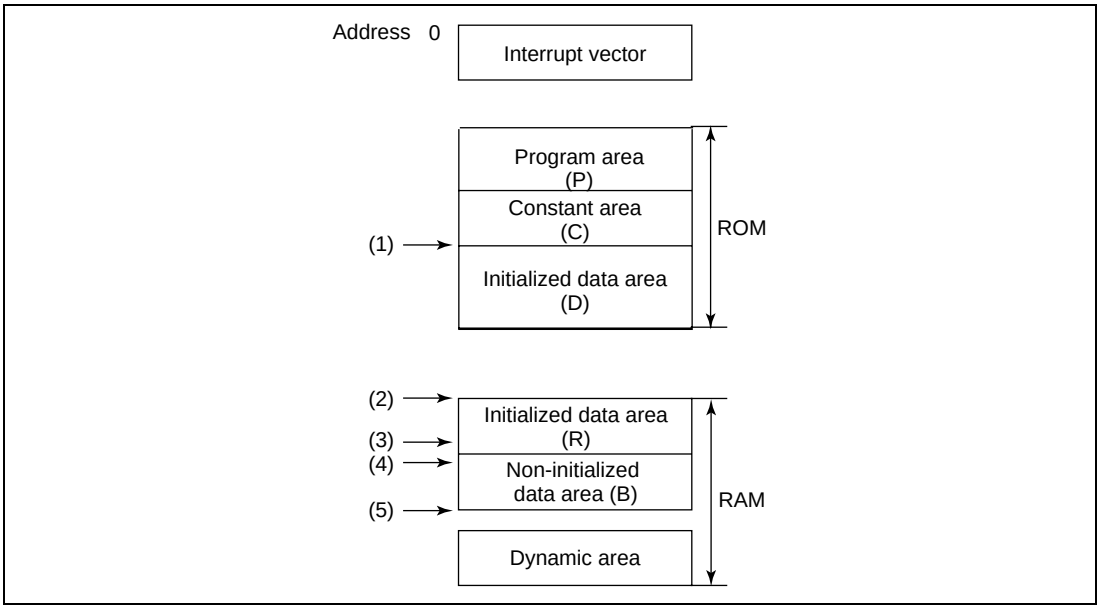
```
        .IMPORT     _ _INITSCT
        .IMPORT     _INT_Vectors

        .IMPORT     _ _INIT_IOLIB
        .IMPORT     _ _CLOSEALL

        .IMPORT     _ _INIT_OTHERLIB

        .IMPORT     _ _CALL_INIT
        .IMPORT     _ _CALL_END

        .IMPORT     _main

        .EXPORT     _PowerON_Reset_PC
        .EXPORT     _Manual_Reset_PC

        .SECTION    ResetPRG,CODE
_PowerON_Reset_PC:
_Manual_Reset_PC:
    MOV.L       #_INT_Vectors-INT_OFFSET,R0
    LDC         R0,VBR                      ; VBR setting

    MOV.L           #_ _INITSCT,R1
    JSR             @R1                     ; call _ _INITSCT
    NOP

    MOV.L           #_ _INIT_IOLIB,R1
    JSR             @R1                     ; call _ _INIT_IOLIB
    NOP

    MOV.L           #_ _INIT_OTHERLIB,R1
    JSR             @R1                     ; call _ _INIT_OTHERLIB
    NOP

    MOV.L           #_ _CALL_INIT,R1
    JSR             @R1                     ; call _ _CALL_INIT
    NOP

    MOV.L           #SR_Init,R0
    MOV.L           #_main,R8
    LDC R0,SR                               ; SR setting
```

```
JSR             @R8                     ; call main function
NOP

MOV.L           #_ _CALL_END,R1
JSR             @R1                     ; call _ _CALL_END
NOP

MOV.L           #_ _CLOSEALL,R1
JSR             @R1                     ; call _ _CLOSEALL
NOP

SLEEP
NOP
.POOL
.END
```

(3) Section Initialization (_ _INITSCT)

To set the C/C++ program execution environment, clear the non-initialized data area with zeros and copy the initialized data area in ROM to RAM. To execute the _ _INITSCT function, the following addresses must be known.

— Start address (1) of initialized data area in ROM
— Start address (2) and end address (3) of initialized data area in RAM
— Start address (4) and end address (5) of non-initialized data area



HITACHI

To obtain the above addresses, create the following assembly program and link it together.

```
        .SECTION D,DATA,ALIGN=4
        .SECTION R,DATA,ALIGN=4
        .SECTION B,DATA,ALIGN=4
        .SECTION C,DATA,ALIGN=4


_ _D_ROM  .DATA.L  (STARTOF D)
; start address of section D                              (1)
_ _D_BGN  .DATA.L  (STARTOF R)
; start address of section R                              (2)
_ _D_END  .DATA.L  (STARTOF R) + (SIZEOF R)
; end address of section R                                (3)
_ _B_BGN  .DATA.L  (STARTOF B)
; start address of section B                              (4)
_ _B_END  .DATA.L  (STARTOF B) + (SIZEOF B)
; end address of section B                                (5)
          .EXPORT _ _D_ROM
          .EXPORT _ _D_BGN
          .EXPORT _ _D_END
          .EXPORT _ _B_BGN
          .EXPORT _ _B_END
          .END
```

Notes: 1. Section names B and D must be the section names for non-initialized data area and initialized data area, which are specified with the **section** option or #pragma section, respectively. B and D indicate the default section names.

2. Section name R must be the section name in RAM area specified with the **rom** option at linkage. R indicates the default section name.

If the above preparation is completed, section initialization routine can be written in C/C++ as shown below.

Example:

(a) Section initialization routine

```
extern int  *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;
#ifdef _ _cplusplus
extern "C"
#endif
void _INITSCT( )
{
        int *p, *q ;

        /* Non-initialized data area is initialized to zeros */

        for (p = _B_BGN ; p < _B_END ; p++)
                *p = 0 ;

        /* Initialized data is copied from ROM to RAM */

        for (p = _D_BGN , q = _D_ROM ; p < _D_END ; p++, q++)
               *p = *q ;
}
```

Note: The declaration of p and q must be a char* type when the section size is not a multiple of four bytes.

(b) Global object initial processing/post-processing routine
_CALL_INIT and _CALL_END routines are provided in the library. When using these routines, _h_c_lib.h must be included.

(4) 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 to be 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 prepared low-level interface routines, the initial setting (_INIT_LOWLEVEL) in accordance with the specifications of the low-level interface routines is necessary.

— When using the rand function or 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

struct _iobuf _iob[IOSTREAM];
unsigned char sml_buf[IOSTREAM];
extern char *_s1ptr;


#ifdef _ _cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
    _INIT_LOWLEVEL();  // Set initial setting for low-level interface routines
    _INIT_IOLIB();     // Set initial setting for I/O library
    _INIT_OTHERLIB();  // Set initial setting for rand function, strtok function
}

void _INIT_LOWLEVEL (void)
{
// Set necessary initial setting for low-level library
}

void _INIT_IOLIB(void)
{
    FILE *fp;
    for( fp = _iob; fp < _iob + _nfiles; fp++ )    // Set initial setting 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("stdin*1", "r", stdin)== NULL)    // Open standard input file
        stdin->_ioflag1 = 0xff;                  // Forbid file access if open fails
    stdin->_ioflag1 |= _IOUNBUF;                 // Disable data buffering*2
    if(freopen("stdout*1", "w", stdout)== NULL) // Open standard output file
        stdout->_ioflag1 = 0xff;                 // Forbid file access if open fails
    stdout->_ioflag1 |= _IOUNBUF;                // Disable data buffering*2
    if(freopen("stderr*1", "w", stderr)== NULL) // Open standard error file
        stderr->_ioflag1 = 0xff;                 // Forbid file access if open fails
    stderr->_ioflag1 |= _IOUNBUF;                // Disable data buffering*2
}

void _INIT_OTHERLIB(void)
{
    srand(1);                   // Set initial setting if using rand function
    _s1ptr=NULL;                // Set initial setting if using strtok function
}
#ifdef _ _cplusplus
}
#endif
```

Notes: 1. Specify the filename for the standard I/O file. This name is used in the low-level interface routine "open".

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

```
                /* File-type data declaration in C language */

struct _iobuf{
             unsigned char *_bufptr;     /* Pointer to buffer */
             long           _bufcnt;     /* Buffer counter */
             unsinged char *_bufbase;    /* Base pointer to buffer */
             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

(5) Closing files (_CLOSEALL)

Normally, output to files is held in a buffer area in memory, and when the buffer becomes full data is 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 embedding in equipment, 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 any files that are open at the time of termination of the main function.

An example of a program to close any open files is shown below.

```
#include <stdio.h>

#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void)
{
int i;

    for( i=0; i < _nfiles; i++ )

        // Check to see whether the file is open or not
        if( _iob[i]._ioflag1 & (_IOREAD | _IOWRITE | _IORW ) )
            fclose( &_iob[i] );        // Close the file
}
```

(6) Low-level interface routines

When using standard I/O or memory management library functions in a C/C++ program, low-level interface routines must be prepared. Table 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 file
write	Writes to file
lseek	Sets the read/write position in a file
sbrk	Allocates area in memory

Initialization necessary for low-level interface routines must be performed on program startup. This initialization should be performed using the _INIT_LOWLEVEL function described in section 9.2.2 (4), 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.

Note: The function names open, close, read, write, lseek, and sbrk are reserved for low-level interface routines. 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 in a one-to-one correspondence with actual files for management. These integers are called file numbers.

In the open routine, a file number is provided for a specified filename. The open routine must be 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, write routines) and read/write positioning (lseek routine) is performed.

When output buffering is being used, the close routine should be executed to write the contents of the buffer to the actual file, so that the data area set by the open routine can be reused.

(b) Specifications of low-level interface routines

In this section, specifications for low-level interface routines are described. For each routine, the interface for calling the routine, its operation, and information for using the routine are described.

The interface for the routines is indicated using the following format. Low-level interface routines should always be given a prototype declaration. Add "extern C" to declare in the C++ program.

(Routine name)

Description	(A summary of the routine operations is given)	
Return value	Normal: (The 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 a parameter)

int open(char *name, int mode, int flg)

Description

Prepares for operations on the file corresponding to the filename of the first parameter. In the open routine, the file type (console, printer, disk file, etc.) must be determined in order to enable writing or reading at a later time. The file type must be referenced using the file number returned by the open routine each time reading or writing is to be performed.

The second parameter, mode, specifies processing to be performed when the file is opened. The meanings of each of the bits of this parameter are as follows.

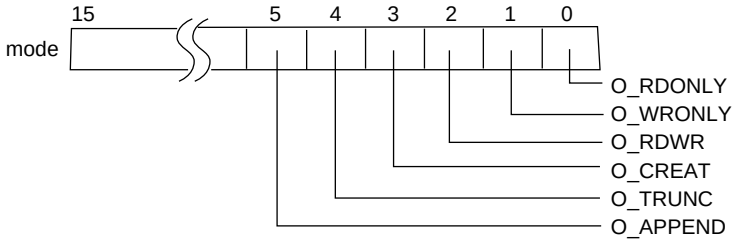


Table 9.4 Explanation of Bits in Parameter "mode" of the File Open Routine

Bit	Description
O_RDONLY (bit 0)	When this bit is 1, the file is opened in read-only mode
O_WRONLY (bit 1)	When this bit is 1, the file is opened in write-only mode
O_RDWR (bit 2)	When this bit is 1, the file is opened for both reading and writing
O_CREAT (bit 3)	When this bit is 1, if a file with the filename given does not exist, it is created
O_TRUNC (bit 4)	When this bit is 1, if a file with the filename given exists the file contents are deleted, and the file size is set to 0
O_APPEND (bit 5)	Sets the position within the file for the next read/write operation When 0: Set to read/write from the beginning of file When 1: Set to read/write from file end

When there is a contradiction between the file processing specified by mode and the properties of the actual file, error processing should be performed. When the file is opened normally, the file number (a positive integer) should be returned which should be used in subsequent read, write, lseek, and close routines. The correspondence between file numbers and the actual files must be managed by low-level interface routines. When a file open operation fails, -1 should be returned.

Return value	Normal:	The file number for the successfully opened file
	Error:	-1
Parameters	name:	The filename for 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)

Description

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 written to the actual file.

When the file is closed successfully, 0 is returned; if the close operation fails, -1 is returned.

Return value	Normal:	0
	Error:	-1
Parameter	fileno:	File number for the file to close

int read(int fileno, char *buf, unsigned int count)

Description

Data is read from the file specified by the first parameter (fileno) to the area in memory specified by the second parameter (buf). The number of bytes of data to be read is specified by the third parameter (count).

When the end of the file is reached, only a number of bytes fewer than or equal to count bytes can be read.

The position for file reading/writing advances by the number of bytes read.

When reading is performed successfully, the actual number of bytes read is returned; if the read operation fails, -1 is returned.

Return value	Normal:	Actual number of bytes read
	Error:	-1

Parameters	fileno	File number of the file to be read
	buf	Memory area to store read data
	count	Number of bytes to read

int write(int fileno, char *buf, unsigned int count)

Description

Writes data to the file indicated by the first parameter (fileno) from the memory area indicated by the second parameter (buf). The number of bytes to be written is indicated by the third parameter (count).

If the device (disk etc.) of the file to be written is full, only a number of bytes fewer than or equal to count bytes can be written. It is recommended that, if the number of bytes actually written is zero a certain number of times in succession, the disk should be judged to be full and an error (-1) should be returned.

The position for file reading/writing advances by the number of bytes written. If writing is successful, the actual number of bytes written should be returned; if the write operation fails, -1 should be returned.

Return value:	Normal:	Actual number of bytes written
	Error:	-1

Parameters:	fileno	File number to which data is to be written
	buf	Memory area containing data for writing
	count	Number of bytes to write

int lseek(int fileno, long offset, int base)

Description:

Sets the position within the file, in byte units, for reading from and writing to the file. The position within a new file should be calculated and set using the following methods, depending on the third parameter (base).

(1) When base is 0: Set the position at offset bytes from the file beginning

(2) When base is 1: Set the position at the current position plus offset bytes

(3) When base is 2: Set the position at the file size plus offset bytes

When the file is a console, printer, or other interactive device, and when the new offset is negative, or when in cases (1) and (2) the file size is exceeded, an error occurs. When the file position is set correctly, the new position for

reading/writing should be returned as an offset from the file beginning; when the operation is not successful, -1 should be returned.

Return value:	Normal:	The new position for file reading/writing, as an offset in bytes from the file beginning
	Error:	-1
Parameters:	fileno	File number
	offset	The position for reading/writing, as an offset (in bytes)
	base	The starting-point of the offset

char *sbrk(int size)

Description: The size of the memory area to be allocated is passed as a parameter.

When calling the sbrk routine several times, memory areas should be allocated in succession starting from lower addresses. If memory area for allocation is insufficient, an error should occur. When allocation is successful, the address of the beginning of the allocated memory area should be returned; if unsuccessful, (char *) -1 should be returned.

Return value:	Normal:	Starting address of allocated memory
	Error:	(char *) -1
Parameter:	size	Size of area to be allocated

(c) Example of Coding the Low-Level Interface Routine

```
/************************************************************************/
/*                                lowsrc.c:                             */
/*----------------------------------------------------------------------*/
/*      SuperH RISC engine Series Simulator/Debugger Interface Routine  */
/*      Only standard I/O (stdin,stdout,stderr) are supported           */
/************************************************************************/
#include <string.h>

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

#define FLMIN 0           /*  Minimum file number */
#define FLMAX 3           /*  Maximum number of files */

/*  File flags  */
#define O_RDONLY 0x0001    /*  Read only */
#define O_WRONLY 0x0002    /*  Write only */
#define O_RDWR 0x0004      /*  Read/Write */

/*  Special character code  */
#define CR 0x0d            /*  Carriage return */
#define LF 0x0a            /*  Line feed */

/*  Area size managed by sbrk  */
#define HEAPSIZE 1024

/************************************************************************/
/*              Reference function declaration                          */
/*  Assembly program reference which inputs/outputs characters to       */
/*  console using simulator/debugger                                    */
/************************************************************************/
extern void charput(char); /*  One character input processing */
extern char charget(void); /* One character output processing */

/************************************************************************/
/*              Static variable definition                              */
/*  Definition of static variables used in low-level interface routine  */
/************************************************************************/
char flmod[FLMAX]; /*  Mode setting location of open file  */

static  union {
    long dummy;/*  Dummy for four-byte alignment   */
    char heap[HEAPSIZE];   /*  Declaration of area managed by sbrk  */
} heap_area;

static char *brk=(char*)&heap_area; /*  End address allocated by sbrk  */
```

```
/*********************************************************************/
/*                              open: Open file                      */
/*                        Return value:  File Number  (Success)      */
/*                                       -1   (Failure)              */
/*********************************************************************/
int open(char *name,   /*  File name  */
         int mode)   /*  File mode  */
{
   /* Check mode according to the file name, and return the file number */

    if (strcmp(name,"stdin")==0) {  /*  Standard input file  */
        if ((mode&O_RDONLY)==0) {
            return (-1);
        }
        flmod[STDIN]=mode;
        return (STDIN);
    }

    else if (strcmp(name,"stdout")==0) {   /*  Standard output file  */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDOUT]=mode;
        return (STDOUT);
    }

    else if (strcmp(name,"stderr")==0){    /* Standard error output file */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDERR]=mode;
        return (STDERR);
    }

    else {
        return (-1);   /* Error  */
    }
}

/*********************************************************************/
/*                            close:  Close file                     */
/*                          Return value       0      (Success)     */
/*                                             -1     (Failure)     */
/*********************************************************************/
int close(int fileno)  /*  File number  */
{
    if (fileno<FLMIN || FLMAX<fileno) {    /*  Check file number range  */
        return -1;
    }

    flmod[fileno]=0;   /*  Reset file mode  */

    return 0;
}
```

```
/*************************************************************************/
/*                                  read:  Read data                     */
/*                        Return value:  Read character count (Success) */
/*                                        -1                 (Failure) */
/*************************************************************************/
int read(int fileno,   /*  File number  */
         char *buf,    /*  Transfer destination buffer address  */
         unsigned int  count)      /*  Read character count  */
{
    unsigned int i;

    /*  Check mode according to file name, input one character each,  */
    /* and store the characters to buffer  */

    if (flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR) {
        for (i=count;i>0;i--) {
            *buf=charget();
            if (*buf==CR) { /*  Replace line feed character  */
                *buf=LF;
            }
        buf++;
        }
        return count;
    }

    else {
        return -1;
    }
}

/*************************************************************************/
/*                                  write:  Write data                   */
/*                          Return value:  Written data count (Success)  */
/*                                        -1                 (Failure)  */
/*************************************************************************/
int write(int fileno,  /*  File number  */
          char *buf,   /*  Transfer source buffer address  */
          unsigned int  count)     /*  Written character count  */
{
    unsigned int i;
    char c;

/*  Check mode according to file name and output one character at a time */

    if (flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR) {
        for (i=count; i>0; i--) {
            c=*buf++;
            charput(c);
        }
        return count;
    }

    else {
        return -1;
    }
}
```

```
/*********************************************************************/
/*                       lseek: Set file read/write position         */
/*Return value: Offset from the beginning of file to be read/written (Success)      */
/*              -1 (Failure)                                         */
/*                    (Console I/O does not support lseek)           */
/*********************************************************************/
long lseek(int fileno,      /*  File number  */
           long offset,     /*  Read/write start position*/
           int  base)       /*  Start of offset  */
{
    return -1;
}

/*********************************************************************/
/*                                  sbrk: Data write                 */
/*                     Return value:  Start address of allocated area  */
/*                                    -1  (Failure)                  */
/*********************************************************************/
char *sbrk(unsigned long size)     /*  Size of area to be allocated  */
{
    char *p;

    /*  Check empty area  */

    if (brk+size>heap_area.heap+HEAPSIZE) {
        return (char *)-1;
    }

    p=brk; /* Allocate area */
    brk+=size; /* Update end address  */
    return p;
}


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                  lowlvl.src                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  SuperH RISC engine Series Simulator/Debugger Interface Routine     ;
;                           Input/Output one character                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    .EXPORT _charput
    .EXPORT _charget
SIM_IO:
    .EQU    H'0000  ; Specify TRAP_ADDRESS

    .SECTION    P, CODE, ALIGN=4
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                              _charput: One character output                  ;
;                    C program interface: charput(char)                        ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

_charput:
    MOV.L   O_PAR,R0       ; Set buffer address
    MOV.B   R4,@R0         ; Set parameter to buffer
    MOV.L   #O_PAR,R1      ; Set parameter block address
    MOV.L   #H'01220000,R0 ; Set function code (PUTC)
    MOV.W   #SIM_IO,R2     ; Set system call address
    JSR     @R2
    NOP
    RTS
    NOP

    .ALIGN  4
O_PAR:                     ;Parameter block area
    .DATA.L OUT_BUF

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                              _charget:One character input                    ;
;                C program interface: char charget(void)                       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .ALIGN  4
_charget:
    MOV.L   #I_PAR,R1      ; Set parameter block address
    MOV.L   #H'01210000,R0 ; Set function code (GETC)
    MOV.W   #SIM_IO,R2     ; Set system call address
    JSR     @R2
    NOP
    MOV.L   I_PAR,R0       ; Set buffer address
    MOV.B   @R0,R0         ; Set the input data as the return value
    RTS
    NOP

          .ALIGN      4
I_PAR:                     ; Parameter block area
          .DATA.L     IN_BUF

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                        Definition of I/O buffer                             ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .SECTION    B,DATA,ALIGN=4

OUT_BUF:
    .RES.L  1   ; Output buffer
IN_BUF:
    .RES.L  1   ; Input buffer

    .END
```

(7) Termination Processing Routine

(a) Example of preparation of a routine for termination processing registration and execution (atexit)

The method for preparation of the library function atexit to register termination processing is described.

The atexit function registers, in a table for termination processing, a function address passed as a parameter. If the number of functions registered exceeds the limit (in this case, the number that can be registered is assumed to be 32), or if an attempt is made to register the same function twice, NULL is returned. Otherwise, a value other than NULL (in this case, the address of the registered function) is returned.

Example:

```
#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 it is already registered
        if(_atexit_buf[i]==f)
             return NULL ;
    if(_atexit_count==32) // Check the limit value of number of registration
        return NULL ;
    else {
        atexit_buf[_atexit_count++]=f;     // Register function address
        return f;
    }
}
```

(b) Example of preparation of a routine for program termination (exit)

The method for preparation of an exit library function for program termination is described. Program termination processing will differ among user systems; refer to the program example below when preparing a termination procedure according to the specifications of the user system.

The exit function performs termination processing for a program according to the termination code for the program passed as an 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 settings function, the function callmain 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
      (*_atexit_buf[i])();           // by the atexit function
  _CLOSEALL();       // Close all open functions
  longjmp(_init_env, 1) ;         // Return to the environment saved by setjmp
}
#ifdef _ _cplusplus
extern "C"
#endif
void callmain(void)
{
      //Save the current environment using setjmp, 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, processing for abnormal termination must be executed in accordance with the specifications of the user system.

In a C++ program, the abort function will also be called in the following cases:

- When exception processing was unable to operate correctly.
- When a pure virtual function is called.
- When dynamic_cast has failed.
- When typeid has failed.
- When information could not be acquired when class array was deleted.
- When the definition of the destructor call for objects of a given class causes a contradiction.

Below is shown an example of a program which outputs a message to the standard output device, then closes all files and begins an 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 is abort !!\n");//Output message
        _CLOSEALL();                    //Close all files
        while(1)    ;                   //Begin endless loop
}
```

9.3 Linking C/C++ Programs and Assembly Programs

Here the following matters to be born in mind when linking C/C++ programs and assembly programs are discussed.

- Method for mutual referencing of external names
- Interface for function calls

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

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

In assembly programs, 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 (definition)

```
     .EXPORT  _a, _b
     .SECTION D,DATA,ALIGN=4
_a:  .DATA.L  1
_b:  .DATA.L  1
     .END
```

C/C++ program (reference)

```
extern int a,b;


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

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

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

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

C/C++ program (definition)

```
int a;
```

Assembly program (reference)

```
        .IMPORT   _a
        .SECTION  P,CODE,ALIGN=2
        MOV.L     A_a,R1
        MOV.L     @R1,R0
        ADD       #1,R0
        RTS
        MOV.L     R0,@R1
        .ALIGN    4
A_a:    .DATA.L   _a
        .END
```

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

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

C++ program (callee)

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

Assembly program (caller)

```
        .IMPORT  _sub
        .SECTION P,CODE,ALIGN=4
              .
              .




        STS.L    PR,@-R15
        MOV.L    R1,@(1,R15)
        MOV      R3,R12
        MOV.L    A_sub,R0
        JSR      @R0
        NOP
        LDS.L    @R15+,PR
              .
              .
A_sub:  .DATA.L  _sub
        .END
```

9.3.2 Function Calling Interface

When either a C/C++ program or an assembly program calls the other, the assembly programs must be written using rules involving the following:

1. Stack pointer
2. Allocating and deallocating stack frames
3. Registers
4. Setting and referencing parameters and return values

(1) Stack Pointer

Valid data must not be stored in a stack area with an address lower than the stack pointer (in the direction of address H'0), since the data may be destroyed by an interrupt process.

(2) Allocating and Deallocating Stack Frames

In a function call (right after the JSR or the BSR instruction has been executed), the stack pointer indicates the lowest address of the stack used by the calling function. Allocating and setting data at addresses greater than this address must be done by the caller.

After the caller deallocates the area it has set with data, control returns to the caller usually with the RTS instruction. The caller then deallocates the area having a higher address (the return value address and the parameter area).

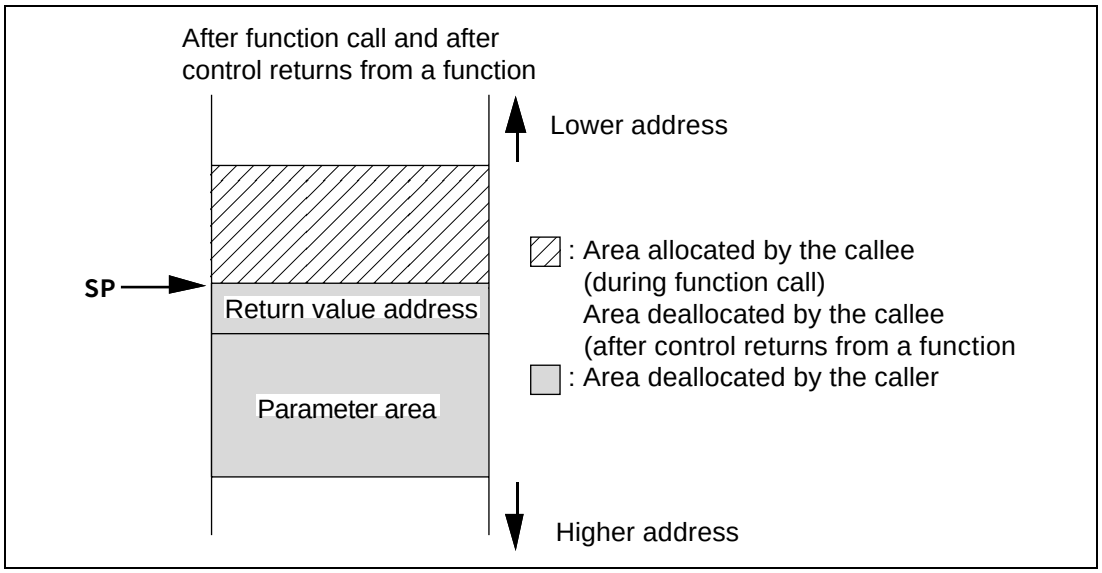


Figure 9.8 Allocation and Deallocation of a Stack Frame

(3) Registers

Some registers may change during a function call, while some may not. Table 9.5 shows the rules to save and restore in registers.

Table 9.5 Rules to Save and Restore in Registers

Item	Registers Used in a Function	Notes on Programming
Caller-save registers	R0 to R7, FR0 to FR11*[1], DR0 to DR10*[2], FPUL*[1]*[2], and FPSCR*[1]*[2]	If registers used in a function contain valid data when a program calls the function, the caller must save the data onto the stack or into the register before calling the function. The data in registers used in called function can be used without being saved.
Callee-save registers	R8 to R15, MACH, MACL, PR, FR12 to FR15*[1], and DR12 to DR14*[2]	The data in registers used in functions is saved onto the stack at function entry, and restored from the stack at function exit. Note that data in the MACH and MACL registers are not guaranteed if the **macsave=0** option is specified.

Notes: 1 Single-precision floating point registers for SH-2E, SH-3E, and SH-4.
2 Double-precision floating point registers for SH-4.

The following examples show the rules on registers.

- A subroutine in an assembly program is called by a C/C++ program

Assembly program (callee)

```
        .EXPORT  _sub
        .SECTION P,CODE,ALIGN=4
_sub:   MOV.L    R14,@-R15
        MOV.L    R13,@-R15
        ADD      #-8,R15

                 .
                 .


        ADD      #8,R15
        MOV.L    @R15+,R13
        RTS
        MOV.L    @R15+,R14
        .END
```

Saves the registers used in the function.

Processing of the function
(R0 to R7 are saved by the caller
they can be used without
saving within the callee.)

Restores the saved registers.

C/C++ program (caller)

```
        #ifdef __cplusplus
        extern "C"
        #endif
        void sub();

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

- A function in a C/C++ program is called by an assembly program

  C/C++ program (callee)

```
void sub()
{
      .
      .
}
```

Assembly program (caller)

```
        .IMPORT  _sub
        .SECTION P,CODE,ALIGN=4
              .
              .




        STS.L    PR,@-R15

        MOV.L    R1,@(1,R15)
        MOV      R3,R12

        MOV.L    A_sub,R0
        JSR      @R0
        NOP
        LDS.L    @R15+,PR
              .
              .
A_sub:  .DATA.L  _sub
        .END
```

} The called function name prefixed with (_) is declared by the .IMPORT directive (C).
The external name generated from the function declaration or definition by the compiler is declared by the .IMPORT directive (C++).

} Stores the PR register (return address storage register) when calling the function.

} If registers R0 to R7 contain valid data, the data is pushed onto the stack or stored in unused registers.

} Calls function sub.

} Restores the PR register.

Address data of function sub.

Note: The compiler uses a rule to convert the external name created by the function name or static data member. When you need to know the external name created by the compiler, refer to the external name created by the compiler using the **code=asm** or **listfile** option. Defining a C++ function with extern "C" specified applies the same generation rules as C functions to external names, although this makes overloading of the function impossible.

(4) Setting and Referencing Parameters and Return Values

This section explains how to set and reference parameters and return values.

This section first explains the general rules concerning parameters and return values, and then how the parameter area is allocated, and how to set return values.

(a) General rules concerning parameters and return values

— Passing parameters

A function is called after parameters have been copied to a parameter area in registers or on the stack. Since the caller does not reference the parameter area after control returns to it, the caller is not affected even if the callee modifies the parameters.

— Rules on type conversion

Type conversion may be performed automatically when parameters are passed or a return value is returned. The following explains the rules on type conversion.

- Type conversion of parameters whose types are declared:

  Parameters whose types are declared by prototype declaration are converted to the declared types.

- Type conversion of parameters whose types are not declared:

  Parameters whose types are not declared by prototype declaration are converted according to the following rules.

  (signed) char, unsigned char, (signed) short, and unsigned short type parameters are converted to (signed) int type parameters.

  float type parameters are converted to double type parameters.

  Types other than the above are not converted.

- Return value type conversion:

  A return value is converted to the data type returned by the function.

Examples:

```
(1)  long f( );
     long f( )
     {   float x;
         return x;
     }

(2)  void p ( int,... );
     void f ( )
     {   char c;
         P ( 1.0, c );
     }
```

The return value is converted to long by a prototype declaration.

c is converted to int because a type is not declared for the parameter.

1.0 is converted to int because the type of the parameter is int.

(b) Parameter area allocation

Parameters are allocated to registers, or when this is impossible, to a stack parameter area. Figure 9.9 shows the parameter area allocation. Table 9.6 lists rules on general parameter area allocation. The this pointer to a nonstatic function member in C++ program is assigned to R4.

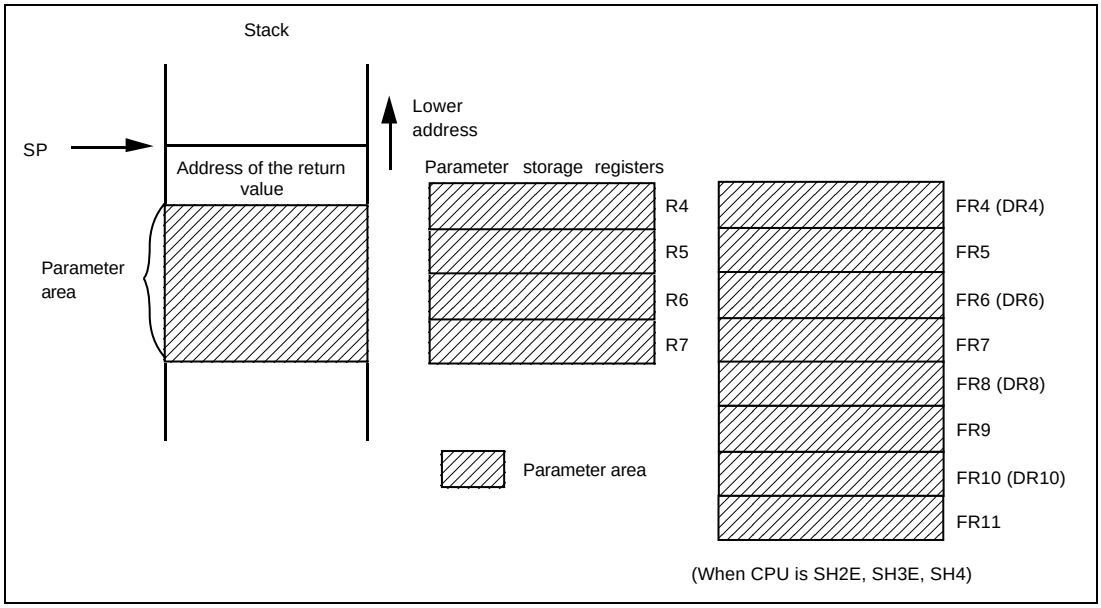


Figure 9.9 Parameter Area Allocation

Table 9.6 General Rules on Parameter Area Allocation

Parameters Allocated to Registers		
Parameter Storage Registers	**Target Type**	**Parameters Allocated to a Stack**
R4 to R7	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, float (when CPU is SH-1, SH-2, SH-3), pointer, pointer to a data member, and reference	(1) Parameters whose types are other than target types for register passing (2) Parameters of a function which has been declared by a prototype declaration to have variable-number parameters*3 (3) When other parameters are already allocated to R4 to R7. (4) When other parameters are already allocated to FR4 (DR4) to FR11 (DR10).
FR4 to FR11 *1	For SH-2E and SH-3E • Parameter is float type. • Parameter is double type and **double=float** option is specified. For SH-4 • Parameter type is float type and **fpu=double** option is not specified. • Parameter type is double type and **fpu=single** option is specified.	
DR4 to DR10 *2	For SH-4 • Parameter type is double type and **fpu=single** option is not specified. • Parameter type is float type and **fpu=double** option is specified.	

Notes: 1. Single-precision floating point register for SH-2E, SH-3E, and SH-4.
2. Double-precision floating point register for SH-4.
3. If a function has been declared to have variable-number parameters by a prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding parameter are allocated to a stack.

Example:

```
int f2(int,int,int,int,...);
     :
     f2(a,b,c,x,y,z); ← x,y, and z are allocated to a stack.
```

(2) Parameter allocation

— Allocation to parameter storage registers

Following the order of their declaration in the source program, parameters are allocated to the parameter storage registers starting with the smallest numbered register. Figure 9.10 shows an example of parameter allocation to registers.

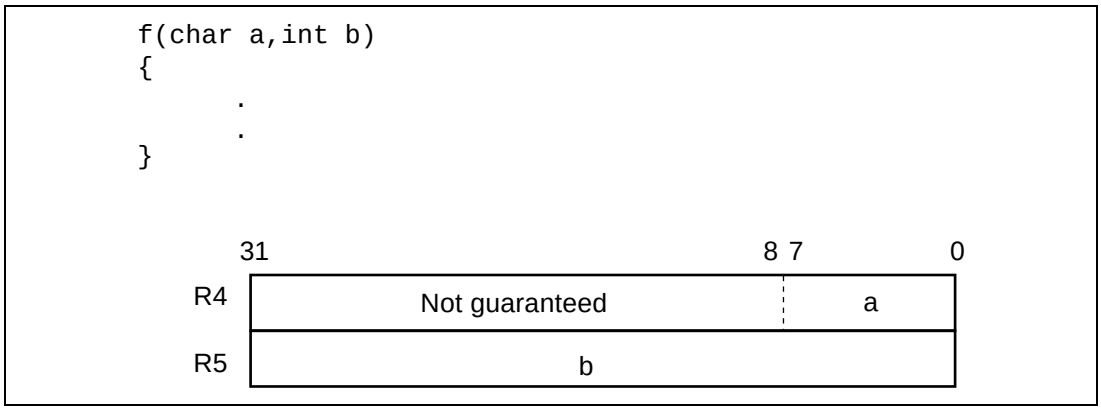


Figure 9.10 Example of Allocation to Parameter Registers

— Allocation to a stack parameter area

Parameters are allocated to the stack parameter area starting from lower addresses, in the order that they are specified in the source program.

Note: Regardless of the alignment determined by the structure type, union type, or class type, parameters are allocated using 4-byte alignment. Also, the area size for each parameter must be a multiple of four bytes. This is because the SuperH RISC engine microcomputer stack pointer is incremented or decremented in 4-byte units.

Refer to section 9.3.3, Examples of Parameter Assignment, for examples of parameter allocation.

- Return value writing area

The return value is written to either a register or memory depending on its type. Refer to table 9.7 for the relationship between the return value type and area.

When a function return value is to be written to memory, the return value is written to the area indicated by the return value address. The caller must allocate the return value setting area in addition to the parameter area, and must set the address of the return value area in the return value address area before calling the function (see figure 9.11). The return value is not written if its type is void.

Table 9.7 Return Value Type and Setting Area

Return Value Type	Return Value Area
(signed) char, unsigned char, (signed) short, unsigned short, (signed) int, unsigned int, long, unsigned long, float, pointer, bool, reference, and pointer to a data member	R0: 32 bits (The contents of the upper three bytes of (signed) char, or unsigned char and the contents of the upper two bytes of (signed) short or unsigned short are not guaranteed.) However, when the **rtnext** option is specified, sign extension is performed for (signed) char or (signed) short type, and zero extension is performed for unsigned char or unsigned short type. FR0: 32 bits (1) For SH-2E and SH-3E • Return value is float type. • Return value is double type and **double=float** option is specified. (2) For SH-4 • Return value is float type and **fpu=double** option is not specified. • Return value is floating-point type and **fpu=single** option is specified.
double, long double, structure, union, class, and pointer to a function member	Return value setting area (memory) DR0: 64 bits For SH-4 • Return value is double type and **fpu=single** option is not specified. • Return value is floating-point type and **fpu=double** option is specified.

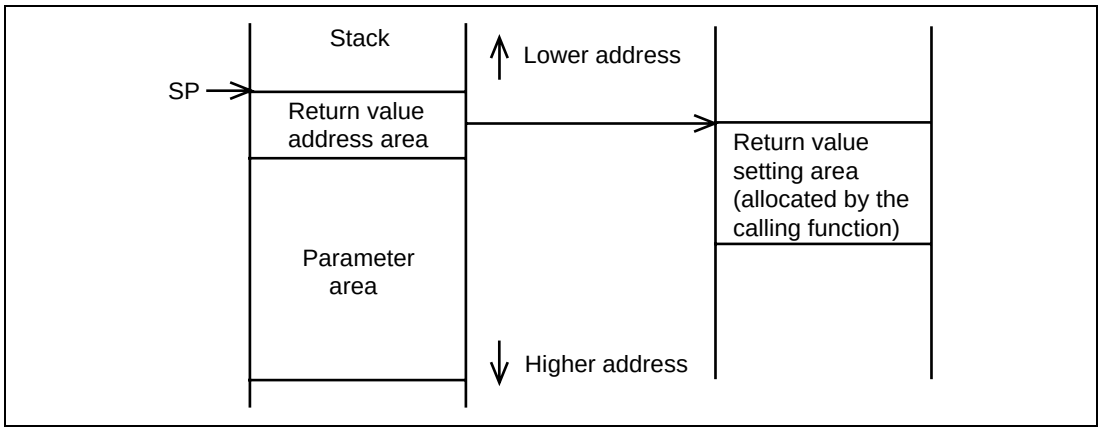
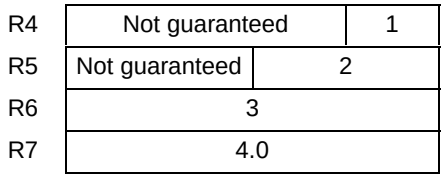


Figure 9.11 Return Value Setting Area Used When Return Value Is Written to Memory

9.3.3 Examples of Parameter Assignment

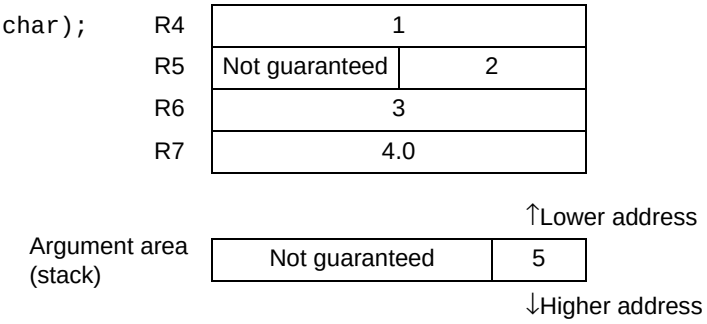
Example 1 Arguments passed by are assigned, in the order in which they are declared, to registers R4 to R7.

```
int f(char,short,int,float);
    :
f(1,2,3,4.0);
```



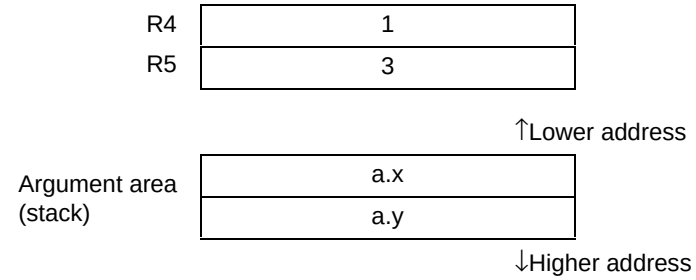
Example 2 Arguments that cannot be assigned to registers are assigned to the stack. When the arguments are (unsigned) char or (unsigned) short types and are assigned to the argument area in the stack, they are first extended to 4 bytes.

```
int f(int,short,long,float,char);
    :
f(1,2,3,4.0,5);
```



Example 3 Arguments of types that cannot be assigned to registers are assigned to the stack.

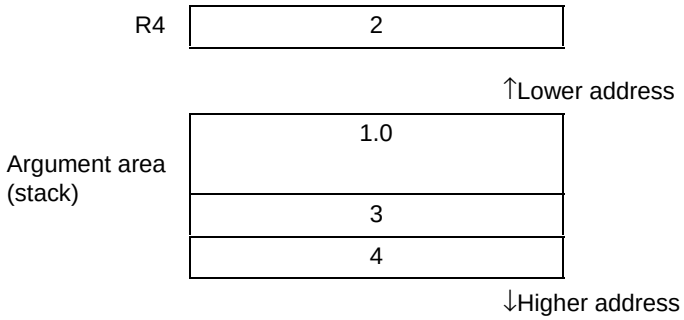
```
struct s{int x,y;}a;
int f(int,struct s,int);
:
f(1,a,3);
```



HITACHI

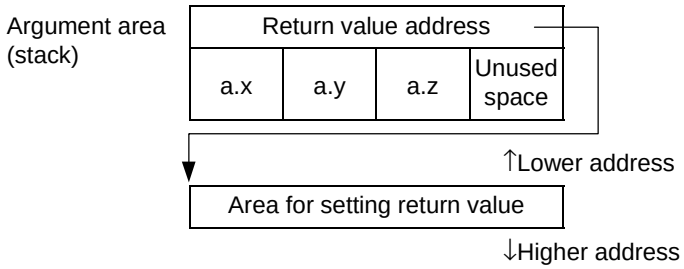
Example 4 When declared in a prototype declaration as a function with a variable number of arguments, the arguments without corresponding types and the immediately preceding argument are assigned to the stack in the order in which they are declared.

```
int f(double, int, int…);
:
f(1.0,2,3,4);
```



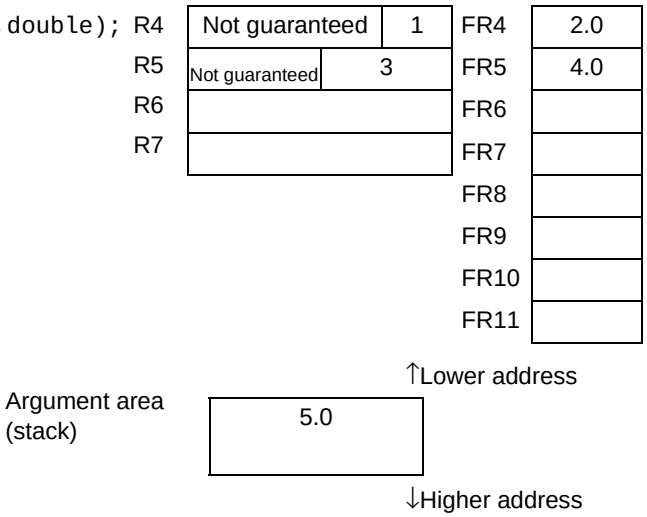
Example 5 When the type returned by a function is more than 4 bytes, or a class, the return value address is set immediately before the argument area. If the size of the class is not a multiple of 4 bytes, unused space is padded.

```
struct s{char x,y,z;}a;
double f(struct s);
:
f(a);
```



Example 6 When the CPU is SH-2E or SH-3E, float type arguments are assigned to the FPU registers.

```
int f(char,float,short,float,double);
:
f(1,2.0,3,4.0,5.0);
```



Example 7 When the CPU is SH-4 and **fpu** option is not specified, float and double type arguments are assigned to the FPU registers.

```
int f(char,float,double,float,short);
  :
f(1,2.0,4.0,5.0,3);
```

R4	Not guaranteed	1
R5	Not guaranteed	3
R6		
R7		

FR4(DR4)	2.0
FR5	5.0
FR6(DR6)	4.0
FR7	
FR8(DR8)	
FR9	
FR10(DR10)	
FR11	

9.3.4 Using the Registers and Stack Area

This section describes how the compiler uses registers and stack areas. Registers and stack areas in functions are controlled by the compiler and the user is not required to have any particular understanding of how these areas are used. Figure 9.11 shows how the register and stack areas are used.

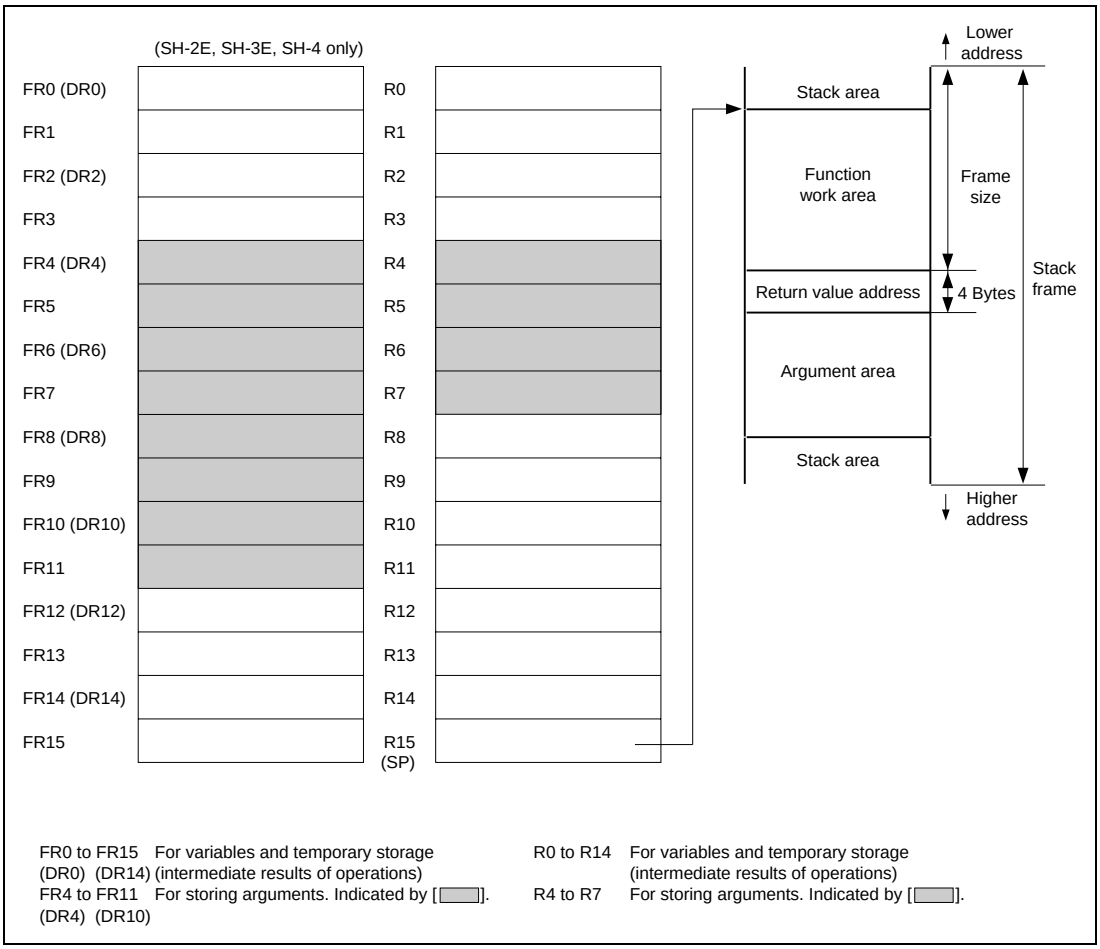


Figure 9.12 Using Register and Stack Areas

HITACHI

9.4 Important Information on Programming

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 discussed.

9.4.1 Important Information on Program Coding

(1) Float Type Parameter Function

Functions must declare prototypes or change float type to double type when receiving and passing float type parameters. Data value cannot be guaranteed when a float type parameter without a prototype declaration receives data.

Example:

```
void f (float);
void g ()
{
   float a;
   ...
   f (a);
}
void f (float x)
{
   .
   .
   .
}
```

Function f has a float type parameter. Therefore, a prototype must be declared.

(2) Expressions whose Evaluation Order is not Specified by the C/C++ Language

The effect of the execution is not guaranteed in a program whose execution results differ depending on the evaluation order.

Example:

`a[i]=a[++i];`	The value of i on the left side differs depending on whether the right side of the assignment expression is evaluated first.
`sub(++i, i);`	The value of i for the second parameter differs depending on whether the first function parameter is evaluated first.

(3) Overflow Operation and Zero Division

At run time if overflow operation or zero division is performed, error messages will not be output. However, if an overflow operation or zero division is included in the operations of constants, error messages will be output at compilation.

Example:

```
    void main()
    {
       int ia;
       int ib;
       float fa;
       float fb;

       ib=32767;
       fb=3.4e+38f;

/* Compilation error messages are output when an overflow       */
/* operation and zero division are included in operations       */
/* of constants.                                               */

       ia=99999999999;  /* (W) Detect integer constant overflow. */
       fa=3.5e+40f;     /* (W) Detect floating pointing constant */
                        /* overflow.                             */
       ia=1/0;          /* (E) Detect division by zero.          */
       fa=1.0/0.0;      /* (W) Detect division by floating point */
                        /* zero.                                 */

/* No error message on overflow at execution is output.         */

       ib=ib+32767;     /* Ignore integer constant overflow.     */
       fb=fb+3.4e+38f;  /* Ignore floating point constant        */
                        /* overflow.                             */

    }
```

(4) Assignment to const Variables

Even if a variable is declared with const type, if assignment is done to a non-constant variable converted from const type or if a program compiled separately uses a parameter of a different type, the compiler cannot detect the error.

Example:

```
const char *p;          /* Because the first parameter p in library */
   .                    /* function strcat is a pointer for char,   */
   .                    /* the area indicated by the parameter p    */
strcat(p, "abc");       /* may change.                              */
```

file 1

```
const int i;
```

file 2

```
extern int i;          /* In file 2, parameter i is not declared as */
:                      /* const, therefore assignment to it in      */
i=10;                  /* file 2 is not an error                    */
```

(5) Precision of Mathematical Function Libraries

For function acos (x) and asin (x), an error is large around x=1. Therefore, precautions must be taken. Note the error range below.

Absolute error for acos (1.0 – ε) double precision 2^{-39} ($\varepsilon = 2^{-33}$)
single precision 2^{-21} ($\varepsilon = 2^{-19}$)

Absolute error for asin (1.0 – ε) double precision 2^{-39} ($\varepsilon = 2^{-28}$)
single precision 2^{-21} ($\varepsilon = 2^{-16}$)

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
}
```

```
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 cvalue1;
   // error

const cvalue2 = 1;
   // internal
```

```
const cvalue1=0;
   // gives initial value

extern const cvalue2 = 1;
   // links externally
   // as a C program
```

(3) Assignment of void*

In C++ programs, if explicit casting is not used, assignment of pointers to other objects (excluding pointers to functions and to members) is not possible.

```
void func(void *ptrv, int *ptri)
{
    ptri = ptrv;
   // error
}
```

```
void func(void *ptrv, int
*ptri)
{
    ptri = (int *)ptrv;
   //OK
}
```

9.4.3 Important Information on Program Development

Important information for program development, from program creation to debugging, is described below.

(1) Information concerning selection of the CPU

(a) The same CPU should be specified at compile time and assembly time.

The CPU specified using the cpu option at compile and assembly time must always be the same. If object programs created for different CPUs are linked, operation of the object program at runtime is not guaranteed.

(b) The same CPU type as the CPU 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 at compile time.

(c) At link time, the standard library appropriate to the CPU should be linked.

A library appropriate to the CPU should always be specified. Operation in the event that an inappropriate library is linked is not guaranteed.

(2) Important information on function interface

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.

— endian = big | little (SH-3, SH-3E, SH-4)
— pic = 0 | 1 (excluding SH-1)
— fpu = single | double (SH-4)
— fpscr = safe | aggressive (SH-4)
— round = zero | nearest (SH-4)
— denormalization = off | on (SH-4)
— double = float (excluding SH-4)
— exception | noexception
— rtti = on | off

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

No.	Item	Compiler Specifications
1	Elements of source character sets and execution environment character sets	Source program character sets and execution environment character sets are both ASCII character sets. However, string literals and character constants can be written in shift JIS code, or EUC Japanese character 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-bit
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 characters not stipulated in language specifications and integer character constants that include extended notation	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 least significant two bytes or least significant one byte of the integer value are the post-conversion 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	–2147483648 to 2147483647	4 bytes
7	unsigned int	0 to 4294967295	4 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 Number Values

		Limits	
No.	**Item**	**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

Notes: 1. The limits for decimal notation are 0 or infinity. Values in parentheses are theoretical values.

2. If double=float is specified, double type is treated as float type. If fpu=single is specified, double and long double types are treated as float type. If fpu=double is specified, float type is treated as double type.

(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 long type
2	Conversion from pointer type to integer type (pointer type size >= integer type size)	Value of least significant byte of pointer type
3	Conversion from pointer type to integer type (pointer type size < integer type size)	Sign extension
4	Conversion from integer type to pointer type (integer type size >= pointer type size)	Value of least significant byte of integer type
5	Conversion from integer type to pointer type (integer type size < pointer type size)	Sign extension
6	Integer type (ptrdiff_t) required to hold difference between pointers to members in the same array	int type

(7) Registers

Table 10.9 Register Specifications

No.	Item	Compiler Specifications
1	Maximum number of register variables that can be assigned to registers	7: char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, pointer 4: float*1 2: double*1
2	Types of register variables that can be assigned to registers	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, float*1, double *1, pointer

Note: *1. When cpu=sh2e, sh3e, or sh4.

(8) Classes, Structures, Unions, Enumeration Types, and Bit Fields

Table 10.10 Class, Structure, Union, 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 and structure members	The maximum data size of the class and structure members is used as the boundary alignment value. For details on assignment, see section 10.1.2 (2), Compound Type (C), Class Type (C++).
3	Sign of bit fields of simple int types	signed int type
4	Order of bit fields within int type size	Assigned from most significant bit.
5	Method of assignation when the size of a bit field assigned after a bit field is assigned within an int type size exceeds the remaining size in the int type	Assigned to next int type area.
6	Permissible type specifiers in bit fields	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, enum type
7	Integer type representing value of enumeration type	int type

For details of assignment of bit fields, see section 10.1.2 (3), Bit Fields.

(9) Modifiers

Table 10.11 Modifier 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 (arithmetic types, structure types, union types) modifying basic types	16 max.

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

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

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

(11) Statements

Table 10.13 Statement Specifications

No.	Item	Compiler Specifications
1	Number of case labels that can be declared in one switch statement	511 max.

(12) Preprocessor

Table 10.14 Preprocessor Specifications

No.	Item	Compiler Specifications
1	Relationship between single-character character constants in constant expressions in a conditional compile, and character sets in the execution environment	Preprocessor statement character constants are the same as the execution environment character set.
2	Method of reading include files	Files enclosed in “<” and “>” are read from the directory specified in the include option. If the specified file is not found, the directory specified in environment variable SHC_INC is searched, followed by the system directory (SHC_LIB).
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	Space characters in string literals after a macro is expanded	A string of space characters are expanded as one space character.
5	Operation of #pragma statements	See section 10.2.1, #pragma Extension.
6	_ _DATE_ _ and _ _TIME_ _ value	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 data type and the internal data representation. 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 three types of alignment; 1-byte alignment in which data can be allocated to any address, 2-byte alignment in which data is allocated to even byte addresses, and 4-byte alignment in which data is allocated to addresses of multiples of four bytes.
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
char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
signed char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
unsigned char	1	1	Unused	0	$2^8 - 1$ (255)
short	2	2	Used	-2^{15} (–32768)	$2^{15} - 1$ (32767)
unsigned short	2	2	Unused	0	$2^{16} - 1$ (65535)
int	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
unsigned int	4	4	Unused	0	$2^{32} - 1$ (4294967295)
long	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
unsigned long	4	4	Unused	0	$2^{32} - 1$ (4294967295)
enum	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
float	4*[3]	4	Used	–∞	+∞
double **long double**	8*[1, 3]	4	Used	–∞	+∞
Pointer	4	4	Unused	0	$2^{32} - 1$ (4294967295)
bool*[2]	4	4	Used	—	—
Reference*[2]	4	4	Unused	0	$2^{32} - 1$ (4294967295)
Pointer to a data member*[2]	4	4	Used	0	$2^{32} - 1$ (4294967295)
Pointer to a function member*[2, 4]	12	4	—	—	—

Notes:
1. The size of double type is 4 bytes if double=float option is specified.
2. These data types are valid for C++ compilation only.
3. If cpu=sh4 and fpu=single options are both specified, double type and long double type are treated as 4 bytes (float type). If cpu=sh4 and fpu=double are both specified, float type is treated as 8 bytes (double type).
4. Pointers to function and virtual function members are represented by classes in the following data structure.

```
class _PMF{
  public:
    long d;                          //Object offset value.
    long i;                          //Index in the virtual
                                     //function table when
                                     //the target function is the
                                     //virtual function.
    union{
      void (*f)();                   //Address of a function when
                                     //the target function is a
                                     //non-virtual function.
      long offset;                   //Object offset value of the
                                     //virtual function table
                                     //when the target function
                                     //is the virtual function.
    };
};
```

(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	Array element alignment	Number of array elements × element size	`char a[10];` Alignment: 1 byte Size: 10 bytes

Structure	Maximum structure member alignment	Total size of members. Refer to Structure Data Allocation, below.	`struct {`
			`  char a,b;`
			`};`
			Alignment: 1 byte
			Size: 2 bytes

Union	Maximum union member alignment	Maximum size of member. Refer to Union Data Allocation, below.	`union {`
			`  char a,b;`
			`};`
			Alignment: 1 byte
			Size: 1 byte

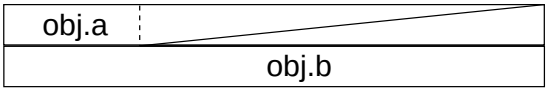
Class	1. Always 4 if a virtual function is included
	2. Other than 1 above: maximum member alignment
		Sum of data members, pointer to the virtual function table, and pointer to the virtual base class Refer to Class Data Allocation, below.	`class B:public A {`
			`  virtual void f();`
			`};`
			Alignment: 4 bytes
			Size: 8 bytes
			`class A {`
			`  char a;`
			`};`
			Alignment: 1 byte
			Size: 1 byte

In the following examples, a rectangle indicates four bytes. The diagonal line represents blank area for alignment.

Structure Data Allocation:

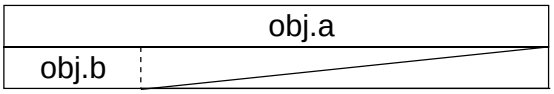
- When structure members are allocated, an unused area may be generated between structure members to align data types.

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



- If a structure has 4-byte alignment and the last member ends at an 1-, 2-, or 3-byte address, the following 3-, 2-, or 1-byte is included in this structure.

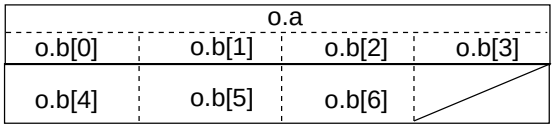
```
struct {
 int a;
 char b;
} obj
```



Union Data Allocation:

- When an union has 4-byte alignment and its maximum member size is not a multiple of four, the remaining bytes up to a multiple of four is included in this union.

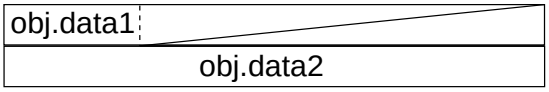
```
union {
 int a;
 char b[7];
} o;
```



Class Data Allocation:

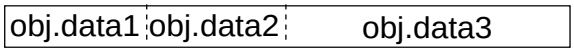
- For classes having no base class or virtual functions, data members are allocated according to the allocation rules of structure data.

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



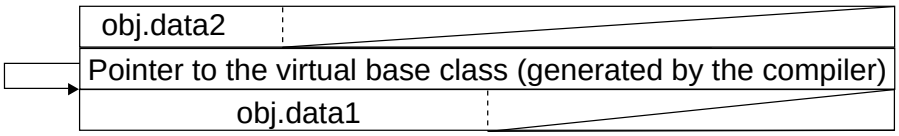
- If a class is derived from a base class of 1-byte alignment and the start member of the derived class is 1-byte data, data member is allocated without padding.

```
class A{
  char data1;
};
class B:public A{
  char data2;
  short data3;
}obj;
```



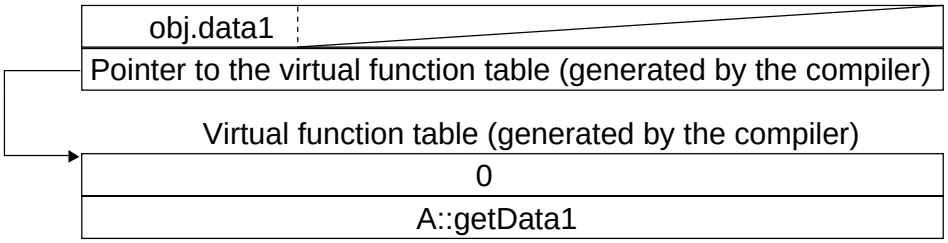
- For a class having a virtual base class, a pointer to the virtual base class is allocated.

```
class A{
    short data1;
};
class B: virtual protected A{
    char data2;
}obj;
```



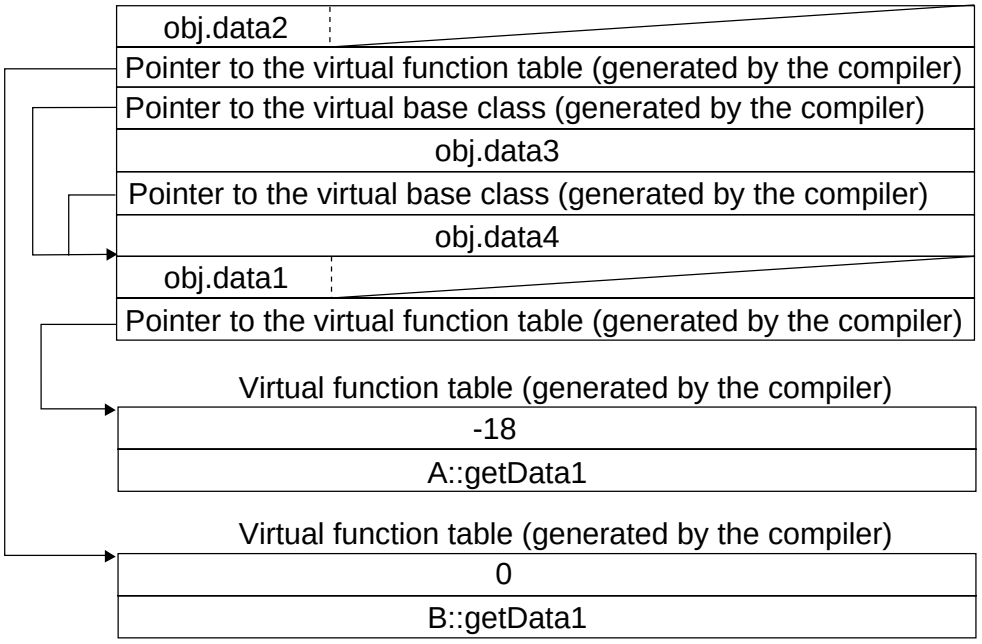
- For a class having virtual functions, the compiler creates a virtual function table and allocates a pointer to the virtual function table.

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



- An example is shown for class having virtual base class, base class, and virtual functions.

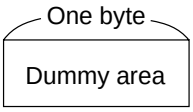
```
class A{
  char data1;
  virtual short getData1();
};
class B:virtual public A{
  char data2;
  char getData2();
  short getData1();
};
class C:virtual protected A{
  int data3;
};
class D:virtual public A,public B,public C{
  public:
  int data4;
  short getData1();
}obj;
```



Note that non-virtual function does not occupy an area in a class.

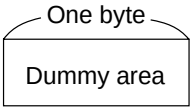
- For an empty class, a 1-byte dummy area is assigned.

```
class A{
  void fun();
}obj;
```



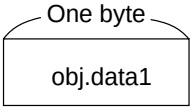
- For an empty class having empty class as its base class, the dummy area is 1 byte.

```
class A{
  void fun();
};
class B: A{
  void sub();
}obj;
```



- Dummy areas shown in the above two examples are allocated only when the class size is 0. No dummy aria is allocated if a base class or a derived class has a data member or has a virtual function.

```
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 or a 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

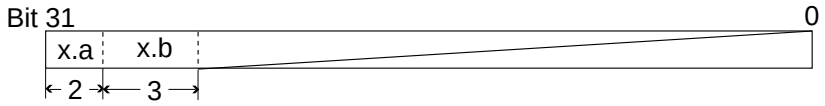
Item	Specifications
Type specifier allowed for bit fields	(signed) char, unsigned char, bool*[1] (signed) short, unsigned short, enum (signed) int, unsigned int (signed) long, unsigned long
How to treat a sign when data is extended to the declared type*[2]	A bit field with no sign (unsigned is specified for type): Zero extension*[3] A bit field with a sign (unsigned is not specified for type): Sign extension*[4]

Notes:
1. The bool type is only valid at C++ compilation.
2. To use a bit field member, data in the bit field is extended to the declared type. One-bit field data with a sign is interpreted as the sign, and can only indicate 0 and –1. To indicate 0 and 1, bit field data must be declared with unsigned.
3. Zero extension: Zeros are written to the upper bits to extend data.
4. Sign extension: The most significant bit of a bit field is used as a sign and the sign is written to all higher-order bits to extend data.

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

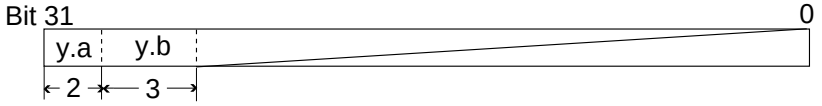
- Bit field members are placed in an area beginning from the left, that is, the most significant bit.

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



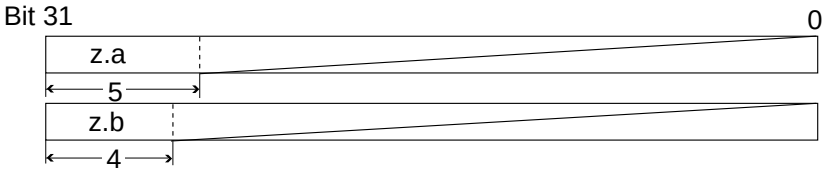
- Consecutive bit field members having type specifiers of the same size are placed in the same area as much as possible.

```
struct b1 {
  long           a:2;
  unsigned int   b:3;
} y;
```



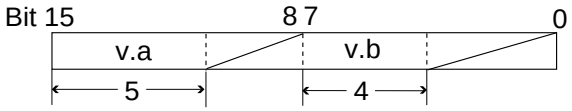
- Bit field members having type specifiers with different sizes are allocated to the separate areas.

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



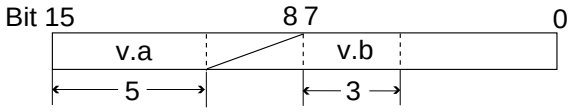
- If the number of remaining bits in the area is less than the next bit field size, though type specifier indicate the same size, the remaining area is not used and the next bit field is allocated to the next area.

```
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;
```



(4) Memory Allocation of Little Endian

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

One-byte data ((signed) char, unsigned char, and bool type): The order of bits in one-byte data for a big endian and a little endian is the same.

Two-byte data ((signed) short and unsigned short type): The upper byte and the lower byte will be reversed in two-byte data between a big endian and a little endian.

Example: When a two-byte data 0x1234 is allocated at an address 0x100:

big endian:	address 0x100: 0x12	little endian:	address 0x100: 0x34
	address 0x101: 0x34		address 0x101: 0x12

Four-byte data ((signed) int, unsigned int, (signed) long, unsigned long, and float type): The upper byte and the lower byte will be reversed in four-byte data between a big endian and a little endian.

Example: When a four-byte data 0x12345678 is allocated at an address 0x100:

big endian:	address 0x100: 0x12	little endian:	address 0x100: 0x78
	address 0x101: 0x34		address 0x101: 0x56
	address 0x102: 0x56		address 0x102: 0x34
	address 0x103: 0x78		address 0x103: 0x12

Eight-byte data (double type): The upper byte and lower byte will be reversed in eight-byte data between a big endian and a little endian.

Example: When an eight-byte data 0x123456789abcdef is allocated at an address 0x100:

big endian:	address 0x100: 0x01	little endian:	address 0x100: 0xef
	address 0x101: 0x23		address 0x101: 0xcd
	address 0x102: 0x45		address 0x102: 0xab
	address 0x103: 0x67		address 0x103: 0x89
	address 0x104: 0x89		address 0x104: 0x67
	address 0x105: 0xab		address 0x105: 0x45
	address 0x106: 0xcd		address 0x106: 0x23
	address 0x107: 0xef		address 0x107: 0x01

Compound-type and class-type data: Members of compound-type and class-type data will be allocated in the same way as that of a big endian. However, the order of byte data of each member will be reversed according to the rule of data size.

Example: When the following function exists at an address 0x100:

```
struct {
   short a;
   int b;
}z= {0x1234,  0x56789abc};
```

big endian:	address 0x100: 0x12	little endian:	address 0x100: 0x34
	address 0x101: 0x34		address 0x101: 0x12
	address 0x102: empty area		address 0x102: empty area
	address 0x103: empty area		address 0x103: empty area
	address 0x104: 0x56		address 0x104: 0xbc
	address 0x105: 0x78		address 0x105: 0x9a
	address 0x106: 0x9a		address 0x106: 0x78
	address 0x107: 0xbc		address 0x107: 0x56

Bit field: Bit fields will be allocated in the same way as a big endian. However, the order of byte data in each area will be reversed according to the rule of data size.

Example: When the following function exists at an address 0x100:

```
struct {
   long a:16;
   unsigned int b:15;
   short c:5
}y= {1,1,1};
```

big endian:	address 0x100: 0x00	little endian:	address 0x100: 0x02
	address 0x101: 0x01		address 0x101: 0x00
	address 0x102: 0x00		address 0x102: 0x01
	address 0x103: 0x02		address 0x103: 0x00
	address 0x104: 0x08		address 0x104: 0x00
	address 0x105: 0x00		address 0x105: 0x08
	address 0x106: empty area		address 0x106: empty area
	address 0x107: empty area		address 0x107: empty area

10.1.3 Floating-Point Number Specifications

(1) Internal Representation of Floating-Point Numbers

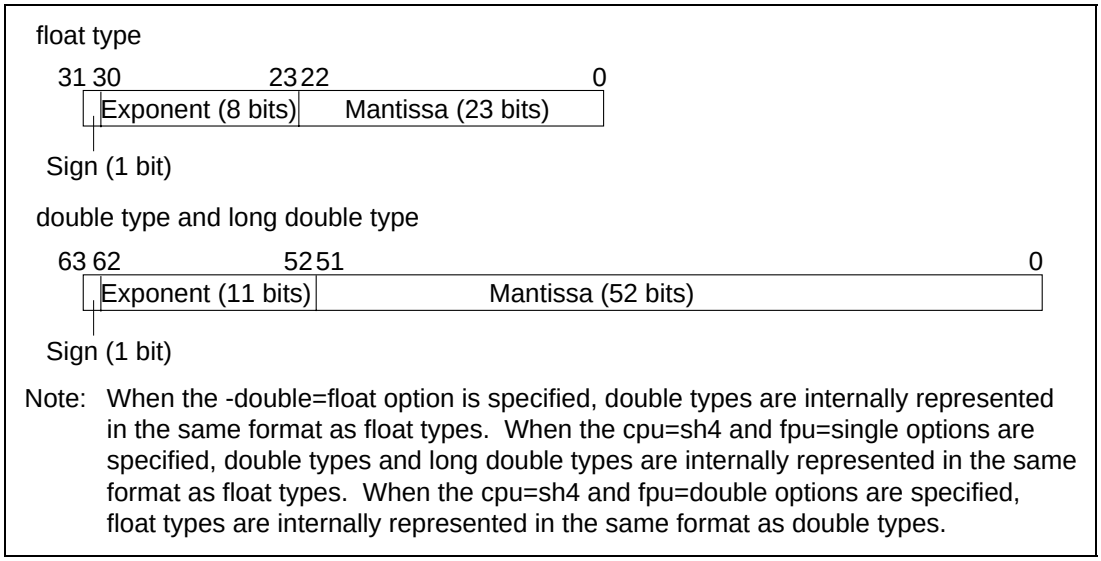
Floating-point numbers handled by this compiler are internally represented in the standard IEEE format. This section outlines the internal representation of floating-point numbers in the IEEE format.

(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.



Note: When the -double=float option is specified, double types are internally represented in the same format as float types. When the cpu=sh4 and fpu=single options are specified, double types and long double types are internally represented in the same format as float types. When the cpu=sh4 and fpu=double options are specified, float types are internally represented in the same format as 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 as a power of 2.

iii. Mantissa

Shows the data corresponding to the significant digits (fraction) 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 numbers

Represents normal real values; the exponent is not 0 or not all bits are 1.

ii. Denormalized numbers

Represents real values having small absolute numbers; the exponent is 0 and the mantissa is other than 0.

iii. Zero

Represents the value 0.0; the exponent and mantissa are 0.

iv. Infinity

Represents infinity; all bits of the exponent are 1 and the mantissa is 0.

v. Not-a-number

Represents the result of operation such as "0.0/0.0", "∞/∞", or "∞-∞", which does not correspond to a number or infinity; all bits of the exponents are 1 and the mantissa is other than 0.

Table 10.18 shows the types of values represented as floating-point numbers.

Table 10.18 Types of Values Represented as Floating-Point Numbers

	Exponent		
Mantissa	**0**	**Not 0 or not all bits are 1**	**All bits are 1**
0	0	Normalized number	Infinity
Other than 0	Denormalized number		Not-a-number

Note: Denormalized numbers are floating-point numbers of small absolute values that are outside the range represented by normalized numbers. There are fewer valid digits in a denormalized number than in a normalized number. Therefore, if the result or intermediate result of a calculation is a denormalized number, the number of valid digits in the result cannot be guaranteed. When the CPU is an SH-4, denormalized numbers are processed as 0 when option
denormalize=off is specified. When denormalize=on is specified, denormalized numbers are processed as denormalized numbers.

(2) float type

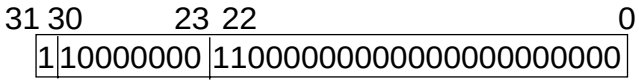
float types are internally represented by a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.

i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 254 (2^8–2). The actual exponent is gained by subtracting 127 from this value. The range is between −126 and 127. The mantissa is between 0 and 2^{23}–1. The actual mantissa is interpreted as the value of which 2^{23}rd bit is 1 and this bit is followed by the decimal point. Values of normalized numbers are as follows:

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

Example:



Sign: –

Exponent: $10000000_{(2)} - 127 = 1$, where $_{(2)}$ indicates binary

Mantissa: $1.11_{(2)} = 1.75$

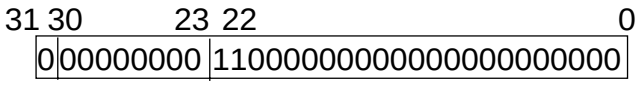
Value: $-1.75 \times 2^1 = -3.5$

ii. Denormalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is 0 and the actual exponent is −126. The mantissa is between 1 and 2^{23}–1, and the actual mantissa is interpreted as the value of which 2^{23}rd bit is 0 and this bit is followed by the decimal point. Values of denormalized numbers are as follows:

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

Example:



Sign: +

Exponent: −126

Mantissa: $0.11_{(2)} = 0.75$, where $_{(2)}$ indicates binary

Value: 0.75×2^{-126}

iii. Zero

The sign is 0 (positive) or 1 (negative), indicating +0.0 or –0.0, respectively. The exponent and mantissa are both 0.

+0.0 and –0.0 are both the value 0.0. See section 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 +∞ or –∞, respectively.
The exponent is 255 (2^8–1).

The mantissa is 0.

v. Not-a-number

The exponent is 255 (2^8–1).

The mantissa is a value other than 0.

Note: When the CPU is SH-2E, SH-3E, or SH-4, not-a-number is called a qNaN when the MSB of the mantissa is 0, or sNaN when the MSB of the mantissa is 1. There are no specifications regarding the values of other mantissa fields or the sign.

(3) double types and long double types

double types and long double types are internally represented by a 1-bit sign, an 11-bit exponent, and a 52-bit mantissa.

i. Normalized numbers

The sign indicates the sign of the value, either 0 (positive) or 1 (negative). The exponent is between 1 and 2046 ($2^{11}-2$). The actual exponent is gained by subtracting 1023 from this value. The range is between –1022 and 1023. The mantissa is between 0 and $2^{52}-1$. The actual mantissa is interpreted as the value of which 2^{52}nd bit is 1 and this bit is followed by the decimal point. Values of normalized numbers are as follows:

$(-1)^{sign} \times 2^{exponent-1023} \times (1+(mantissa) \times 2^{-52})$

Example:

63	62 … 52	51 … 0
0	01111111111	1110000000000000000000000000000000000000000000000000

Sign: +

Exponent: $1111111111_{(2)} - 1023 = 0$, where $_{(2)}$ indicates binary

Mantissa: $1.111_{(2)} = 1.875$

Value: $1.875 \times 2^{0} = 1.875$

ii. Denormalized numbers

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

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

Example:

63	62 … 52	51 … 0
1	00000000000	1110000000000000000000000000000000000000000000000000

Sign: –

Exponent: –1022

Mantissa: $0.111_{(2)} = 0.875$, where $_{(2)}$ indicates binary

Value: 0.875×2^{-1022}

iii. Zero

The sign is 0 (positive) or 1 (negative), indicating +0.0 or –0.0, respectively. The exponent and mantissa are both 0.

+0.0 and –0.0 are both the value 0.0. 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 +∞ or –∞, respectively. The exponent is 2047 (2^{11}–1).

The mantissa is 0.

v. Not-a-number

The exponent is 2047 (2^{11}–1).

The mantissa is a value other than 0.

Note: When the CPU is SH-2E, SH-3E, or SH-4, not-a-number is called a qNaN when the MSB of the mantissa is 0, or sNaN when the MSB of the mantissa is 1. There are no specifications regarding the values of other mantissa fields or the sign.

(4) Floating-Point Operation Specifications

This section describes the specifications for arithmetic operations on floating-point numbers in C/C++, and for converting between the decimal representation of floating-point numbers and their internal representation during compilation and in library processing.

(a) Specifications for arithmetic operations

i. Rounding of results

When the result of arithmetic operations on floating-point numbers exceeds the number of valid 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.

c. When the CPU is SH-2E or SH-3E, the portion that exceeds the valid digits is truncated.

d. When the CPU is SH-4, and the round = nearest option is specified, the portion that exceeds the valid digits is rounded to the nearest value. When the round = zero option is specified, the portion that exceeds the valid digits is rounded toward zero.

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. If an overflow results from converting a floating point number to an integer, the result is not guaranteed.

Note: Operations are performed on constant expressions during compilation. If an overflow, underflow, or illegal operation occurs, a warning level error message is output.

iii. Notes on operations on special values

The following are notes on operations on special values (zero, infinity, and not-a-number).

a. The sum of a positive zero plus negative zero is a positive zero.

b. The difference between two zeros of the same sign is a positive zero.

c. The result of operations that include not-a-number in one or both operands is always a not-a-number.

d. In comparative operations, positive zeros and negative zeros are processed as equal.

e. The result of comparative operations or equivalence operations where either one or both operands are not-a-number is true for "!=" and false in all other cases.

(b) Conversion between decimal and internal representation

This section describes the specifications for conversions between floating-point numbers in a source program and internal representation, and conversion by library functions between the decimal representation of floating-point numbers in ASCII strings and their internal representation.

i. When converting from decimal to internal representation, the decimal value is first converted to its normalized form. The normalized form of a decimal value is $\pm M \times 10^{\pm N}$, where M and N are in the following range:

a. Normalized form of float types

$0 \leq M \leq 10^{9}-1$

$0 \leq N \leq 99$

b. Normalized form of double and long double types

$0 \leq M \leq 10^{17}-1$

$0 \leq N \leq 999$

If a decimal value cannot be converted to its normalized form, an overflow or underflow occurs. If the decimal representation contains more valid numerals than the normalized form, the trailing digits are truncated. In this case, a warning level error message is output when compiling and the corresponding error number is set in **errno** when the program is executed. For conversion to its normalized form, the original decimal representation must, in the form of ASCII strings, be within 511 characters. If not, an error occurs when compiling and the corresponding error number is set in **errno** when the program is executed. When converting from internal representation to decimal, the value is first converted to the normalized decimal form, then converted to ASCII strings according to the specified format.

ii. Conversion between normalized form of decimals and internal representation

When converting from the normalized form of decimals to internal representation, and vice versa, errors cannot be avoided when the exponent is large or small. The following describes the range within which conversion is accurate, and the error limits when the values are outside that range.

a. Range for accurate conversion

The rounding shown in (a) i, "Rounding of results" is correctly applied for floating-point numbers within the ranges shown below. No overflow or underflow will occur within these ranges.

(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 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 Specifications

The compiler supports the following two kinds of extended specifications:

- **#pragma** extension specifiers
- Intrinsic functions

10.2.1 #pragma Extension

Tables 10.20 to 10.22 list **#pragma** extension specifiers.

Table 10.20 Extended Specifications Relating to Memory Allocation

#pragma Extension Specifier	Function
#pragma section	Switches sections
#pragma abs16	Treats a variable or function address as two-byte data

Table 10.21 Extended Specifications Relating to Functions

#pragma Extension Specifier	Function
#pragma interrupt	Creates an interrupt function
#pragma inline	Performs inline expansion of functions
#pragma inline_asm	Expands an assembly-language description function.
#pragma regsave, #pragma noregsave, #pragma noregalloc	Generates or does not generate save and restore code at the start and end of functions

Table 10.22 Other Extended Specifications

#pragma Extension Specifier	Function
#pragma global_register	Allocates global variables to registers
#pragma gbr_base, #pragma gbr_base1	Specifies GBR base variables

For some of the extended functions above, data members and function members can be specified. Specification format is (class name::member name). For the specifiable member types, see the format of each function.

(1) Extended Specifications Related to Memory Allocation

#pragma section

Description Format: #pragma 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>
Non-initialized data area		B*	B<xx>

Note: The default section name can be modified by the **section** option.
If <name> and <number> are 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. **#pragma section** can be declared only outside the function definition.
2. Up to 64 section names can be declared for each of **#pragma section** in one file.

#pragma abs16

Description Format: #pragma abs16 (<identifier> [,…])

Description: A variable allocated at addresses H'00000000 to H'00007FFF or H'FFFF8000 to H'FFFFFFFF specified using an identifier or an address of a function is treated as two-byte data. Then, program size can be reduced.
For the identifier, variables, global functions, static data members, and function members can be specified.

Example:

```
#pragma abs16(x, y, z)
extern int x();
int y;
long z;
void f(void){
  z = x()+ y;
}
```

Remarks:

1. Directive **#pragma abs16** cannot be used to specify an automatic object or non-static data member.
2. Variables declared using directive **#pragma abs16** must be allocated in the address range from H'00000000 to H'00007FFF or from H'FFFF8000 to H'FFFFFFFF.

(2) Extended Specifications Related to Functions

#pragma interrupt

Description Format: #pragma interrupt (<function name>[(interrupt specification)][,…])

Description: Declares an interrupt function.
Global functions and static function members can be specified for the function name. Table 10.24 lists interrupt specifications.

Table 10.24 Interrupt Specifications

Item	Form	Options	Specifications
Stack switching	sp=	{<variable> |&<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>	The interrupt function exits with the TRAPA instruction. <constant>: Constant value (trap vector number)

An interrupt function will guarantee register values before and after processing (all registers used by the function are pushed onto and popped from the stack when entering and exiting the function). The **RTE** instruction directs the function to return. However, if the trap-instruction return is specified, the **TRAPA** instruction is executed at the end of the function. An interrupt function with no specifications is processed in the usual procedure. The stack switching specification and the trap-instruction return specification can be specified together.

Example:

```
#pragma interrupt(f(sp = ptr, tn = 10),A::g)
extern int  STK[100];
class A{
public:
  static void g();
};
int *ptr = STK +100;
```

Explanation:

(a) Stack switching specification: **ptr** is set as the stack pointer used by interrupt function **f**.

(b) Trap-instruction return specification: After the interrupt function has completed its processing, **TRAPA #H'10** is executed. The SP at the beginning of trap exception processing is shown in figure 10.2. After the previous PC (program counter) and SR (status register) are popped from the stack by the **RTE** instruction in the trap routine, control is returned from the interrupt function.

(c) The function member that can be specified in C++ program is the static function member. In the example, static function member g of class A is specified as an interrupt function. Note that nonstatic function members cannot be specified.

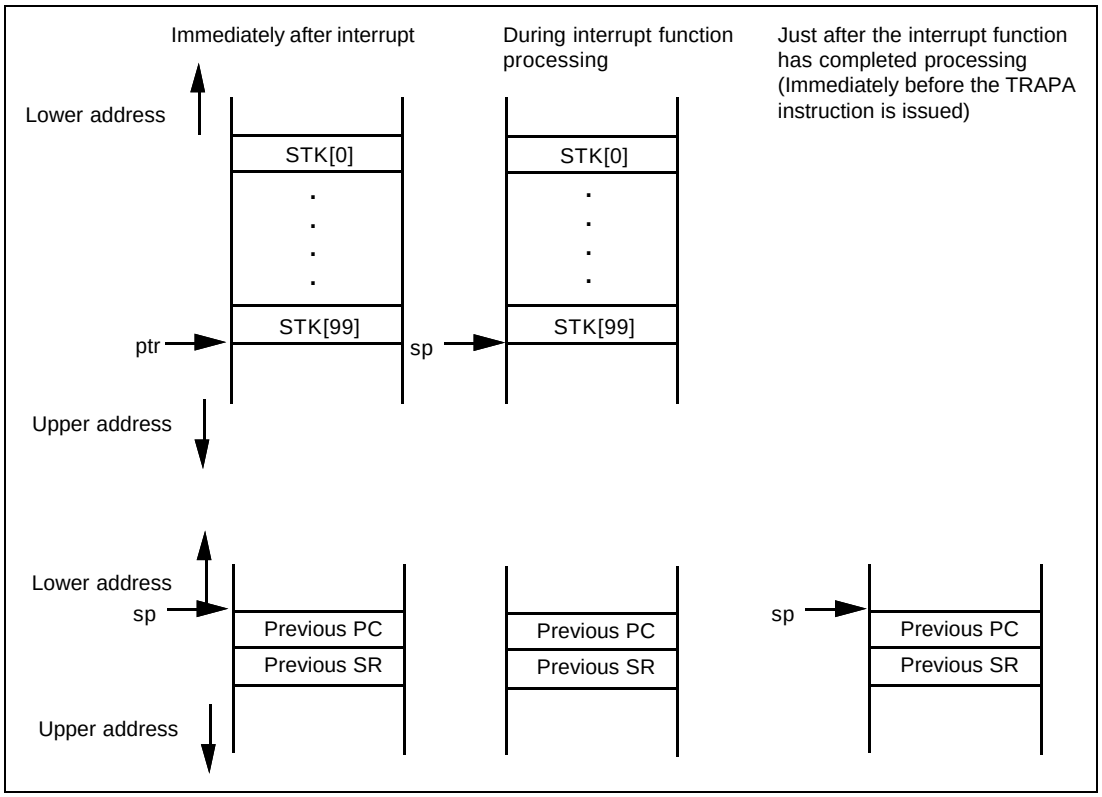


Figure 10.2 Stack Processing by an Interrupt Function

Remarks:

1. The interrupt operation in the SH-3, SH-3E, and SH-4 is different from that in the SH-1, SH-2, and SH-2E, and requires interrupt handlers.

2. Functions that can be specified for an interrupt function definition are the global function (in C/C++ program) and static function member (in C++ program). A global function is assumed to be of the extern storage class even if **static** is specified.

   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(){...} ..................... (a)
int f2(){...}  ..................... (b)
```

   Description: (a) is a correct declaration.
   (b) returns type that is not void, thus (b) is an incorrect declaration. An error will occur.

3. A function declared as an interrupt function cannot be called within the program. If attempted, an error will occur. However, if the function is called within a program which does not have a declaration of the interrupt function, an error does not occur but correct program execution is not guaranteed.

   Example 1 (An interrupt function is declared):

```
#pragma interrupt(f1)
void f1(){...}
int f2(){ f1();} ............. (a)
```

   Description: Function f1 cannot be called in the program because it is declared as an interrupt function. An error occurs at (a).

   Example 2 (An interrupt function is not declared):

```
int f1();
int f2(){ f1();} ............. (b)
```

   Description: Because function f1 is not declared as an interrupt function, an object is generated as a non-interrupt function, **int f1();**. If function f1 is declared as an interrupt function in another file, correct program execution cannot be guaranteed.

#pragma inline

Description Format: #pragma inline (<function name>[,…])

Description: Declares a function for which inline expansion is performed.
A name of a global function or a static function member can be specified as a function name.
A function specified by **#pragma inline** or a function with specifier inline (C++) will be expanded where the function is called.

Example: Source Program

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

Inline Expansion Image

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

Remarks:

1. A function will not be expanded in the following cases:
   — a function definition exists before the **#pragma inline** specification
   — a function has variable parameters
   — a parameter address is referenced in a function
   — an address of a function to be expanded is used to call the function
2. Specify **#pragma inline** before defining a function.
3. When a program file includes the definition of a function to be inlined, be sure to specify **static** before the function declaration because an external definition is generated for a function specified by **#pragma inline**. If **static** is specified, an external definition will not be created. External definition will not be created for functions for which inline (C++) is specified.

#pragma inline_asm

Description Format: #pragma inline_asm (<function name>[(size=numeric value)][,...])

Description: Performs inline expansion for the functions written in assembly language declared by **#pragma inline_asm**.
Only a global function can be specified as a function name. Function members cannot be specified.

Parameters of a function that is written in an assembly language are referenced from an **inline_asm** function because they are stacked or stored in registers in the same way as general function calls. Return values of an inline function written in an assembly language should be set in R0. When the **cpu** is SH-2E, SH-3E, or SH-4, return values of single-precision floating point type should be set in FR0. When the **cpu** is SH-4, return values of double-precision floating point type should be set in DR0. A different register may be used depending on the combination of options. For details, see table 9.7.
Length of an inline function written in an assembly language can be specified by (size=numeric value).

Example: Source program

```
#pragma inline_asm(rot1)
static int rotl (int a)
{
  ROTL R4
  MOV R4,R0
}
int x;
main( )
{
  x = 0x55555555;
  x = rotl(x);
}
```

Output result (partial)

```
      :
_main                                ;function main
                                     ;frame size = 4
      MOV.L    R14,@-R15
      MOV.L    L220+2,R14            ;_x
      MOV.L    L220+6,R3             ;H'55555555
      MOV.L    R3,@R14
      MOV      R3,R4
      BRA      L219
      NOP
L220:
     .RES.W    1
     .DATA.L   _x
     .DATA.L   H'55555555
L219:
     ROTL      R4
     MOV       R4,R0
     .ALIGN  4
     MOV.L     R0,@R14
     RTS
     MOV.L     @R15+,R14
     .SECTION  B,DATA,ALIGN=4
_x:                                  ;static: x
     .RES.L    1
     .END
```

Remarks:

1. Specify **#pragma inline_asm** before the definition of a function.
2. 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 for a function specified by **#pragma inline_asm**. If **static** is specified, an external definition will not be created.
3. Be sure to use local labels in a function written in an assembly language.
4. When registers R8 to R14 are used in a function written in an assembly language, the contents of these registers must be saved and restored at the start and end of the function. Also, when registers FR12 to FR15 (in SH-2E, SH-3E, or SH-4 cpu) are used, or when registers DR12 to DR14 (with the SH-4 cpu) are used, the contents of these registers must be saved and restored at the start and end of the inline function written in the assembly language.
5. Do not use **RTS** at the end of a function written in an assembly language.
6. When **#pragma inline_asm** is used, be sure to compile programs using the **code=asmcode** option to generate assembly code.
7. When specifying a number by (**size=numeric value**), specify a number larger than the actual object size. If a value smaller than the actual object size is specified, correct operation will not be guaranteed. If a floating point or a numeric value less than 0 is specified, an error will occur.
8. Even when a register specified by the **#pragma global_register** function is used, the contents of this register must be saved and restored at the start and end of the inline function written in an assembly language.
9. Only a global function can be specified as a function name. Function members cannot be specified.
10. Do not use a statement that generates a literal pool. (**MOV.L #100000,R0** etc.)

#pragma regsave
#pragma noregsave
#pragma noregalloc

Description Format: #pragma regsave (<function name>[,…])
#pragma noregsave (<function name>[,…])
#pragma noregalloc (<function name>[,…])

Description:

1. Global functions and function members can be specified as the function name.
2. Functions specified by **#pragma regsave** save and restore the contents of callee-save registers (see table 9.5) at the start and end of a function, respectively. Inside the function specified by **#pragma regsave**, callee-save registers (R8 to R14, and FR12 to FR15 if FPU exists) will not carry a value over a child function call.
3. Functions specified by **#pragma noregsave** do not save or restore the contents of callee-save registers at the start and end of a function.
4. Functions specified by **#pragma noregalloc** do not save or restore the contents of callee-save registers at the start and end of a function. Inside the function specified by **#pragma regsave**, callee-save registers (R8 to R14, and FR12 to FR15 if FPU exists) will not carry a value over a child function call.
5. **#pragma regsave** and **#pragma noregalloc** can specify the same function at the same time. In this case, the contents of registers R8 to R14 (and FR12 to FR15 if FPU exists) are saved and restored at the start and end of a function if they are used. Inside the function specified by **#pragma regsave**, callee-save registers (R8 to R14, and FR12 to FR15 if FPU exists) will not carry a value over a child function call.
6. Functions specified by **#pragma noregsave** can be used in the following conditions:
   a. A function is the first function activated and is not called from any other function.
   b. A function is called from a function that is specified by **#pragma regsave**.
   c. A function is called from a function that is specified by **#pragma regsave** via **#pragma noregalloc**.

Example:

```
#pragma noregsave(f, A::j)
#pragma noregalloc(g)
#pragma regsave(h)
class A{
public:
  static void j();
};
void f();
void g();
void h();
void h()
{
   g( );
   f( );   /* Function f declared with #pragma         */
           /* noregsave is directly called by h        */
}          /* declared with #pragma regsave            */

void g( )
{
   f( );   /* Functions f and A::j declared with       */
           /* #pragma noregsave are indirectly called  */
           /* by h via g declared with #pragma         */
           /* noregalloc                               */
   A::j();
}

void f( )
{
}
```

Remarks: The result of a call of a function declared with **#pragma noregsave** is not guaranteed if it is called in a way other than that shown above.

(3) Other Extended Specifications

#pragma global_register

Description Format: #pragma global_register (<variable name>=<register name>[,…])

Description: Allocates the global variable specified in <variable name> to the register specified in <register name>.
Global variables and static data members can be specified as the variable name.

Example:

```
#pragma global_register(a = R8,A::b = R9)
class A(
public:
static int b;
);
int a;
void g()
{
a = A::b;
}
```

Remarks:

1. This function is used for a simple or pointer type variable in the global variable. In a CPU other than SH-4, a double type variable can be specified only when **double=float** option is specified.
2. Only use registers R8 to R14, FR12 to FR15 (in SH-2E, SH-3E, or SH-4 cpu) and DR12 to DR14 (in SH-4 cpu).
3. The initial value cannot be set. In addition, the address of the specified variable cannot be referenced.
4. The reference of the specified variable from outside of the file is not guaranteed.
5. Static data members can be specified. Nonstatic data members cannot be specified.
   - Type of variables that can be set in FR12 to FR15:
     For SH-2E and SH-3E cpu
     float type variables
     double type variables (when **double=float** option is specified)
     For SH-4 cpu
     float type variables (when **fpu=double** option is not specified)
     double type variables (when **fpu=single** option is specified)
   - Type of variables that can be set in DR12 to DR14
     For SH-4 cpu
     float type variables (when **fpu=double** option is specified)
     double type variables (when **fpu=single** option is not specified)

#pragma gbr_base
#pragma gbr_base1

Description Format: #pragma gbr_base (variable name[,...])
#pragma gbr_base1 (variable name[,...])

Description: Specifies variables to be accessed using a GBR register and an offset value. For the variable name, variables and static data members can be specified.

The variable specified by **#pragma gbr_base** is assigned to section $G0, and the variable specified by **#pragma gbr_base1** is assigned to section $G1.

The directive **#pragma gbr_base** specifies that the variable is located in an offset of 0 to 127 bytes from the address specified by the GBR register. The directive **#pragma gbr_base1** specifies that the variable is located in an offset of 128 or more bytes from the address specified by the GBR register, that is, a variable is in a range beyond the range specified by **#pragma gbr_base**. An offset value is 255 bytes at maximum for a char or unsigned char type, 510 bytes at maximum for a short or unsigned short, and 1020 bytes at maximum for an int, unsigned, long, unsigned long, float, or double type. Based on the above specification, the compiler generates an object program in a GBR relative addressing mode that is optimized according to variable reference and settings.

The compiler also generates an optimized bit instruction in the GBR indirect addressing to char or unsigned char type data in the $G0 section.

Remarks:

1. If the total data size after the linker gathers sections $G0 exceeds 128 bytes, the correct operation will not be guaranteed. In addition, if there is data that has an offset value exceeding those specified above for **#pragma gbr_base1** in section $G1, correct operation will not be guaranteed.
2. Section $G1 must be allocated immediately after 128 bytes of section $G0 in linkage.
3. In using these **#pragma**'s, be sure to set the start address of section $G0 in the GBR register at the start of program execution.
4. Static data members can be specified, but non-static data members cannot be specified.

10.2.2 Intrinsic Functions

The compiler provides functions that cannot be written in C/C++, as intrinsic functions. The following functions can be specified as intrinsic functions.

- Setting and referencing the status register
- Setting and referencing the vector base register
- I/O functions using the global base register
- System instructions which do not compete with register sources in C/C++ language
- Multimedia instructions using the floating point unit and setting and referencing control registers

Intrinsic functions can be written in the same call format as regular functions.

Table 10.25 lists intrinsic functions.

Table 10.25 Intrinsic Functions

Item	Specifications	Function
Status register (SR)	void set_cr(int cr)	Writes to the status register
	int get_cr(void)	Reads the status register
	void set_imask(int mask)	Writes to the interrupt mask bit
	int get_imask(void)	Reads the interrupt mask bit
Vector base register (VBR)	void set_vbr(void *base)	Writes to VBR
	void *get_vbr(void)	Reads VBR
Global base register (GBR)	void set_gbr(void *base)	Writes to GBR
	void *get_gbr(void)	Reads GBR
	unsigned char gbr_read_byte(int offset)	Reads a GBR-based byte
	unsigned short gbr_read_word(int offset)	Reads a GBR-based word
	unsigned short gbr_read_long(int offset)	Reads a GBR-based longword
	void gbr_write_byte (int offset, unsigned char data)	Writes a GBR-based byte
	void gbr_write_word (int offset, unsigned short data)	Writes a GBR-based word

Table 10.25 Intrinsic Functions (cont)

Item	Specification	Function
Global base register (GBR) (cont)	void gbr_write_long (int offset, unsigned long data)	Writes a GBR-based longword
	void gbr_and_byte (int offset, unsigned char mask)	ANDs a GBR-based byte
	void gbr_or_byte (int offset, unsigned char mask)	ORs a GBR-based byte
	void gbr_xor_byte (int offset, unsigned char mask)	XORs a GBR-based byte
	int gbr_tst_byte (int offset, unsigned char mask)	Tests a GBR-based byte
Special instruc-tions	void sleep(void)	SLEEP instruction
	int tas(char *addr)	TAS instruction
	int trapa(int trap_no)	TRAPA instruction
	int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)	OS system call
	void prefetch (void *p)	PREF instruction
	void trace(long v)	TRACE instruction
Multiply and accumulate operation	int macw (short *ptr1, short *ptr2, unsigned int count) int macwl (short *ptr1, short *ptr2, unsigned int count, unsigned int mask)	MAC.W instruction
	int macl (int *ptr1, int *ptr2, unsigned int count) int macll (int *ptr1, int*ptr2, unsigned int count, unsigned int mask)	MAC.L instruction
Floating point unit	void set_fpscr(int cr)	Sets FPSCR.
	int get_fpscr()	Refers to FPSCR.

Table 10.25 Intrinsic Functions (cont)

Item	Specification	Function
Single-precision floating point vector operation	float fipr(float vect1[4], float vect2[4])	FIPR instruction
	void ftrv(float vec1[4],float vec2[4])	FTRV instruction
	void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	Transforms 4-dimensional vector by 4x4 matrix, and adds the result to 4-dimensional vector
	void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	Transforms 4-dimensional vector by 4x4 matrix, and subtracts 4-dimensional vector from the result
	void add4(float vec1[4], float vec2[4], float vec3[4])	Performs addition of 4-dimension vectors
	void sub4(float vec1[4], float vec2[4], float vec3[4])	Performs subtraction of 4-dimension vectors
	void mtrx4mul(float mat1[4][4], float mat2[4][4])	Performs multiplication of 4x4 matrices

Table 10.25 Intrinsic Functions (cont)

Item	Specification	Function
Single-precision floating point vector operation (cont)	void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	Performs multiplication and addition of 4x4 matrices
	void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	Performs multiplication and subtraction of 4x4 matrices
Access to extension register	void ld_ext(float mat[4][4])	Loads mat (4x4 matrix) to extension register.
	void st_ext(float mat[4][4])	Stores contents of extension register to mat (4x4 matrix).

<machine.h>, **<umachine.h>**, or **<smachine.h>** must be specified when intrinsic functions are used.

<machine.h> is divided into <umachine.h> and <smachine.h> as shown in table 10.26 to correspond to the SH-3, SH-3E, SH-4 execution mode:

Table 10.26 Dividing <machine.h>

Include File	Contents
<machine.h>	Overall intrinsic functions
<smachine.h>	Intrinsic functions that can be used in the privileged mode
<umachine.h>	Intrinsic functions other than those in <smachine.h>

void set_cr(int cr)

Description: Sets **cr** (32 bits) to the status register (SR).

Header: <machine.h> or <smachine.h>

Parameters: cr Setting value

Example:
```
#include <machine.h>
void main(void)
{
     set_cr(0x60000000);/* Supervisor,RBank=1,BL=0,Imask0 */
}
```

int get_cr(void)

Description: Reads the status register (SR).

Header: <machine.h> or <smachine.h>

Return value: Status register value

Example:
```
#include <machine.h>
void main(void)
{
   set_cr(get_cr() | 0x1000000); /* Set BL bit */
}
```

void set_imask(int mask)

Description: Sets **mask** (4 bits) to the interrupt mask bits (4 bits).

Header: <machine.h> or <smachine.h>

Parameters: mask Setting value (4 bits)

Example:
```
#include <machine.h>
void main(void)
{
  set_imask(15);
}
```

int get_imask(void)

Description: Reads the interrupt mask bit (4 bits).

Header: <machine.h> or <smachine.h>

Return value: Value of the interrupt mask bit

Example:
```
#include <machine.h>
void main(void)
{
  int mask;
  mask = get_imask();
}
```

void set_vbr(void base)

Description: Sets base (32 bits) to the vector base register (VBR).

Header: <machine.h> or <smachine.h>

Parameters: base Setting value

Example:
```
#include <machine.h>
#define VBR 0x0000FC00
void main(void)
{
   set_vbr((void *)VBR);
}
```

void *get_vbr(void)

Description: Reads the vector base register (VBR).

Header: <machine.h> or <smachine.h>

Return value: Value of the vector base register

Example:
```
#include <machine.h>
void main(void)
{
   void *vbr;
   vbr = get_vbr();
}
```

void set_gbr(void *base)

Description: Sets **base** (32 bits) to the global base register (GBR).

Header: <machine.h> or <umachine.h>

Parameters: base Setting value

Example:
```
#include <machine.h>
#define IOBASE 0x05fffec0
void main(void)
{
   set_gbr((void *)IOBASE);
}
```

Remarks: As GBR is a control register whose contents are not guaranteed by all functions in this compiler, take care when changing GBR settings.

void *get_gbr(void)

Description: Reads the global base register (GBR).

Header: <machine.h> or <umachine.h>

Return value: Value of the vector base register

Example:
```
#include <machine.h>
void main(void)
{
   void *gbr;
   gbr = get_gbr();
}
```

unsigned char gbr_read_byte (int offset)

Description: Reads a byte (8 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Return value: Byte data (8 bits) reference value

Parameter: offset Offset address

Example:
```
#include <machine.h>
#define BDATA 0
void main(void)
{
   if(gbr_read_byte(BDATA) !=0)
      :
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.

unsigned short gbr_read_word (int offset)

Description: Reads a word (16 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Return value: Word data (16 bits) reference value

Parameter: offset Offset address

Example:
```
#include <machine.h>
#define WDATA 0
void main(void)
{
   if(gbr_read_word(WDATA) !=0)
      :
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +510 bytes.

unsigned long gbr_read_long (int offset)

Description: Reads a longword (32 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Return value: Long-word data (32 bits) reference value

Parameter: offset Offset address

Example:
```
#include <machine.h>
#define LDATA 0
void main(void)
{
   if(gbr_read_long(LDATA) !=0)
      :
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +1020 bytes.

void gbr_write_byte(int offset, unsigned char data)

Description: Sets a byte (8 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Parameter: offset Offset address
data Setting value (8 bits)

Example:
```
#include <machine.h>
#define BDATA 0
void main(void)
{
   gbr_write_byte(BDATA,0);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.

void gbr_write_word(int offset, unsigned short data)

Description: Sets a word (16 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
data	Setting value (16 bits)

Example:

```
#include <machine.h>
#define WDATA 0
void main(void)
{
   gbr_write_word(WDATA,0);
}
```

Remarks:
1. **offset** must be constants.
2. The specification range for offsets is +510 bytes.

void gbr_write_long(int offset, unsigned long data)

Description: Sets a longword (32 bits) at the address indicated by adding GBR and the offset specified.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
data	Setting value (32 bits)

Example:

```
#include <machine.h>
#define LDATA 0
void main(void)
{
   gbr_write_long(LDATA,0);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +1020 bytes.

void gbr_and_byte(int offset, unsigned char mask)

Description: ANDs a mask and a byte (8 bits) at the address indicated by adding GBR and the offset specified, and stores the result to the address indicated by adding GBR and the specified offset.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
mask	data (8 bits)

Example:

```
#include <machine.h>
#define BDATA 0
void main(void)
{
   gbr_and_byte(BDATA,0x01);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.
3. The specification range for mask is 0 to +255.

void gbr_or_byte(int offset, unsigned char mask)

Description: ORs a mask and a byte (8 bits) at the address indicated by adding GBR and the offset specified, and stores the result to the address indicated by adding GBR and the specified offset.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
mask	Data (8 bits)

Example:

```
#include <machine.h>
#define BDATA 0
void main(void)
{
   gbr_or_byte(BDATA,0x01);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.
3. The specification range for mask is 0 to +255.

void gbr_xor_byte(int offset, unsigned char mask)

Description: Exclusively ORs a mask and a byte (8 bits) at the address indicated by adding GBR and the offset specified, and stores the result to the address indicated by adding GBR and the specified offset.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
mask	Data (8 bits)

Example:

```
#include <machine.h>
#define BDATA 0
void main(void)
{
   gbr_xor_byte(BDATA,0x01);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.
3. The specification range for mask is 0 to +255.

int gbr_tst_byte(int offset, unsigned char mask)

Description: ANDs a mask and a byte (8 bits) at the address indicated by adding GBR and the offset specified, checks whether the result is 0 or not, and sets the T bit according to the result of the check.

Header: <machine.h> or <umachine.h>

Parameter:

offset	Offset address
mask	Data (8 bits)

Example:

```
#include <machine.h>
#define BDATA 0
void main(void)
{
   gbr_tst_byte(BDATA,0);
}
```

Remarks:
1. **offsets** must be constants.
2. The specification range for offsets is +255 bytes.
3. The specification range for mask is 0 to +255.

GBR Intrinsic Function Example:

```
#include <machine.h>
#define CDATA1 0
#define CDATA2 1
#define CDATA3 2
#define SDATA1 4
#define IDATA1 8
#define IDATA2 12

struct{
   char  cdata1;          /*  offset 0      */
   char  cdata2;          /*  offset 1      */
   char  cdata3;          /*  offset 2      */
   short sdata1;          /*  offset 4      */
   int   idata1;          /*  offset 8      */
   int   idata2;          /*  offset 12     */
}table;
void f();

void f()
{
   set_gbr( &table); /* Sets the start address of */
       :             /* table to GBR.             */
   gbr_write_byte( CDATA2, 10);
                     /* Sets 10 to table.cdata2.  */
   gbr_write_long( IDATA2, 100);
                     /* Sets 100 to table.idata2. */
       :
   if(gbr_read_byte( CDATA2) != 10)
                     /* Reads table.cdata2.       */
      gbr_and_byte( CDATA2, 10);
                    /* ANDs 10 and table.cdata2,   */
                    /* and sets it in table.cdata2.*/
   gbr_or_byte( CDATA2, 0x0F);
                    /* ORs 0x0F and table.cdata2,  */
         :          /* and sets it in table.cdata2.*/
   sleep();         /* Expanded to the sleep       */
                    /* instruction                 */

}
```

Effective Use of GBR Intrinsic Functions:

1. Allocate frequently accessed object to memory and set the start address of the object to GBR.

2. Byte data that frequently uses logical operations should be declared within 128 bytes of the start address of the structure.
   As a result, the load instruction of start address for accessing a structure can be reduced and load/store instructions necessary for performing logical operation can be reduced.

void sleep(void)

Description: Expanded to SLEEP instruction, which makes the CPU enter the low-power consumption mode .

Header: <machine.h> or <smachine.h>

Example:
```
#include <machine.h>
void main(void)
{
    sleep();
}
```

int tas(char *addr)

Description: Expanded to the **TAS.B @Rn** instruction.

Header: <machine.h> or <umachine.h>

Parameters: addr Address specified in the TAS instruction

Example:
```
#include <machine.h>
char a;
void main(void)
{
    tas(&a);
}
```

int trapa(int trap_no)

Description: Expanded to **TRAPA #trap_no**.

Header: <machine.h> or <umachine.h>

Parameters: trap_no Trap number

Example:
```
#include <machine.h>
void main(void)
{
   trapa(0);
}
```

int trapa_svc(int trap_no, type1 para1, type2 para2, type3 para3, type4 para4)

Description: Enables executing HI7000 and other OS system calls. When trapa_svc is executed, code is specified in R0, and para1 to para4 in R4 to R7, respectively.
Then, **TRAPA #trap_no** is executed.

Header: <machine.h> or <umachine.h>

Parameters:

trap_no	Trap number
code	Function code
para1 to para4	Parameters (0 to 4 variables) Types type1 to type4 are integer type or pointer type.

Example:
```
#include <machine.h>
#define SIG_SEM 0xffc8
void main(void)
{
   trapa_svc(63, SIG_SEM, 0x05);
}
```

void prefetch(void *p)

Description: An area indicated by the pointer (16-byte data from (int)p&0xfffffff0) is written to the cache memory.

Header: <machine.h> or <umachine.h>

Parameters: p Prefetch address

Example:
```
#include <machine.h>
char a[1200];
void main(void)
{
   int *pa = a;
   prefetch(pa);
}
```

Remarks: This function is valid only when the **cpu=sh3|sh3e|sh4** option is specified. This function does not affect the logical operation of the program.

void trace(long v)

Description: Expanded to the **TRACE** instruction.

Header: <machine.h> or <umachine.h>

Parameters: v Output variable

Example:
```
#include <machine.h>
void main(void)
{
   long v;
   trace(v);
}
```

Remarks: This function is valid only when the **cpu=sh4** option is specified.

int macw(short *ptr1,short*ptr2,unsigned int count)
int macwl(short *ptr1,short*ptr2,unsigned int count,unsigned int mask)

Description: Expanded to the multiply-and-accumulate instruction, **MAC.W** that multiplies and accumulates contents of two data tables.

Header: <machine.h> or <umachine.h>

Return value: Result of the MAC operation

Parameters:

ptr1	Start address of data to be multiplied or accumulated
ptr2	Start address of data to be multiplied or accumulated
count	Number of times the operation is performed
mask	Address mask that corresponds to the ring buffer

Example:

```
#include <machine.h>
short tbl1[]={a1,a2,a3,a4};
short tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
   result1=macw(tbl1,tbl2,3);
                     /* Executes a1*b1 + a2*b2    */
                     /* + a3*b3                   */
   result2=macwl(tbl1,tbl2,4,0xfffffffb);
                     /* Executes a1*b1 + a2*b2    */
                     /* + a3*b1 + a4*b2           */
}
```

Remarks: The multiply and accumulate operation intrinsic function does not check for parameters. Therefore, keep the following in mind:

a. Tables indicated by **ptr1** and **ptr2** must be aligned on the boundaries of multiples of 2 bytes.
b. The table indicated by **ptr2** in **macwl** must be aligned on the boundary of a multiple of (ring buffer **mask** × **2**).

int macl(int *ptr1,int*ptr2,unsigned int count)
int macll(int *ptr1,int*ptr2,unsigned int count,unsigned int mask)

Description: Expanded to the multiply-and-accumulate instruction, **MAC.L** that multiplies and accumulates contents of two data tables.

Header: <machine.h> or <umachine.h>

Return value: Result of the MAC operation

Parameters:

ptr1	Start address of data to be multiplied or accumulated
ptr2	Start address of data to be multiplied or accumulated
count	Number of times the operation is performed
mask	Address mask that corresponds to the ring buffer

Example:

```
#include <machine.h>
short tbl1[]={a1,a2,a3,a4};
short tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
   esult1=macl(tbl1,tbl2,3);
                      /* Executes a1*b1 + a2*b2    */
                      /* + a3*b3                   */
   result2=macll(tbl1,tbl2,4,0xfffffffb);
                      /* Executes a1*b1 + a2*b2    */
                      /* + a3*b1 + a4*b2           */
}
```

Remarks:

1. This function is valid only when the **cpu=sh2|sh2e|sh3|sh3e|sh4** option is specified.
2. The multiply and accumulate operation intrinsic function does not check parameters. Therefore, keep the following in mind:
   a. Tables indicated by **ptr1** and **ptr2** must be aligned on the boundaries of multiples of 4 bytes.
   b. The table indicated by **ptr2** in **macll** must be aligned on the boundary of a multiple of (ring buffer **mask × 2**).

void set_fpscr(int cr)

Description: Sets **cr** (32 bits) to the floating-point status control register FPSCR.

Header: <machine.h> or <umachine.h>

Parameters: cr Setting value (32 bits)

Example:
```
#include <machine.h>
void main(void)
{
   set_fpscr(0);
}
```

Remarks: This function is valid only when the **cpu=sh2e|sh3e|sh4** option is specified.

int get_fpscr()

Description: Refers to the floating-point status control register FPSCR.

Header: <machine.h> or <umachine.h>

Return value: FPSCR value

Example:
```
#include <machine.h>
int cr;
void main(void)
{
   cr = get_fpscr();
}
```

Remarks: This function is valid only when the **cpu=sh2e|sh3e|sh4** option is specified.

float fipr(float vect1[4], float vect2[4])

Description: Calculates inner product of two vectors.

Header: <machine.h> or <umachine.h>

Return value: Operation result

Parameters:

vect1	Vector
vect2	Vector

Example:

```
#include <machine.h>
extern float data1[4],data2[4];
float result;
void main(void)
{
   result=fipr(data1,data2);
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

float ftrv(float vec1[4], float vec2[4])

Description: Transforms vec1 (vector) by tbl (4x4 matrix), and stores the result to vec2 (vector). Note that tbl needs be loaded using intrinsic function ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

vec1	Vector
vec2	Vector

Example:

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4],data2[4];
void main(void)
{
  ld_ext(tbl);
  ftrv(data1,data2);
     /* As i=0,1,2,3 the result in data2 will be as */
     /* follows: data2[i]=data1[0]*tbl[0][i]+       */
     /* data1[1]*tbl[1][i] + data1[2]*tbl[2][i]      */
     /* data1[3]*tbl[3][i]                           */
}
```

Remarks:

1. This function is valid only when the **cpu = sh4** option is specified.
2. Intrinsic functions **ld_ext()** and **st_ext()** change the floating point register bank bit (FR) of the floating point status control register (FPSCR) to access the extension registers. Therefore, when using intrinsic functions **ld_ext()** or **st_ext()** in an interrupt function, change the interrupt mask before and after the vector operation intrinsic function as shown in the following example.

Example

```
extern float mat1[4][4];
extern float vec1[4],vec2[4];
#pragma interrupt (intfunc)
void intfunc(){
  :
   ld_ext();
  :
}
void normfunc(){
       :
   int maskdata=get_imask();
   set_imask(15);
   ld_ext(mat1);
   ftrv(vec1,vec2);
   set_imask(maskdata);
       :
}
```

void ftrvadd(float vec1[4], float vec2[4], float vec3[4])

Description: Transforms vec1 (vector) by tbl (4x4 matrix), adds the result to vec2 (vector), then stores the sum to vec3 (vector). Note that tbl need be loaded using intrinsic function ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

vec1	Vector
vec2	Vector
vec3	Vector

Example:

```
#include <machine.h>
extern float tb1[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
  ld_ext(tbl);
  ftrvadd(data1,data2,data3);
     /* data3 = data1 x tbl + data2                    */
     /* As i=0,1,2,3 the result in data3 will be as  */
     /* follows: data3[i]=data1[0]*tbl[0][i]         */
     /*                  +data1[1]*tbl[1][i]          */
     /*                  +data1[2]*tbl[2][i]          */
     /*                  +data1[3]*tbl[3][i]          */
     /*                  +data2[i]                    */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

void ftrvsub(float vec1[4], float vec2[4], float vec3[4])

Description: Transforms vec1 (vector) by tbl (4x4 matrix), subtracts vec2 (vector) from the result, then stores the difference to vec3 (vector). Note that tbl needs be loaded using intrinsic function ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

vec1	Vector
vec2	Vector
vec3	Vector

Example:

```
#include <machine.h>
extern float tb1[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
  ld_ext(tbl);
  ftrvsub(data1,data2,data3);
     /* data3 = data1 x tbl - data2                    */
     /* As i=0,1,2,3 the result in data3 will be as  */
     /* follows: data3[i]=data1[0]*tbl[0][i]          */
     /*                  +data1[1]*tbl[1][i]          */
     /*                  +data1[2]*tbl[2][i]          */
     /*                  +data1[3]*tbl[3][i]          */
     /*                  -data2[i]                    */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

void add4(float vec1[4], float vec2[4], float vec3[4])

Description: Stores the sum of vec1 (vector) and vec2 (vector) to vec3 (vector).

Header: <machine.h> or <umachine.h>

Parameters:

vec1	Vector
vec2	Vector
vec3	Vector

Example:

```
#include <machine.h>
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
  add4(data1,data2,data3);   /* data3 = data1 + data2 */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

void sub4(float vec1[4], float vec2[4], float vec3[4])

Description: Stores the difference between vec1 (vector) and vec2 (vector) to vec3 (vector).

Header: <machine.h> or <umachine.h>

Parameters:

vec1	Vector
vec2	Vector
vec3	Vector

Example:

```
#include <machine.h>
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
  sub4(data1,data2,data3);   /* data3 = data1 - data2 */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

void mtrx4mul(float mat1[4], float mat2[4])

Description: Transforms mat1 (4x4 matrix) by tbl (4x4 matrix), and stores the result to mat2. Note that tbl needs be loaded using intrinsic instruction ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

mat1	4x4 matrix
mat2	4x4 matrix

Example:

```
#include <machine.h>
extern float tb1[4][4];
extern float tb11[4][4];
extern float tb12[4][4];
void main(void)
{
  ld_ext(tbl);
  mtrx4mul(tb1l,tbl2);   /* tbl2 = tbl1 x tbl */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

This function is 4x4 matrix operation and therefore is not commutative.

Example

```
extern float matA[][];
extern float matB[][];
int judge(){
   float data1[4][4], data2[4][4];
   set_imask(15);
   ld_ext(matA);
   mtrx4mul(matB,data1);/* data1=matB × matA */
   ld_ext(matB);
   mtrx4mul(matA,data2);/* data2=matA × matB */
   ..../* elements of data1[][] and data2[][] do */
       /* not necessarily match.                */
}
```

void mtrx4muladd(float mat1[4], float mat2[4], float mat3[4])

Description: Transforms mat1 (4x4 matrix) by tbl (4x4 matrix), adds the result of mat2 (4x4 matrix), and stores the sum to mat3 (4x4 matrix).
Note that tbl needs be loaded using intrinsic instruction ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

mat1	4x4 matrix
mat2	4x4 matrix
mat3	4x4 matrix

Example:

```
#include <machine.h>
extern float tb1[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
  ld_ext(tbl);
  mtrx4muladd(tb1l,tbl2, tbl3);
                              /* tbl2 = tbl1 x tbl +tbl2 */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

This function is 4x4 matrix operation and therefore is not commutative.

void mtrx4mulsub(float mat1[4], float mat2[4], float mat3[4])

Description: Transforms mat1 (4x4 matrix) by tbl (4x4 matrix), subtracts mat2 (4x4 matrix) from the result, and stores the difference to mat3 (4x4 matrix).
Note that tbl needs be loaded using intrinsic instruction ld_ext().

Header: <machine.h> or <umachine.h>

Parameters:

mat1	4x4 matrix
mat2	4x4 matrix
mat3	4x4 matrix

Example:

```
#include <machine.h>
extern float tb1[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
  ld_ext(tbl);
  mtrx4mulsub(tb1l,tbl2, tbl3);
                            /* tbl2 = tbl1 x tbl - tbl2 */
}
```

Remarks: This function is valid only when the **cpu = sh4** option is specified.

This function is 4x4 matrix operation and therefore is not commutative.

void ld_ext(float mat1[4] [4])

Description: Loads mat (4x4 matrix) to extension register.

Header: <machine.h> or <umachine.h>

Parameters: mat 4x4 matrix

Example:

```
#include <machine.h>
extern float tb1[4][4];
void main(void)
{
  ld_ext(tbl);
}
```

Remarks:

1. This function is valid only when the **cpu = sh4** option is specified.
2. Intrinsic functions **ld_ext()** changes the floating point register bank bit (FR) of the floating point status control register (FPSCR) to access extension register. Therefore, when this function is used in an interrupt function, change the interrupt mask before and after the vector operation intrinsic function.

void st_ext(float mat1[4] [4])

Description: Stores contents of extension register to mat (4x4 matrix).

Header: <machine.h> or <umachine.h>

Parameters: mat 4x4 matrix

Example:

```
#include <machine.h>
extern float tb1[4][4];
void main(void)
{
  st_ext(tbl);
}
```

Remarks:

1. This function is valid only when the **cpu = sh4** option is specified.
2. Intrinsic functions **st_ext()** changes the floating point register bank bit (FR) of the floating point status control register (FPSCR) to access extension register. Therefore, when this function is used in an interrupt function, change the interrupt mask before and after the vector operation intrinsic function.

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.

(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 10.27 shows the various library types and the corresponding standard include files.

Table 10.27 Library Types and Corresponding Standard Include Files

Library Type	Description	Standard Include Files
Program diagnostics	Outputs program diagnostic information.	<assert.h>
Character handling	Handles and checks characters.	<ctype.h>
Mathematics	Performs numerical calculations such as trigonometric functions.	<math.h> <mathf.h>
Non-local jumps	Supports transfer of control between functions.	<setjmp.h>
Variable arguments	Supports access to variable arguments for functions with such arguments.	<stdarg.h>
Input/output	Performs input/output handling.	<stdio.h>
General utilities	Performs C program standard processing such as storage area management.	<stdlib.h>
String handling	Performs string comparison, copying, etc.	<string.h>

In addition to the above standard include files, standard include files consisting solely of macro name definitions, shown in table 10.28, are provided to improve programming efficiency.

Table 10.28 Standard Include Files Comprising Macro Name Definitions

Standard Include File	Description
<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 Functional overview

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.
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, which handles a single character, drove the input/output device and the OS functions. To solve this problem, a storage area called a buffer is normally provided, and the data in the buffer is input or output at one time.

From the viewpoint of the program, on the other hand, it is more convenient to call input/output functions for each character.

Using the library functions, character-by-character input/output can be performed efficiently without awareness of the buffer status within the program by automatically performing buffer management.

Those library functions enable a programmer to write a program considering the input/output as a single data stream, making the programmer be able to implement data input/output efficiently without being aware of the detailed procedure. Such capability is called stream input/output.

(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 expansion cannot be invalidated even if the user declares a function with the same name.

(ii) If an expression with a side effect as a macro parameter (assignment expression, increment, decrement) is specified, its result will not be 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**, that 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 character

The end of a string literal in 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 system. With text files, characters input or output by a library function do not necessarily correspond to a physical stream of data in the file.

(ii) Binary files

A binary file is configured as a row of byte data. Data input or output by a library function corresponds to a physical list of data in the file.

(i) Standard input/output files

Files that can be used as standard by input/output library functions by default without preparations such as opening file are called standard input/output files. Standard input/output files comprise the standard input file (stdin), standard output file (stdout), and standard error output file (stderr).

(i) Standard input file (stdin)

Standard file to be input to a program.

(ii) Standard output file (stdout)

Standard file to be output from a program.

(iii) Standard error output file (stderr)

Standard file for storing 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 source program, floating-point numbers are represented by decimal numbers, but inside the computer they are normally represented by binary numbers.

In the case of binary numbers, the floating-point representation is as follows:

$2^n \times m$ (n: integer, m: binary fraction)

Here, n is called the exponent of the floating-point number, and m is called the mantissa. The 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 (i.e., 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.

$2^5 \times 1.0_{(2)}$ ($_{(2)}$ indicates a binary number)

$2^6 \times 0.1_{(2)}$

Usually, a representation in which the leading digit is not 0 is used, in order to secure the number of valid digits. This is called a normalized floating-point number, and the operation that converts a floating-point number to this kind of representation is called normalization.

(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 for rounding to be carried out. However, this alone does not permit an accurate result to be achieved in the event of digit dropping, etc. For this reason, the intermediate result is saved with an extra bit, called a guard bit.

(k) File access mode

This is 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 10.29.

Table 10.29 File Access Modes

Access Mode	Meaning
'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 by 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 for each compiler.

When a library is used, the behavior is undefined if the contents of these macros are redefined.

(b) With libraries, errors are not detected in all cases. The behavior is undefined 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 an operation using the sizeof operator.
Constant (macro)	NULL	Indicates the value when a pointer is not pointing at anything. This value is such that the result of a comparison with 0 using the equality operator (==) is true.
Variable (macro)	errno	If an error occurs during library function processing, the error code defined in the respective library is set in errno. By setting 0 in errno before calling a library function and checking the error code set in errno after the library function processing has ended, it is possible to check whether an error occurred during the library function processing.

Implementation Define

Item	Compiler Specifications
Value of macro NULL	The pointer type value 0 is set to void.
Contents of macro ptrdiff_t	int type

<assert.h>

Adds diagnostics into programs.

Type	Definition Name	Description
Function (macro)	assert	Adds diagnostics into programs.

To invalidate the diagnostics defined by <assert.h>, define macro name NDEBUG with a **#define** statement (**#define NDEBUG**) before including <assert.h>.

Note: If a **#undef** statement is used for macro name assert, the result of subsequent assert calls will not be 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 the expression is true, the assert macro terminates processing without returning a value. If expression is false, it outputs diagnostic information to the standard error file in the form defined by the compiler, and then calls the abort function.

The diagnostic information includes the parameter's program text, source file name, and source line numbers.

Implementation define:

The following message is output when the expression is false in 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.
	iscntrl	Tests for a control character.
	isdigit	Tests for a decimal digit.
	isgraph	Tests for a printing character except space.
	islower	Tests for a lowercase letter.
	isprint	Tests for a printing character, including space.
	ispunct	Tests for a special character.
	isspace	Tests for a 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 undefined. Character types are listed in table 10.30.

Table 10.30 Character Types

Character Type	Description
Uppercase letter	Any of the following 26 characters ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, ‘Z’
Lowercase letter	Any of the following 26 characters ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’, ‘l’, ‘m’, ‘n’, ‘o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’
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

Item	Compiler Specifications
The character set inspected by the isalnum function, isalpha function, iscntrl function, islower function, isprint function, and isupper functions	Character set represented by the unsigned char type. Table 10.31 shows the character set that results in a true return value.

Table 10.31 True Character

Function Name	True Characters
isalnum	'0' to '9', 'A' to 'Z', 'a' to 'z'
isalpha	'A' to 'Z', 'a' to 'z'
iscntrl	'\x00' to '\x1f', '\x7f'
islower	'a' to 'z'
isprint	'\x20' to '\x7E'
isupper	'A' to 'Z'

int isalnum(int c)

Description: Tests for an alphabetic character or a 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 followings 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.
	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.

Type	Definition Name	Definition Value	Description
Constant (macro)	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.
	FLT_POS_EPS	5.9604648328104311e–8F	Indicates the minimum floating-point value x for which $1.0 + x \neq 1.0$ in float type.

Type	Definition Name	Definition Value	Description
Constant (macro)	DBL_POS_EPS	1.1102230246251567e–16	Indicates the minimum floating-point value x for which 1.0 + x ≠ 1.0 in double type.
	LDBL_POS_EPS	1.1102230246251567e–16	Indicates the minimum floating-point value x for which 1.0 + x ≠ 1.0 in long double type.
	FLT_NEG_EPS	2.9802324164052156e–8F	Indicates the minimum floating-point value x for which 1.0 – x ≠ 1.0 in float type.
	DBL_NEG_EPS	5.5511151231257834e–17	Indicates the minimum floating-point value x for which 1.0 – x ≠ 1.0 in double type
	LDBL_NEG_EPS	5.5511151231257834e–17	Indicates the minimum floating-point value x for which 1.0 – x ≠ 1.0 in long double type.
	FLT_POS_EPS_EXP	–23	Indicates the minimum integer n for which 1.0 + $(radix)^n \neq 1.0$ in float type.
	DBL_POS_EPS_EXP	–52	Indicates the minimum integer n for which 1.0 +$(radix)^n \neq 1.0$ in double type.
	LDBL_POS_EPS_EXP	–52	Indicates the minimum integer n for which 1.0 + $(radix)^n \neq 1.0$ in long double type.
	FLT_NEG_EPS_EXP	–24	Indicates the minimum integer n for which 1.0 – $(radix)^n \neq 1.0$ in float type.
	DBL_NEG_EPS_EXP	–53	Indicates the minimum integer n for which 1.0 – $(radix)^n \neq 1.0$ in double type.
	LDBL_NEG_EPS_EXP	–53	Indicates the minimum integer n for which 1.0 – $(radix)^n \neq 1.0$ in long double type.

<limits.h>

Defines various limits relating to the internal representation of integer type data.
The followings 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 int type variable can have as a value.
	INT_MAX	217483647	Indicates the maximum value that an int type variable can have as a value.
	INT_MIN	–2147483647–1	Indicates the minimum value that an int type variable can have as a value.
	UINT_MAX	4294967295U	Indicates the maximum value that an unsigned int type variable can have as a value.
	LONG_MAX	217483647L	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 followings 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, Standard 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 followings 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	Computes the smallest integral value not less than or equal to the given floating-point number.
	fabs	Computes the absolute value of a floating-point number.
	floor	Computes the largest integral value not greater than or equal to the given floating-point number.
	fmod	Computes the floating-point 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 compiler.

(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.

3. In the following cases, errno will not be set by the fabs and sqrt function even though an error has occrred in the function.
   (1) cpu=sh3e and double=float options are specified.
   (2) cpu=sh4 option is specified.
   (3) cpu=sh2e and double=float options are specified (only fabs function).

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-numbers, refer to section 10.1.3, Floating-Point Number Specifications.
Whether errno is set to the value of macro ERANGE if an underflow error occurs in a mathematical function	Not specified
Whether a range error occurs if the second argument in the fmod function is 0	A range error occurs.

double acos(double d)

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, π] 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 [–π/2, +π/2] by the radian.

double atan(double d)

Description: Computes the arc tangent of a floating-point number.

Header file: <math.h>

Return values: Normal: Arc tangent of **d**

Abnormal: —

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 the 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 π. If x = 0.0, the result is $\pm\pi/2$, depending on whether **y** is positive or negative.

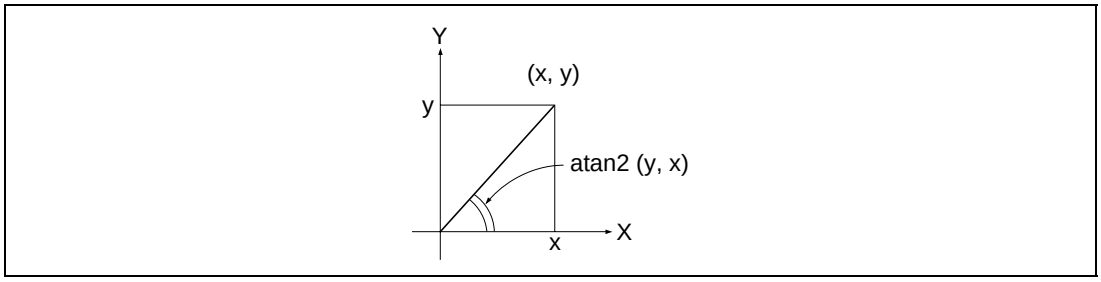


Figure 10.5 Meaning of atan2 Function

double cos(double d)

Description: Computes the cosine of a floating-point radian value.

Header file: <math.h>

Return values: Normal: Cosine of **d**

Abnormal: —

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: Normal: Sine of **d**

Abnormal: —

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: Normal: Tangent of **d**

Abnormal: —

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: Normal: Hyperbolic cosine of **d**

Abnormal: —

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: Normal: Hyperbolic sine of **d**

Abnormal: —

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: Normal: Hyperbolic tangent of **d**

Abnormal: —

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: Normal: Exponential value of **d**

Abnormal: —

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 int *e)

Description: Breaks a floating-point number into a [0.5, 1.0) value and a power of 2.

Header file: <math.h>

Return values: Normal: If value is 0.0: 0.0
If value is not 0.0: Value of **ret** defined by
ret * $2^{\text{value pointed to by e}}$ = value

Abnormal: —

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 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: Normal: Result of e * 2^f operation

Abnormal: —

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: Normal: Fractional part of a

Abnormal: —

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 or equal to the given floating-point number.

Header file: <math.h>

Return values: Normal: Smallest integral value not less than or eaual to **d**

Abnormal: —

Parameters: d Floating-point number for which smallest integral value not less than that number is to be computed

Example:
```
#include <math.h>
double d, ret;
    ret=ceil(d);
```

Remarks: The **ceil** function returns the smallest integral value not less than or equal to **d**, expressed as a double. 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: Normal: Absolute value of **d**

Abnormal: —

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 or equal to the given floating-point number.	
Header file:	<math.h>	
Return values:	Normal:	Largest integral value not greater than or equal to **d**
	Abnormal:	—
Parameters:	d	Floating-point number for which largest integral value not greater than that number is to be computed

Example:

```
#include <math.h>
double d, ret;
    ret=floor(d);
```

Remarks: The **floor** function returns the largest integral value not greater than or equal to **d**, expressed as a double. 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 floating-point remainder of division of two floating-point numbers.

Header file: <math.h>

Return values:
- Normal: When **y** is 0.0: x
  When **y** is not 0.0: Remainder of division of **x** by **y**
- Abnormal: —

Parameters:

x	Dividend
y	Divisor

Example:

```
#include <math.h>
double x, y, ret;
    ret=fmod(x, y);
```

Remarks: In the **fmod** function, the relationship between parameters **x** and **y** and return value **ret** is as follows:

x = y * I + ret (where I is an integer)

The sign of return value **ret** is the same as the sign of **x**.

If the quotient of x/y cannot be expressed, the value of the result will not be guaranteed.

<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 are does 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 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	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 a floating-point number.
	sqrtf	Computes the positive square root of a floating-point number.
	ceilf	Computes the smallest integral value not less than or equal to the given floating-point number.

Type	Definition Name	Description
Function	fabsf	Computes the absolute value of a floating-point number.
	floorf	Computes the largest integral value not greater than or equal to the given floating-point number.
	fmodf	Computes the floating-point 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 compiler.

2. Range error

   A range error occurs if the result of a function cannot be represented as a float type value. In this case, the value of ERANGE is set in **errno**. If the result overflows, the function returns the value of HUGE_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.

3. In the following cases, errno will not be set by the fabs and sqrt function even though an error has occrred in the function.
   (1) cpu=sh3e option is specified.
   (2) cpu=sh4 option is specified.
   (3) cpu=sh2e option is specified (only fabsf function).

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-numbers, refer to section 10.1.3, Floating-Point Number Specifications.
Whether errno is set to the value of macro ERANGE if an underflow error occurs in a mathematical function	Not specified
Whether a range error occurs if the second argument in the fmod function is 0	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, π] 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 [–π/2, +π/2] by the radian.

float atanf(float f)

Description: Computes the arc tangent of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Arc tangent of **f**

Abnormal: —

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 (–π, +π] 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 π. If x = 0.0, the result is ±π/2, depending on whether **y** is positive or negative.

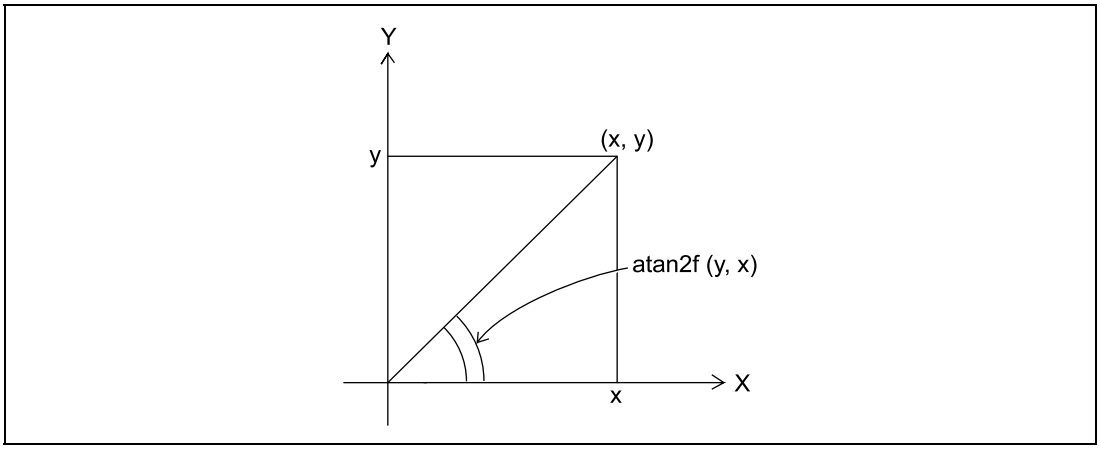


Figure 10.6 Meaning of atan2f Function

float cosf(float f)

Description: Computes the cosine of a floating-point radian value.

Header file: <mathf.h>

Return values: Normal: Cosine of **f**

Abnormal: —

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: Normal: Sine of **f**

Abnormal: —

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: Normal: Tangent of **f**

Abnormal: —

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: Normal: Hyperbolic cosine of **f**

Abnormal: —

Parameters: f Floating-point number for which hyperbolic cosine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=coshf(f);
```

float sinhf(float f)

Description: Computes the hyperbolic sine of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Hyperbolic sine of **f**

Abnormal: —

Parameters: f Floating-point number for which hyperbolic sine is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=sinhf(f);
```

float tanhf(float f)

Description: Computes the hyperbolic tangent of a floating-point number.

Header file: <mathf.h>

Return values: Normal: Hyperbolic tangent of **f**

Abnormal: —

Parameters: f Floating-point number for which hyperbolic tangent is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=tanhf(f);
```

float expf(float f)

Description: Computes the exponential function of a floating-point number.

Header file: <mathf.h>

Return values:
- Normal: Exponential value of **f**
- Abnormal: —

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 int *e)

Description: Breaks a floating-point number into a [0.5, 1.0] value and a power of 2.

Header file: <mathf.h>

Return values:
- Normal: If value is 0.0: 0.0
  If value is not 0.0: Value of **ret** defined by
  ret * $2^{\text{value pointed to by e}}$ = value
- Abnormal: —

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 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.

float ldexpf (float ret, int f)

Description: Multiplies a floating-point number by a power of 2.

Header file: <mathf.h>

Return values: Normal: Result of e * 2^f operation

Abnormal: —

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=idexpf(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: Normal: Fractional part of a

Abnormal: —

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 or equal to the given floating-point number.

Header file: <mathf.h>

Return values: Normal: Smallest integral value not less than or equal to **f**

Abnormal: —

Parameters: f Floating-point number for which smallest integral value not less than that number is to be computed

Example:
```
#include <mathf.h>
float f, ret;
    ret=ceilf(f);
```

Remarks: The **ceilf** function returns the smallest integral value not less than or equal to **f**, expressed as a float. 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: Normal: Absolute value of **f**

Abnormal: —

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 or equal to the given floating-point number.

Header file: <mathf.h>

Return values: Normal: Largest integral value not greater than or equal to **f**

Abnormal: —

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 **floor** function returns the largest integral value not greater than or equal to **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 floating-point remainder of division of two floating-point numbers.

Header file: <mathf.h>

Return values:
Normal: When **y** is 0.0: x
When **y** is not 0.0: Remainder of division of **x** by **y**
Abnormal: —

Parameters:

x	Dividend
y	Divisor

Example:

```
#include <mathf.h>
float x, y, ret;
    ret=fmodf(x, y);
```

Remarks: In the **fmodf** function, the relationship between parameters **x** and **y** and return value **ret** is as follows:

x = y * i + ret (where i is an integer)

The sign of return value **ret** is the same as the sign of **x**.

If the quotient of x/y cannot be expressed, the value of the result will not be guaranteed.

<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 executing environment defined by jmp_buf of the currently executing function in the specified storage area.
	longjmp	Restores the function executing 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 executing 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 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 executing environment of the currently executing function in the specified storage area.

Header file: <setjmp.h>

Return values:
- Normal: When **setjmp** function is called: 0
  On return from **longjmp** function: Nonzero
- Abnormal: —

Parameters:
- env: Pointer to storage area in which executing environment is to be saved

Example:

```
#include <setjmp.h>
int ret;
jmp_buf env;
    ret=setjmp(env);
```

Remarks: The executing 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 executing environment, such as the intermediate result of expression evaluation, may be lost. The **setjmp** function should only 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 executing environment saved by the **setjmp** function, and transfers control to the program location at which the **setjmp** function was called.

Header file: <setjmp.h>

Parameters:

env	Pointer to storage area in which executing environment was saved
ret	Return code to **setjmp** 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 executing 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 will not be 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 types of variables used in common by the va_start, va_arg, and va_end macros in order to reference variable parameters.
Function (macro)	va_start	Executes initialization processing for performing variable parameter referencing.
	va_arg	Enables referencing of the argument following the argument currently being referenced for a function with variable parameters.
	va_end	Terminates referencing of the arguments of a function with variable parameters.

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 parameter, and function prlist is implemented, outputting that number of subsequent parameters.

In line 18, the variable parameter reference is initialized by **va_start**. Each time an parameter is output, the next parameter is referenced by the **va_arg** macro (line 20). In the **va_arg** macro, the type name of the parameter (in this case, int type) is specified in the second parameter.

When parameter referencing ends, the **va_end** macro is called (line 22).

void va_start(va_list, 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>
va_list(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 a function with no variable name, the **va_start** macro call must be executed first of all.

type va_arg(va_list ap, type)

Description: Enables referencing of the parameter following the parameter currently being referenced for a function with variable parameters.

Header file: <stdarg.h>

Return values: Normal: Parameter value

Abnormal: —

Parameters:

ap	Variable for accessing variable parameters
type	Type of parameter to be accessed

Example:

```
#include <stdarg.h>
va_list ap;
type ret;
    ret=va_arg(ap, type);
```

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 parameter is specified as the type of **type**. If this kind of type is specified, operation will not be guaranteed.

void va_end(va_list ap)

Description: Terminates referencing of the parameters of a function with variable arguments.

Header file: <stdarg.h>

Parameters: ap Variable for accessing variable parameters

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 will not be 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	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	Indicates the number of files for which simultaneous opening is guaranteed by the implementation.
	TMP_MAX	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.

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 (stdout) according to a format.
	vsprintf	Outputs a variable parameter list to the specified area according to a format.
	fgetc	Inputs one character from a stream input/output file.
	fgets	Inputs a string from a stream input/output file.
	fputc	Outputs one character to a stream input/output file.
	fputs	Outputs a string to a stream input/output file.
	getc	(macro) Inputs one character from a stream input/output file.
	getchar	(macro) Inputs one character from the standard input file.
	gets	Inputs a string from the standard input file.
	putc	(macro) Outputs one character to a stream input/output file.
	putchar	(macro) Outputs one character to the standard output file.
	puts	Outputs a string to the standard output file.
	ungetc	Returns one character to a stream input/output file.
	fread	Inputs data from a stream input/output file to the specified storage area.
	fwrite	Outputs data from a storage area to a stream input/output file.
	fseek	Shifts the current read/write position in a stream input/output file.

Type	Definition Name	Description
Function	ftell	Computes 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).

Specification Defined by the Implementation

Item	Compiler Specifications
Whether the last line of the input text requires a line feed character indicating end	Not specified. Depends on the low-level interface routine specifications.
Whether the blank characters written immediately before the carriage return character are 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	
Whether a file with file length 0 exists	
File name configuration rule	
Whether the same file is 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 previous character and following characters.
Value of errno specified by the fgetpos or ftell function	The fgetpos function is not supported. The ftell function does not specify the errno value. The errno value depends on the low-level interface routine specifications.
Output format of messages generated by the perror function	See (a) below for the output message format.
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
<string literal>:<error message for the error number specified in error>

(b) Table 10.32 shows the format when displaying the infinite of floating points and not-a-numbers in **printf** and **fprintf** functions.

Table 10.32 Display Format of Infinite and Not-a-Numbers

Value	Display Format
Infinite of positive number	+++++
Infinite of negative number	-----
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 is returned; these are set in variables **ifp** and **ofp**.

After successful opening, input/output is performed using these FILE type data.

When file processing ends, the files are closed with the **fclose** function.

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 when 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 as **type**
Input/output is fully buffered.

(2) When _IOLBF is specified as **type**
Input/output is line buffered. That is, input/output data is fetched from the buffer area when a new-line character is written, when the buffer area is full, or when input is requested.

(3) When _IONBF is specified as **type**
Input/output is unbuffered.
The **setvbuf** function usually returns 0. However, when an illegal value is specified for type or size, or when the request on how to use the buffer could not be accepted, a value other than 0 is returned.

The buffer area must not be released before the 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;
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 characters converted and output when the function is terminated successfully, or a negative value if an error occurs.

The format specifications are shown below.

(1) Overview of formats

The string literal 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 …] { [*] / [Field width] } [. { [*] / [Precision] }] [Parameter size specification] Conversion string

When there is no parameter to be actually output for this conversion specification, the behavior will not be guaranteed. Also, when the number of parameters to be actually output is greater than the conversion specification, 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 flag that can be specified, and their meanings, are shown in table 10.33.

Table 10.33 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 10.35. (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 suffixed to the data.)

If the number of converted data characters exceeds the field width, the field width is extended to allow the converted result to be output.

If the field width specification begins with 0, 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 10.35.

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.

(1) For d, i, o, u, x, and X conversions
The minimum number of digits in the converted data is specified.

(2) For e, E, and f conversions
The number of digits after the decimal point in the converted data is specified.

(3) For g and G conversions
The maximum number of significant digits in the converted data is specified.

(4) 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 10.35), 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 10.34 shows the types of size specification and their meanings.

Table 10.34 Parameter Size Specification Types and Meanings

Type	Meaning
h	For d, i, o, u, x, and X conversions, specifies that the data to be converted is of short type or unsigned short type.
l	For d, i, o, u, x, and X conversions, specifies that the data to be converted is of long type, unsigned long type, or double type.
L	For e, E, f, g, and G conversions, specifies that the data to be converted is of long double type.

(e) Conversion character

Specifies the format into which the 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 will not be guaranteed except when a character array is converted by s conversion or when a pointer is converted by p conversion. Table 10.35 shows the conversion character and conversion methods. If a letter which is not shown in this table is specified as the conversion character, the behavior will not be guaranteed. The behavior, if the other character is specified, depends on the compiler.

Table 10.35 Conversion Characters and Conversion Methods

Conversion Character	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 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	
g	g conversion (or G conversion)	Whether f conversion format output or e conversion (or E conversion) format output is performed is determined by the value to be converted and the precision value that specifies the number of significant digits. Then double type data is output. If the exponent of the converted data is less than –4, or larger than the precision that indicates the number of significant digits, conversion to e (or E) format is performed.	double type	The precision specification indicates the maximum number of significant digits in the converted data.
G			double type	

Table 10.35 Conversion Characters and Conversion Methods (cont)

Conversion Character	Conversion Type	Conversion Method	Data Type Subject to Conversion	Notes on Precision
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	Assuming data as a pointer, conversion is performed to a string of compiler-defined printable characters.	Pointer to void type	The precision specification is invalid.
n	no conversion is performed.	Data is regarded as 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:	Input data ends before input data conversion is performed: EOF

Parameters:

fp	File pointer
control	Pointer to string indicating format
ptr,...	Pointer to storage area that stores input data

Example:

```
#include <stdio.h>
FILE *fp;
const char *control=”%d”;
int ret,buffer[10];
    ret=fscanf(fp, control, buffer);
```

Remarks: The **fscanf** function inputs data from the stream input/output file indicated by file pointer **fp**, converts and edits it according to the string 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) 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 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 stores input data for the conversion specification in the format, the behavior will not be guaranteed. Also, when a pointer to a storage area that stores input data remains though the format is exhausted, that pointer is ignored.

(2) Description of conversion specification

(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 10.37), specifies the size (short type, long type, or long double type) of the converted data. In other conversions, this specification is ignored. Table 10.36 shows the types of size specification and their meanings.

Table 10.36 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 character

The input data is converted according to the type of conversion specified by the conversion character. However, processing is terminated when a white-space character is read, when a character for which conversion is not permitted is read, or when the specified field width has been exceeded.

Table 10.37 Conversion Specifiers and Conversion Methods

Conversion Specifier	Conversion Type	Conversion Method	Data Type Subject to Conversion
d	d conversion	A decimal string is converted to integer type data.	Integer type
i	i conversion	A decimal string with a sign prefixed, or a decimal string with u (U) or l (L) suffixed is converted to integer type data. A string beginning with 0x (or 0X) is interpreted as hexadecimal, and the string is converted to int type data. A string beginning with 0 is interpreted as octal, and the string is converted to int type data.	Integer type
o	o conversion	An octal string is converted to integer type data.	Integer type
u	u conversion	An unsigned decimal string is converted to integer type data.	Integer type
x	x conversion	A hexadecimal string is converted to integer type data. There is no difference in meaning between x conversion and X conversion.	Integer type
X	X conversion		
s	s conversion	Characters are converted as a single string until a space, horizontal tab, or new-line character is read. A null character is appended at the end of the string. (The string in which the converted data is set must be large enough to include the null character.)	Character type
c	c conversion	One character is input. The input character is not skipped even if it is a white-space character. To read only non-white-space characters, specify %1s. If the field width is specified, the number of characters equivalent to that specification are read. In this case, therefore, the storage area that stores the converted data needs the specified size.	char type
e	e conversion	A string indicating a floating-point number is converted to floating-point type data. There is no difference in meaning between the e conversion and E conversion, or between the g conversion and G conversion. The input format is a floating-point number that can be represented by the strtod function.	Floating-point type
E	E conversion		
f	f conversion		
g	g conversion		
G	G conversion		
p	p conversion	A string converted by p conversion of the fprintf function is converted to pointer type data.	Pointer to void type
n	no conversion is performed.	Data input is not performed; the number of data characters input so far is set.	Integer type
[	[conversion	A 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 10.37, the behavior will not be guaranteed. For the 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>
const char *control;
int ret;
char buffer[]="Hello World\n";
    ret=printf(control, buffer);
```

Remarks: The **printf** function converts and edits parameter arg according to the string that 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:

Normal:	Number of characters converted
Abnormal:	—

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;
int ret;
char buffer[]="Hello World\n";
    ret=sprintf(s, control, buffer);
```

Remarks: The **sprintf** function converts and edits parameter arg according to the string that 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 when 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 when an error occurs.

Within the **vfprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter arg, indicating the 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.

Within the **vprintf** function, the **va_end** macro is not invoked.

For details of the format specifications, see the description of the **fprintf** function.

Parameter arg, indicating the 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:

```
#include <stdarg.h>
#include <stdio.h>
char *s;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vsprintf(s,control,buffer);
   va_end(ap);
}
```

Remarks: The **vsprintf** 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 storage area pointed to by **s**.

A null character is appended at the end of the converted and output string. This null character is not included in the return value (number of characters output).

For details of the format specifications, see the description of the **fprintf** function.

Parameter arg, indicating the argument list, must be initialized beforehand by the **va_start** and **va_arg** macros.

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:

When a read error occurs, the error indicator for that file is set.

Remarks: The **fgetc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **fgetc** function normally returns the input character, but returns EOF at end-of-file or when an error occurs. At end-of-file, the end-of-file indicator for that file is set.

char *fgets(char *s, 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 a null pointer 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 will not be guaranteed when 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:

When a write error occurs, the error indicator for that file is set.

Remarks: The **fputc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.

The **fputc** function normally returns **c**, the output character, but returns EOF when an error occurs.

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 when 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:

When a read error occurs, the error indicator for that file is set.

Remarks: The **getc** function inputs one character from the stream input/output file indicated by file pointer **fp**.

The **getc** function normally returns the input character, but returns EOF at end-of-file or 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:

When a read error occurs, the error indicator for that file is set.

Remarks: The **getchar** function inputs one character from the standard input file (stdin).

The **getchar** function normally returns the input character, but returns EOF at end-of-file or 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 a null pointer at the end of the standard input file or when an error occurs.

The contents of the storage area pointed to by **s** do not change at the end of the standard input file, but will not be guaranteed when 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:

When a write error occurs, the error indicator for that file is set.

Remarks: The **putc** function outputs character **c** to the stream input/output file indicated by file pointer **fp**.

The **putc** function normally returns **c**, the output character, but returns EOF when an error occurs.

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:
When a write error occurs, the error indicator for that file is set.

Remarks: The **putchar** function outputs character **c** to the standard output file (stdout).

The **putchar** function normally returns **c**, the output character, but returns EOF when an error occurs.

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 c, ret;
    ret=puts(s);
```

Remarks: The **puts** function outputs the string pointed to by **s** to the standard output file (stdout). The null character indicating the end of the string is not output, but a new-line character is output instead.

The **puts** function normally returns zero, but returns nonzero when an error occurs.

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 will not be guaranteed when the **ungetc** function is called more than once without intervening **fflush**, **fseek**, or **rewind** function execution. When the **ungetc** function is executed, the current file position indicator for that file is moved back one position; however, when this file position indicator has already been positioned at the beginning of the file, its value will not be guaranteed.

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:

Normal:	When **size** or **n** is 0: 0 When **size** and **n** are both nonzero: Number of successfully input members
Abnormal:	—

Parameters:

ptr	Pointer to storage area to which data is input
size	Number of bytes in one member
n	Number of members to be input
fp	File pointer

Example:

```
#include <stdio.h>
void *ptr;
size_t size;
size_t n, ret;
FILE *fp;
    ret=fread(ptr, size, n, fp);
```

Remarks: The **fread** function inputs **n** members whose size is specified by **size**, from the stream input/output file indicated by file pointer **fp**, into the storage area pointed to by **ptr**. The file position indicator for the file is advanced by the number of bytes input.

The **fread** function returns the number of members successfully input, which is normally the same as the value of **n**. However, at end-of-file or when an error occurs, the number of members successfully input so far is returned, and then the return value will be less than **n**. The **ferror** and **feof** functions should be used to distinguish between end-of-file and error occurrence.

When the value of **size** or **n** is zero, zero is returned as the return value and the contents of the storage area pointed to by **ptr** are unchanged. When an error occurs, or when only a part of the members can be input, the file position indicator will not be guaranteed.

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: Normal: Number of successfully output members

Abnormal: —

Parameters:

ptr	Pointer to storage area storing data to be output
size	Number of bytes in one member
n	Number of members to be input
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 inputs **n** members whose size is specified by **size**, from the storage area pointed to by **ptr**, to the stream input/output file indicated by file pointer **fp**. The file position indicator for the file is advanced by the number of bytes output.

The **fwrite** function returns the number of members successfully output, which is normally the same as the value of **n**. However, when an error occurs, the number of members successfully output so far is returned, and then the return value will be less than **n**.

When an error occurs, the file position indicator will not be guaranteed.

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, which **fp** indicates, in the stream input/output file by the offset bytes from the position specified by **type** (the type of offset).
The types of offset are shown in table 10.38.
The **fseek** function normally returns zero, but returns nonzero in response to an invalid request.

Table 10.38 Types of Offset

Offset Type	Meaning
SEEK_SET	Shifts to a position which is located offset bytes away from the beginning of the file. The value specified by offset must be zero or positive.
SEEK_CUR	Shifts to a position which is located offset bytes away from the current position in the file. The shift is toward the end of the file if the value specified by offset is positive, and toward the beginning of the file if negative.
SEEK_END	Shifts to a position which is located offset bytes away from end-of-file. The value specified by offset 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:
- Normal: Current file position indicator position (text file)
  Number of bytes from beginning of file to current position (binary file)
- Abnormal: —

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, which **fp** indicates, in the stream input/output file.

For a binary file, the **ftell** function returns the number of bytes from the beginning of the file to the current position. For a text file, it returns, as the position of the file position indicator, an implementation-defined value that can be used by the **fseek** function.

When the **ftell** function is used twice for a text file, the difference in the return values will not necessarily represent the actual distance in the file.

void rewind(FILE *fp)

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:
Normal: End-of-file: Nonzero
Otherwise: 0

Abnormal: —

Parameters:
fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=feof(fp);
```

Remarks: The **feof** function tests for the end of the stream input/output file indicated by file pointer **fp**.

The **feof** function tests the end-of-file indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to indicate that the file is at its end. If the end-of-file indicator is not set, the **feof** function returns zero to show that the file is not yet at its end.

int ferror(FILE *fp)

Description: Tests for stream input/output file error state.

Header file: <stdio.h>

Return values: Normal: If file is in error state: Nonzero
Otherwise: 0
Abnormal: —

Parameters: fp File pointer

Example:
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=ferror(fp);
```

Remarks: The **ferror** function tests whether the stream input/output file indicated by file pointer **fp** is in the error state.

The **ferror** function tests the error indicator for the specified stream input/output file, and if the indicator is set, returns nonzero to show that the file is in the error state. If the error indicator is not set, the **ferror** function returns zero to show that the file is not in the error state.

void perror(const char *s)

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.

<stdlib.h>

Defines standard functions for standard processing of C program.

The following macros are implementation-defined.

Type	Definition Name	Description
Type (macro)	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: Normal: Converted data as a double type floating point number

Abnormal: —

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 not be guaranteed. 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: Normal: Converted data as an int type integer

Abnormal: —

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 not be guaranteed. 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: Normal: Converted data as a long type integer

Abnormal: —

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 not be guaranteed. When there are probabilities of a conversion error, use the **strtol** function.

double strtod(const char *nptr, char **endptr)

Description: Converts a string which represents a number to a double type floating point number.

Header file: <stdlib.h>

Return values:

Normal:	If the string pointed by **nptr** begins with a character that does not represent a floating point number: 0	
	If the string pointed by **nptr** begins with a character that represents a floating point number: converted data as a double type floating point number	
Abnormal:	If the converted data overflows: HUGE_VAL with the same sign as that of the string before conversion	
	If the converted data underflows: 0	

Parameters:

nptr	Pointer to a string representing a number to be converted
endptr	Pointer to the storage area containing a pointer to the first character that does not comprise 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 to **errno**.

Remarks: According to the rules described in section 10.1.3 (4), Floating-Point Operation Specifications, the **strtod** function converts data, from the first digit or the decimal point up to the character immediately before the character that does not comprise 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, the compiler assumes that the decimal point comes next to the last digit in the string. In the area pointed by **endptr**, the function sets up a pointer to the first character that does not consist a floating point number. If some characters that do not consist a floating point number come before numerals, the value of **nptr** is set. If **endptr** is NULL, nothing is set.

long strtol(const char *nptr, char **endptr, int base)

Description: Converts a string which represents a number to a long type integer.

Header file: <stdlib.h>

Return values:

Normal:	If the string pointed by **nptr** begins with a character that does not represent an integer: 0 If the string pointed by **nptr** begins with a character that represents an integer: Converted data as a long type integer
Abnormal:	If the converted data overflows: LONG_MAX or LONG_MIN depending on the sign of the string before conversion

Parameters:

nptr	Pointer to a string representing a number to be converted
endptr	Pointer to the storage area containing a pointer to the first character that does not comprise 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 to **errno**.

Remarks: The **strtol** function converts data, from the first numeral to the first character that does not represent an integer, into a long type integer.

In the storage area pointed by **endptr**, the function sets up a pointer to the first character that does not represent an integer. If some characters that do not represent an integer come before the first numeral, the value of **nptr** is set in this area. If **endptr** is NULL, nothing is set in this area.

If the value of **base** is 0, the rules described in section 10.1.1 (4), Integers, are observed at conversion. If the value of **base** is 2 to 36, it indicates the radix of conversion, where a (or A) to z (or Z) in the string to be converted correspond to numbers 10 to 35. If a character that is not smaller than the **base** value is found in the string to be converted, conversion stops immediately. A 0 after a sign is ignored at conversion. Similarly, 0x (or 0X) at base 16 is ignored.

int rand (void)

Description: Generates a pseudo-random integers from 0 to RAND_MAX.

Header file: <stdlib.h>

Return values: Normal: Pseudo-random integers

Abnormal: —

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 generation of the **rand** function. If pseudo-random number series generation by the **rand** function is repeated and if the same initial value is set up again by the **srand** function, the same pseudo-random number 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 a storage area and clears all bits in the allocated storage area to 0.

Header file: <stdlib.h>

Return values: Normal: Starting address of an allocated storage area

Abnormal: Storage allocation failed, or either of the parameter is 0: NULL

Parameters:

nelem	Number of elements
elsize	Number of bytes occupied by a single element

Example:

```
#include <stdlib.h>
size_t_nelem, elsize;
void *ret;
    ret=calloc(nelem, elsize);
```

Remarks: The **calloc** function allocates as many storage units of size **elsize** as the number specified by **nelem**. The function also clears all the bits in the allocated storage area 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 when the area has already been released by the **free** or **realloc** function, operation will not be guaranteed. Operation result of reference to released storage area will not also be guaranteed.

void *malloc(size_t size)

Description: Allocates storage area.

Header file: <stdlib.h>

Return values: Normal: Starting address of allocated storage area

Abnormal: Storage allocation failed, or **size** is 0: NULL

Parameters: size Size in number of bytes of storage area to allocate

Example:
```
#include <stdlib.h>
size_t size;
void *ret;
    ret=malloc(size);
```

Remarks: The **malloc** function allocates a storage area of a specified number of bytes by size.

void *realloc(void *ptr, size_t size)

Description: Changes the size of a storage area to a specified value.

Header file: <stdlib.h>

Return values:
- Normal: Starting address of storage area whose size has been changed
- Abnormal: Storage area allocation has failed, or **size** is 0: NULL

Parameters:
- ptr: Starting address of storage area to be changed
- size: Size of storage area in number of bytes after the change

Example:

```
#include <stdlib.h>
size_t size;
void *ptr, *ret;
    ret=realloc(ptr, size);
```

Remarks: The **realloc** function changes the size of a storage area specified by **ptr** to the number of bytes specified by **size**. If the newly allocated storage area is smaller than the old one, the contents are left unchanged up to the size of the newly allocated area.

When **ptr** is not a pointer to the storage area allocated by the **calloc**, **malloc**, or **realloc** function or when **ptr** is a pointer to the storage area released by the **free** or **realloc** function, operation will not be 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:

Normal:	If a matching member is found: pointer to the matching member If no matching member is found: NULL
Abnormal:	—

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 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 the table to be sorted
nmemb	Number of members to sort
size	Number of bytes of a member to be sorted
compar	Pointer to a function to perform comparison

Example:

```
#include <stdlib.h>
const void *base;
size_t nmemb, size;
int (*compar)(const void *, const void *)
    qsort(base, nmemb, size, compar);
```

Remarks: The **qsort** function sorts out data on the table pointed to by **base**. The data arrangement order is specified by the pointer to a function to perform comparison. This comparison function should receive pointers **p1** (first parameter) and **p2** (second parameter) as two data to be compared, 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: Normal: Absolute value of i

Abnormal: —

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, operation will not be guaranteed.

div_t div(int numer, int denom)

Description: Carries out division of int type integer and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Normal: Quotient and remainder of division of **numer** by **denom**

Abnormal: —

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: Normal: Absolute value of **j**

Abnormal: —

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 an long type integer, operation will not be guaranteed.

ldiv_t ldiv(long numer, long denom)

Description: Carries out division of long type integer and obtains the quotient and remainder.

Header file: <stdlib.h>

Return values: Normal: Quotient and remainder of division of **numer** by **denom**

Abnormal: —

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 a part of the source storage area and a part of the destination storage area overlap, correct copy is performed.

Implementation-Defined

Item	Compiler Specifications
Error message returned by the strerror function	Refer to section 12.3, Standard Library Error Messages.

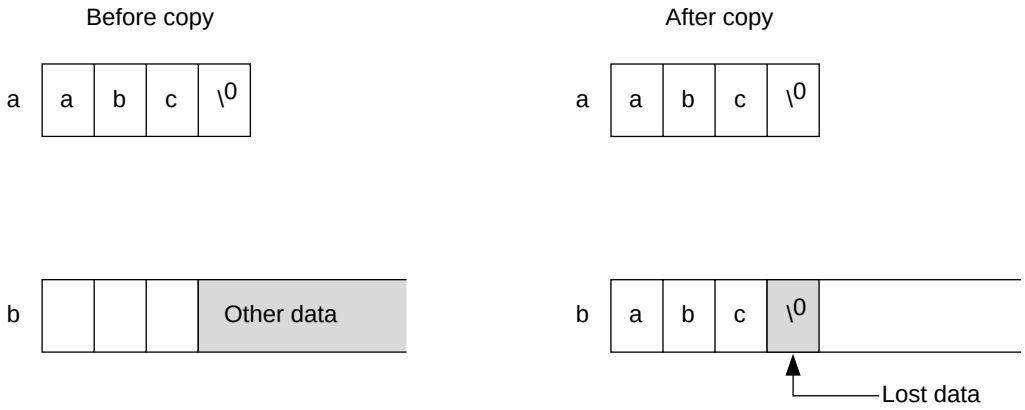
When using functions defined in this standard include file, note the following.

(1) On copying a string, if the destination area is smaller than the source area, operation will not be guaranteed.

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**.

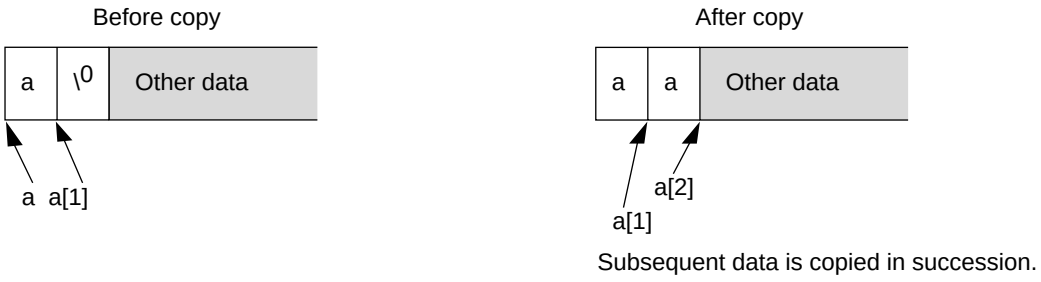


(2) On copying a string, if the source area overlaps the destination area, operation will not be guaranteed.

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 *sl, 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: Normal: **s1** value

Abnormal: —

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source storage area
n	Number of characters to be copied

Example:

```
#include <string.h>
long *ret, *s1;
const void *s2;
size_t n;
    ret=memcpy(s1, s2, n);
```

char *strcpy(char *sl, 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: Normal: **s1** value

Abnormal: —

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: Normal: **s1** value

Abnormal: —

Parameters:

s1	Pointer to destination storage area
s2	Pointer to source string
n	Number of characters to be copied

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
    ret=strncpy(s1, s2, n);
```

Remarks: The **strncpy** function copies a string pointed by **s2** up to **n** characters 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: Normal: **s1** value

Abnormal: —

Parameters:

s1	Pointer to the string after which another string is appended
s2	Pointer to the string to be added after the other string

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
    ret=strcat(s1, s2);
```

Remarks: The **strcat** function copies 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: Normal: **s1** value

Abnormal: —

Parameters:

s1	Pointer to the string after which another string is appended
s2	Pointer to the string to be appended after the other string
n	Number of characters to concatenate

Example:

```
#include <string.h>
char *s1, *ret;
const char *s2;
size_t n;
    ret=strncat(s1, s2, n);
```

Remarks: The strncat function copies up to **n** characters from the beginning of the string specified by **s2** at the end of another string specified by **s1**. The null character at the end of the **s1** string is replaced by the first character of the **s2** string. A null character is appended to the end of the concatenated string.

int memcmp(const void *s1, const void *s2, size_t n)

Description: Compares two storage areas specified.

Header file: <string.h>

Return values:

Normal:	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	
Abnormal:	—	

Parameters:

s1	Pointer to the reference storage area to be compared
s2	Pointer to the storage area to compare to the reference
n	Number of characters to compare

Example:

```
#include <string.h>
const void *s1, *s2;
size_t n;
int ret;
    ret=memcmp(s1, s2, n);
```

Remarks: The **memcmp** function compares the contents of the first **n** characters in the storage areas pointed by **s1** and **s2**. The 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:
- Normal: 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
- Abnormal: —

Parameters:
- s1: Pointer to the reference string to be compared
- s2: Pointer to the string to compare to the reference

Example:

```
#include <string.h>
const char *s1, *s2;
int ret;
    ret=strcmp(s1, s2);
```

Remarks: The **strcmp** function compares the contents of the strings pointed by **s1** and **s2**, and sets up the comparison result as a return value. The rule of comparison are implementation-defined.

int strncmp(const char *s1, const char *s2, size_t n)

Description: Compares two strings specified up to a specified length.

Header file: <string.h>

Return values:

Normal:	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	
Abnormal:	—	

Parameters:

s1	Pointer to the reference string to be compared
s2	Pointer to the string to compare to the reference
n	Maximum number of characters to compare

Example:

```
#include <string.h>
const char *s1, *s2;
size_t n;
int ret;
    ret=strncmp(s1, s2, n);
```

Remarks: The strncmp function compares the contents of the strings pointed by **s1** and **s2**, up to **n** characters. The 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:
- Normal: If the character is found: Pointer to the found character
  If the character is not found: NULL
- Abnormal: —

Parameters:

s	Pointer to the storage area to be searched
c	Character to search for
n	Number of characters to search

Example:

```
#include <string.h>
const void *s;
int c;
size_t n;
void *ret;
    ret=memchr(s, c, n);
```

Remarks: The **memchr** function searches the storage area specified by **s** from the beginning up to **n** characters, looking for the first occurrence of the character specified as **c**. If the **c** character is found, the function returns the pointer to the found character.

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:
- Normal: If the character is found: Pointer to the found character
  If the character is not found: NULL
- Abnormal: —

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 pointer to the found character.

The null character at the end of the **s** string is included in the search object.

size_t strcspn(const char *s1, const char *s2)

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:

Normal:	Number of characters at the beginning of the **s1** string that are not included in the **s2** string
Abnormal:	—

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check **s1**

Example:

```
#include <string.h>
const char *s1, *s2;
size_t ret;
    ret=strcspn(s1, s2);
```

Remarks: The **strcspn** function checks from the beginning of the string specified by **s1**, 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 the character that is included in another string specified.

Header file: <string.h>

Return values:
- Normal: If the character is found: Pointer to the found character
  If the character is not found: NULL
- Abnormal: —

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:
- Normal: If the character is found: Pointer to the found character
  If the character is not found: NULL
- Abnormal: —

Parameters:
- s: Pointer to the string to be searched
- c: Character to search for

Example:

```
#include <string.h>
const char *s;
int c;
char *ret;
    ret=strrchr(s, c);
```

Remarks: The **strrchr** function searches the string specified by **s** looking for the last occurrence of the character specified by **c**. If the **c** character is found, the function returns the pointer to the last occurrence of that character.

The null character at the end of the **s** string is included in the search objective.

size_t strspn(const char *s1, const char *s2)

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:

Normal:	Number of characters at the beginning of the **s1** string that are included in the **s2** string
Abnormal:	—

Parameters:

s1	Pointer to the string to be checked
s2	Pointer to the string used to check **s1**

Example:

```
#include <string.h>
const char *s1, *s2;
size_t ret;
    ret=strspn(s1, s2);
```

Remarks: The **strspn** function checks from the beginning of the string specified by **s1**, 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:
- Normal: If the string is found: Pointer to the found string
  If the string is not found: NULL
- Abnormal: —

Parameters:
- s1 Pointer to the string to be searched
- s2 Pointer to the string to search for

Example:

```
#include <string.h>
const char *s1, *s2;
char *ret;
    ret=strstr(s1, s2);
```

Remarks: The **strstr** function searches the string specified by **s1** looking for the first occurrence of another string specified by **s2**, and returns the pointer to the first occurrence.

char *strtok(char *s1, const char *s2)

Description: Divides a specified string into some tokens.

Header file: <string.h>

Return values:
- Normal: If division into tokens is successful: Pointer to the first token divided
  If division into tokens is unsuccessful: NULL
- Abnormal: —

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 a pointer to the beginning of that token. Otherwise, the function returns NULL.

(2) Second and subsequent calls

Starting from the next character separated before as the token, the function repeats division at a character included in the string pointed by **s2**. If a token has been separated, the function returns a pointer to the beginning of that token. Otherwise, the function returns NULL.

At the second and subsequent calls, specify NULL as the first parameter.

The string pointed by **s2** can be changed at each call.

The null character is appended 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, a pointer to character 'a' is returned, and the null character is embedded at '@,' the first delimiter after character 'a.' Thus string 'a' has been separated.

Specify NULL for the first parameter to consecutively separate tokens from the same string, and repeat calling the **strtok** function.

Consequently, the function separates strings 'b,' 'c,' and 'd.'

void *memset(void *s, 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: Normal: Value of **s**

Abnormal: —

Parameters:

s	Pointer to storage area to set characters in
c	Character to be set
n	Number of characters to be set

Example:

```
#include <string.h>
void *s, *ret;
int c;
size_t n;
    ret=memset(s, c, n);
```

Remarks: The **memset** function sets the character specified by **c** 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:
Normal: Pointer to the error message (string) corresponding to the specified error number
Abnormal: —

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, operation will not be guaranteed.

size_t strlen(const char *s)

Description: Calculates the length of a string.

Header file: <string.h>

Return values:
Normal: Number of characters of the string
Abnormal: —

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 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: Normal: Value of **s1**

Abnormal: —

Parameters:

s1	Pointer to the destination storage area
s2	Pointer to the source storage area
n	Number of characters to be copied

Example:

```
#include <string.h>
void *ret, *s1
const void *s2;
size_t n;
     ret=memmove(s1, s2, n);
```

10.3.3 Reentrant Library

Table 10.40 lists reentrant libraries. A function that is marked with Δ in the table sets the **errno** variable. Such a function can be assumed to be reentrant unless the program refers to **errno**.

Table 10.40 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	iscntrl	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	Δ
		17	asin	Δ
		18	atan	Δ
		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

Table 10.40 Reentrant Library List (cont)

No.	Standard Include File		Function Name	Reentrant
4	math.h(cont)	29	log	Δ
		30	log10	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
		52	log10f	Δ
		53	modff	Δ
		54	powf	Δ
		55	sqrtf	Δ
		56	ceilf	Δ
		57	fabsf	Δ
		58	floorf	Δ
		59	fmodf	Δ

Table 10.40 Reentrant Library List (cont)

No.	Standard Include File		Function Name	Reentrant
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
		84	getc	X
		85	getchar	X
		86	gets	X
		87	putc	X
		88	putchar	X
		89	puts	X
		90	ungetc	X
		91	fread	X

Table 10.40 Reentrant Library List (cont)

No.	Standard Include File		Function Name	Reentrant
8	stdio.h (cont)	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
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

Table 10.40 Reentrant Library List (cont)

No.	Standard Include File		Function Name	Reentrant
10	string.(cont)	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 10.41 lists the libraries not supported by this compiler.

Table 10.41 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.

Section 11 Assembly Specifications

11.1 Program Elements

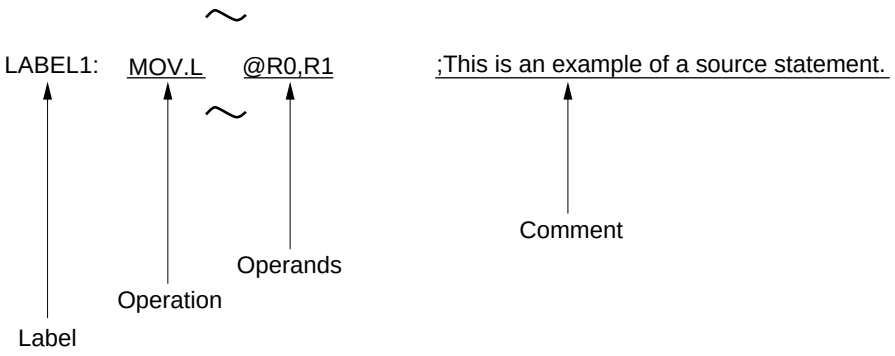
11.1.1 Source Statements

(1) Source Statement Structure

The following shows the structure of a source statement.

[<label>] [Δ<operation>[Δ<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, a DSP instruction, an extended instruction, an assembler directive, or a directive statement is written as the operation.
Executable instructions and DSP instructions are microprocessor instructions defined by the assembler.
Extended instructions are instructions that are expanded into executable instructions and constant data (literals) or several executable instructions.
Assembler directives are instructions that give directions to the assembler.
Directive statements are used for file inclusion, conditional assembly, 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. String literals and comments can include Japanese characters (shift JIS code or EUC code) or LATIN1 code character.
In principle, a single statement must be written on a single line. The maximum length of a line is 8192 bytes.

(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 symbols.

- Register names
  R0 to R15, FR0 to FR15, DR0 to DR14 (only even values), XD0 to XD14 (only even values), FV0 to FV12 (only multiples of four), R0_BANK to R7_BANK, SP*, SR, GBR, VBR, MACH, MACL, PR, PC, SSR, SPC, FPUL, FPSCR, MOD, RE, RS, DSR, A0, A0G, A1, A1G, M0, M1, X0, X1, Y0, Y1, XMTRX, DBR, SGR
- Operators
  STARTOF, SIZEOF, HIGH, LOW, HWORD, LWORD, $EVEN, $ODD, $EVEN2, $ODD2
- Location counter
  $

Note: R15 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.
— 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.
        ~
```

```
        ~
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=4
                          ;.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

Reserved words cannot be used as symbols. Names of the following type must not be used because such names are used as internal symbols by the assembler.
_$$nnnnn (n is 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 and .GLOBAL directives. External references can be defined by .IMPORT and .GLOBAL 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 character codes.
Character constants are written by enclosing up to four-byte 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.
```

(3) Floating-Point Constants

Floating-point constants can be specified as operands in assembler directives for reserving floating-point constants.

(a) Floating-Point Constant Representation:

Floating-point constants can be represented in decimal and hexadecimal.

- Decimal representation

F'[{±}] {n[.[m]] / .m} [t[[{±}]xx]]

F'	Indicates that the number is decimal. It cannot be omitted.
[{±}] {n[.[m]] / .m}	"n" indicates the integer part in decimal. "m" indicates the fraction part in decimal. Either the integer part or the fraction part can be omitted. If the sign (±) is omitted, the assembler assumes it is positive.
t	Indicates that the number is in either of the following precisions S: Single precision D: Double precision If omitted, the assembler assumes the operation size of the assembler directive.
[{±}]xx	Indicates the exponent part in decimal. If omitted, the assembler assumes 0. If the sign (±) is omitted, the assembler assumes it is positive.

Example:

F'0.5S–2 = 0.5×10^{-2} = 0.005 = H'3BA3D70A

F'.123D3 = 0.123×10^{3} = 123 = H'405EC00000000000

- Hexadecimal representation

  H'xxxx[.t]

H'	Indicates that the number is hexadecimal. It cannot be omitted.
xxxx	Indicates the bit pattern of the floating-point constant in hexadecimal. If the bit pattern is shorter than the specified data length, it is aligned to the right end of the reserved area and 0s are added to the remaining bits in the reserved area. If the bit pattern is longer than the specified data length, the right-side bits of the bit pattern are allocated for the specified data length and the remaining bits of the bit pattern are ignored.
t	Indicates that the number is in either of the following precisions S: Single precision D: Double precision If omitted, the assembler assumes the operation size of the assembler directive.

  This format directly specifies the bit pattern of the floating-point constant to represent data that is difficult to represent in decimal format, such as 0s or infinity for the specified precision.

  Example:

  H'0123456789ABCDEF.S = H'89ABCDEF

  H'FFFF.D = H'000000000000FFFF

(b) Floating-Point Data Range:

Table 11.1 lists the floating-point data types.

Table 11.1 Floating-Point Data Types

Data Type	Description
Normalized number	The absolute value is between the underflow and overflow boundaries including the boundary values.
Denormalized number	The absolute value is between 0 and the underflow boundary.
Zero	The absolute value is 0.
Infinity	The absolute value is larger than the overflow boundary.
Not-a-Number (NAN)	A value that is not a numerical value. Includes sNAN (signaling NAN) and qNAN (quiet NAN).

These data types are shown on the following number line. NAN cannot be shown on the number line because it is not handled as a numerical value.

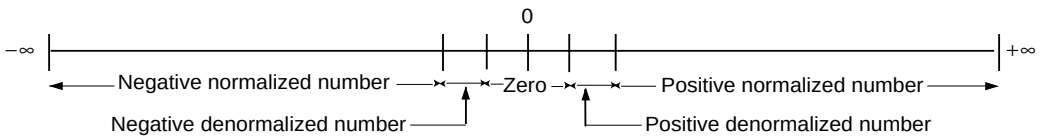


Table 11.2 lists the numerical value ranges the assembler can use.

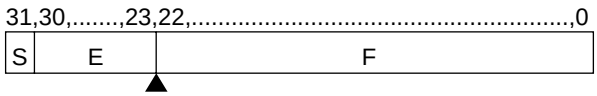
Table 11.2 Data Types and Numerical Value Ranges (Absolute Value)

Data Type		Single Precision	Double Precision
Normalized number	Maximum value	3.40×10^{38}	1.79×10^{308}
	Minimum value	1.18×10^{-38}	2.23×10^{-308}
Denormalized number	Maximum value	1.17×10^{-38}	2.22×10^{-308}
	Minimum value	1.40×10^{-45}	4.94×10^{-324}

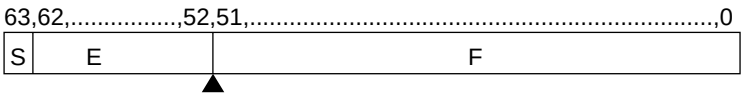
(c) Floating-Point Data Format:

The floating-point data format is shown below:

Single Precision:



Double Precision:



▲ : Decimal point
S : Sign bit
E : Exponent part
F : Fraction part

- Sign bit (S)

  Indicates the sign of a value. Positive and negative are represented by 0 and 1, respectively.

- Exponent part (E)

  Indicates the exponent of a value. The actual exponent value is obtained by subtracting the bias value from the value specified in this exponent part.

- Fraction part (F)

  Each bit has its own significance and corresponds to 2^{-1}, 2^{-2}, ..., 2^{-n} from the start bit, respectively ("n" is the bit length of the fraction part).

Table 11.3 shows the size of each parameter in data format.

Table 11.3 Data Format Size

Parameter	Single Precision	Double Precision
Bit length	32 bits	64 bits
Sign bit (S)	1 bit	1 bit
Exponent part (E)	8 bits	11 bits
Fraction part (F)	23 bits	52 bits
Bias of exponent value	127	1023

A floating-point number is represented using the symbols in table 11.3 as follows:

$$2^{E\text{-bias}} \cdot (-1)^{s} \cdot \left\{ \begin{matrix} (1.\ F) \\ (0.\ F) \end{matrix} \right\} \begin{matrix} : \text{Normalized number} \\ : \text{Denormalized number} \end{matrix}$$

$$(1.\ F) = 1 + b_0 \times 2^{-1} + b_1 \times 2^{-2} + ... + b_{n-1} \times 2^{-n}$$
$$(0.\ F) = b_0 \times 2^{-1} + b_1 \times 2^{-2} + ... + b_{n-1} \times 2^{-n}$$

b: Bit location in the fraction part
n: Bit length of the fraction part

Table 11.4 shows the floating-point representation for each data type. NAN cannot be represented because it is not handled as a numerical value.

Table 11.4 Floating-Point Representation for Each Data Type

Data Type	Single Precision	Double Precision
Normalized number	$(-1)^{s} \cdot 2^{E-127} \cdot (1.\ F)$	$(-1)^{s} \cdot 2^{E-1023} \cdot (1.\ F)$
Denormalized number	$(-1)^{s} \cdot 2^{-126} \cdot (0.\ F)$	$(-1)^{s} \cdot 2^{-1022} \cdot (0.\ F)$
Zero	$(-1)^{s} \cdot 0$	$(-1)^{s} \cdot 0$
Infinity	$(-1)^{s} \cdot \infty$	$(-1)^{s} \cdot \infty$
Not-a-Number (NAN)	quiet NAN, signaling NAN	quiet NAN, signaling NAN

(d) Rounding of Floating-Point Constants:

When converting floating-point constants used in assembler directives for reserving floating-point numbers into object codes, the assembler rounds them in the following two modes to set the valid range.

- Round to Nearest even (RN)

  Rounds the least significant bit in the object code to its nearest absolute value. When two absolute values are at the same distance, rounds the least significant bit to become zero.
- Round to Zero (RZ)

  Rounds the least significant bit toward zero.

Example:

Object code of .FDATA.S F' 1S-1

RN: H'3DCCCCCD

RZ: H'3DCCCCCC

(e) Handling Denormalized Numbers:

The assembler handles denormalized numbers differently depending on the target CPU. In a CPU that does not handle denormalized numbers, if a value in the denormalized number range is used, warning 841 occurs and the object code is output as zero.
In a CPU that handles denormalized numbers, if a value in the denormalized number range is used, warning 842 occurs and the object code is output in denormalized numbers.
How to handle denormalized numbers can be switched with the **denormalize** command line option.

Example:

CPU not handling denormalized numbers:

.FDATA.S F' 1S-40 Warning 841, Object code H'00000000

CPU handling denormalized numbers:

.FDATA.S F' 1S-40 Warning 842, Object code H'000116C2

(4) Fixed-Point Constants

Fixed-point constants can be specified as operands in the assembler directive for reserving fixed-point data.

(a) Fixed-Point Number Representation:

Fixed-point numbers express real numbers ranging from –1.0 to 1.0 in decimal.
Word size and longword size are available for fixed-point numbers.

- Word-size fixed-point numbers

  Two-byte signed integers expressing real numbers ranging from –1.0 to 1.0.
  The real number expressed by 2-byte signed integer x (–32,768 < = x < = 32,767) is x/32768.

  Example:

Fixed-point number	Word-size representation
–1.0	H'8000
–0.5	H'C000
0.0	H'0000
0.5	H'4000
1.0	H'7FFF

- Longword-size fixed-point numbers

  Four-byte signed integers expressing real numbers ranging from –1.0 to 1.0. The real number expressed by 4-byte signed integer x (–2,147,483,648 < = x < = 2,147,483,647) is x/2147483648.

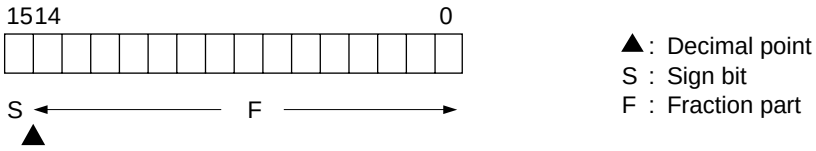
  Example:

Fixed-point number	Longword-size representation
–1.0	H'80000000
–0.5	H'C0000000
0.0	H'00000000
0.5	H'40000000
1.0	H'7FFFFFFF

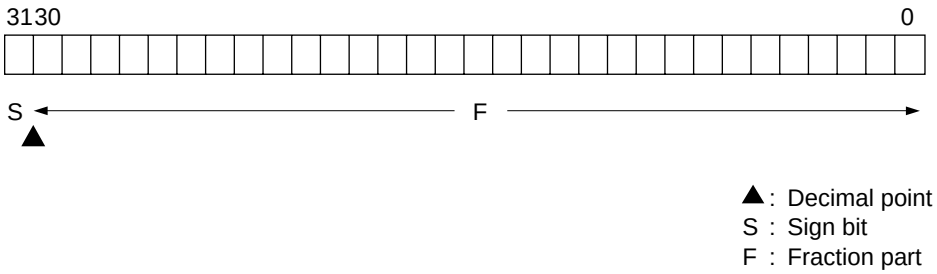
(b) Fixed-Point Data Format:

The fixed-point data format consists of a sign bit and a 15-bit fraction part in word size, and a sign bit and a 31-bit fraction part in longword size. The decimal point is assumed to be fixed on the right of the sign bit.

- Word size



- Longword size



— Sign bit (S)

Indicates the sign of a value. Positive and negative are represented by 0 and 1, respectively.

— Fraction part (F)

Each bit has its own significance and corresponds to 2^{-1}, 2^{-2}, ..., 2^{-31} from the start bit, respectively.

(c) Valid Range for Fixed-Point Numbers:

In longword size, 31 bits can represent nine digits of data in decimal, but the assembler handles ten digits in decimal as a valid number, rounds the 35th bit in RN (round to the nearest absolute value) mode, and uses the high-order 31 bits of the result as fixed-point data.

Note: The actual fixed-point data range is –1.0 to 0.9999999999, but the assembler assumes 1.0 as 0.9999999999 and represents it as H'7FFFFFFF.

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
                                ;a length of 2 bytes.
                                ;The location counter changes to H'00001002.
.DATA.W         H'F0            ;The object code generated by this assembler directive has
                                ;a length of 2 bytes.
                                ;The location counter changes to H'00001004.
.DATA.W         H'10            ;The object code generated by this assembler directive has
                                ;a length of 2 bytes.
                                ;The location counter changes to H'00001006.
.ALIGN          4               ;The value of the location counter is corrected to be a
                                ;multiple of 4.
                                ;The location counter changes to H'00001008.
.DATA.L         H'FFFFFFFF      ;The object code generated by this assembler directive has
                                ;a length of 4 bytes.
                                ;The location counter changes to H'0000100C.
                                ;.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.
                                ;.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 symbol.

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 reference ($)
- 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 11.5 shows the operators supported by the assembler.

Table 11.5 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>
Even/odd operations	$EVEN	1 when the value is a multiple of 2, and 0 otherwise	$EVEN <symbol>
	$ODD	0 when the value is a multiple of 2, and 1 otherwise	$ODD <symbol>
	$EVEN2	1 when the value is a multiple of 4, and 0 otherwise	$EVEN2 <symbol>
	$ODD2	0 when the value is a multiple of 4, and 1 otherwise	$ODD2 <symbol>
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>

(c) Parentheses

Parentheses modify the operation precedence.

(2) 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 multiple 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 11.6 shows the operator precedence and the association rule.

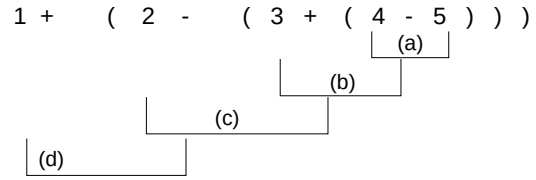
Table 11.6 Operator Precedence and Association Rules

Precedence	Operator	Association Rule
1 (high)	+ – ~ STARTOF SIZEOF $EVEN $ODD $EVEN2 $ODD2 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:

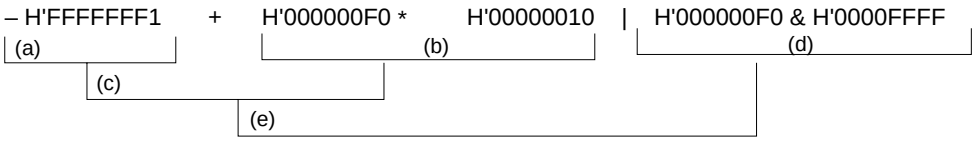


The assembler calculates this expression in the order (a) to (d).

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:

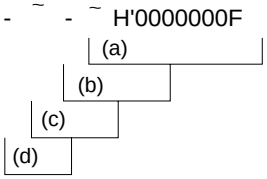


The assembler calculates this expression in the order (a) to (e).

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.

(3) 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      SH1
         .SECTION  INIT_RAM,DATA,ALIGN=4
         .RES.B    H'100
         .SECTION  INIT_DATA,DATA,ALIGN=4
INIT_BGN .DATA.L   (STARTOF  INIT_RAM).....................; (1)
INIT_END .DATA.L   (STARTOF  INIT_RAM) + (SIZEOF INIT_RAM)...; (2)
;
;
         .SECTION  MAIN,CODE,ALIGN=4
INITIAL:
         MOV.L     DATA1,R6
         MOV       #0,R5
         MOV.L     DATA1+4,R3
         BRA       LOOP2
         MOV.L     @R3,R4
LOOP1:
         MOV.L     R5,@R4
         ADD       #4,R4
LOOP2:
         MOV.L     @R6,R3
         CMP/HI    R3,R4
         BF        LOOP1
         RTS
         NOP
DATA1:
        .DATA.L    INIT_END
        .DATA.L    INIT_BGN
        .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

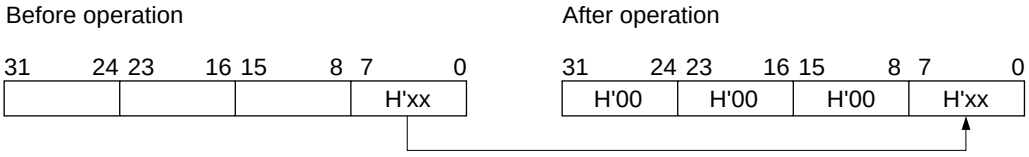
Extracts the high-order byte from the low-order two bytes of a 4-byte value.

Example:

```
LABEL  .EQU   H'00007FFF
       .DATA  HIGH LABEL        ;  Reserves integer data H'0000007F on memory.
```

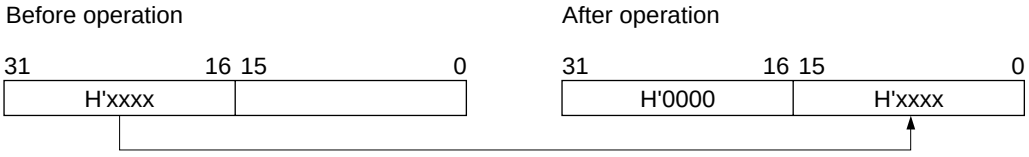
(d) LOW Operation

Extracts the lowest-order one byte from a 4-byte value.



(e) HWORD Operation

Extracts the high-order two bytes from a 4-byte value.



(f) LWORD Operation

Extracts the low-order two bytes from a 4-byte value.

(g) Even/Odd Operation

Determines if the value of the address symbol is a multiple of 2 or 4.
Table 11.7 shows the even/odd operations.

Table 11.7 Even/Odd Operations

Operator	Operation
$EVEN	1 when the value is a multiple of 2, and 0 otherwise
$ODD	0 when the value is a multiple of 2, and 1 otherwise
$EVEN2	1 when the value is a multiple of 4, and 0 otherwise
$ODD2	0 when the value is a multiple of 4, and 1 otherwise

Example:

To obtain the current program counter value using an $ODD2 operator.

```
LAB:
        MOVA    @(0,PC),R0
        ADD     #-4+2*$ODD2 LAB,R0      ;$ODD2 gives 0 when LAB is
                                        ;a multiple of 4, and gives 1 when
                                        ;LAB is not a multiple of 4.
```

(4) Notes on Expressions

(a) Internal Processing

The assembler regards expression values as 32-bit signed values.

Example:

~H'F0

The assembler regards H'F0 as H'000000F0.

Therefore, the value of ~H'F0 is H'FFFFFF0F. (Note that this is not H'0000000F.)

(b) Arithmetic Operators

Where values must be determined at assembly, the multiplication and division operators cannot take terms that contain relative values 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.
```

(c) Logic Operators

The logic operators cannot take terms that contain relative values as their operands.

11.1.7 String Literals

String literals are sequences of character data.
The following ASCII characters can be used in string literals.

ASCII code H'09 (tab)
H'20 (space) to H'7E (tilde)

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** option.
String literals 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    " " " 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 string literals is as follows.
Character constants are numeric values. They have a data size of either 1 byte, 2 bytes, or 4 bytes.
String literals 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, .ASSIGN, .REG, or .FREG
- Operand in .EXPORT, .IMPORT, or .GLOBAL

Example:

```
LABEL1:                                 ;Local block 1 start
?0001:
                ~
                CMP/EQ      R1,R2
                BT          ?0002       ;Branches to ?0002 of local block 1
                BRA         ?0001       ;Branches to ?0001 of local block 1
?0002:
                ~
LABEL2:                                 ;Local block 2 start
?0001:          ~
                CMP/GE      R1,R2
                BT          ?0002       ;Branches to ?0002 of local block 2
                BRA         ?0001       ;Branches to ?0001 of local block 2
?0002:
LABEL3:                                 ;Local block 3 start
```

(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, 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, .ASSIGN, .REG, or .FREG 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>:] △<mnemonic>[.<operation size>] [△<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)
S	Single precision (4 bytes)
D	Double precision (8 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 11.8 lists the addressing modes.

Table 11.8 Addressing Modes

Addressing Mode	Name	Description
Rn	Register direct	The contents of the specified register.
@Rn	Register indirect	A memory location. The value in Rn gives the start address of the memory accessed.
@Rn+	Register indirect with post-increment	A memory location. The value in Rn (before being incremented*1) gives the start address of the memory accessed. The microprocessor first uses the value in Rn for the memory reference, and increments Rn afterwards.
@–Rn	Register indirect with pre-decrement	A memory location. The value in Rn (after being decremented*2) gives the start address of the memory accessed. The microprocessor first decrements Rn, and then uses that value for the memory reference.
@(disp,Rn)	Register indirect with displacement*3	A memory location. The start address of the memory access is given by: the value of Rn plus the displacement (disp). The value of Rn is not changed.
@(R0,Rn)	Register indirect with index	A memory location. The start address of the memory access is given by: the value of R0 plus the value of Rn. The values of R0 and Rn are not changed.
@(disp,GBR)	GBR indirect with displacement	A memory location. The start address of the memory access is given by: the value of GBR plus the displacement (disp). The value of GBR is not changed.
@(R0,GBR)	GBR indirect with index	A memory location. The start address of the memory access is given by: the value of GBR plus the value of R0. The values of GBR and R0 are not changed.
@(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).

Notes 1 to 3: See next page.

Table 11.8 Addressing Modes (cont)

Addressing Mode	Name	Description
symbol	PC relative specified with symbol	[When used as the operand of a branch instruction] The symbol directly indicates the destination address. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol – PC.
		[When used as the operand of a data move instruction] A memory location. The symbol indicates the start address of the memory accessed. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol – PC.
		[When used as the operand of an instruction that specifies the RS or RE register (LDRS or LDRE instruction)] A memory location. The symbol indicates the start address of the memory accessed. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol – PC.
#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 (two bytes), and 4 when the operation size is a longword (four 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.

The values that can be used for the displacement vary with the addressing mode and the operation size.

Table 11.9 Allowed Displacement Values

Addressing Mode	Displacement*
@(disp,Rn)	When the operation size is byte (B): H'00000000 to H'0000000F (0 to 15)
	When the operation size is word (W): H'00000000 to H'0000001E (0 to 30)
	When the operation size is longword (L): H'00000000 to H'0000003C (0 to 60)
@(disp,GBR)	When the operation size is byte (B): H'00000000 to H'000000FF (0 to 255)
	When the operation size is word (W): H'00000000 to H'000001FE (0 to 510)
	When the operation size is longword (L): H'00000000 to H'000003FC (0 to 1020)
@(disp,PC)	[When used as an operand of a move instruction]
	When the operation size is word (W): H'00000000 to H'000001FE (0 to 510)
	When the operation size is longword (L): H'00000000 to H'000003FC (0 to 1020)
	[When used as an operand of an instruction that sets the RS or RE register (LDRS or LDRE)]
	H'FFFFFF00 to H'000000FE (–256 to 254)

Note: Units are bytes, and numbers in parentheses are decimal.

Table 11.9 Allowed Displacement Values (cont)

Addressing Mode	Displacement*
symbol	[When used as a branch instruction operand] When used as an operand for a conditional branch instruction (BT, BF, BF/S, or BT/S): { H'00000000 to H'000000FF (0 to 255) H'FFFFFF00 to H'FFFFFFFF (–256 to –1) When used as an operand for an unconditional branch instruction (BRA or BSR) { H'00000000 to H'00000FFF (0 to 4095) H'FFFFF000 to H'FFFFFFFF (–4096 to –1) [When used as the operand of a data move instruction] When the operation size is word (W): H'00000000 to H'000001FE (0 to 510) When the operation size is longword (L): H'00000000 to H'000003FC (0 to 1020) [When used as an operand of an instruction that sets the RS or RE register (LDRS or LDRE)] H'FFFFFF00 to H'000000FE (–256 to 254)

Note: Units are bytes, and numbers in parentheses are decimal.

The values that can be used for immediate values vary with the executable instruction.

Table 11.10 Allowed Immediate Values

Executable Instruction	Immediate Value	
TST, AND, OR, XOR	H'00000000 to H'000000FF	(0 to 255)
MOV	{ H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to –1)*1
ADD, CMP/EQ	{ H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to –1)*1
TRAPA	H'00000000 to H'000000FF	(0 to 255)
SETRC	H'00000001 to H'000000FF	(1 to 255)*2

Notes: 1. Values in the range H'FFFFFF80 to H'FFFFFFFF can be written as positive decimal values.

2. When zero is set as the immediate values of the SETRC instruction, warning number 835 occurs and the object code is output as zero. In this case, the range to be repeated is executed once.
When an externally referenced symbol is set as the immediate values of the SETRC instruction, the linkage editor checks the range from H'00000000 to H'000000FF (0 to 255).

Note: The assembler corrects the value of displacements under certain conditions.

Condition		Type of Correction
When the operation size is a word and the displacement is not a multiple of 2	→	The lowest bit of the displacement is discarded, resulting in the value being a multiple of 2.
When the operation size is a longword and the displacement is not a multiple of 4	→	The lower 2 bits of the displacement are discarded, resulting in the value being a multiple of 4.
When the displacement of the branch instruction is not a multiple of 2	→	The lowest bit of the displacement is discarded, resulting in the value being a multiple of 2.

Be sure to take this correction into consideration when using operands of the mode @(disp,Rn), @(disp,GBR), and @(disp,PC).

— Example:

MOV.L @(63,PC),R0

The assembler corrects the 63 to be 60, and generates object code identical to that for the statement MOV.L @(60,PC),R0, and warning 870 occurs.

11.2.2 Notes on Executable Instructions

(1) Notes on the Operation Size

The operation size that can be specified vary with the mnemonic and the addressing mode combination.

(a) SH-1 Executable Instruction and Operation Size Combinations:

Table 11.11 shows the SH-1 allowable executable instruction and operation size combinations.

Symbol meanings:

Rn, Rm	A general register (R0 to R15)
R0	General register R0
SR	Status register
GBR	Global base register
VBR	Vector base register
MACH, MACL	High-order and Low-Order Multiplication and accumulation registers
PR	Procedure register
PC	Program counter
imm	An immediate value
disp	A displacement value
symbol	A symbol
B	Byte
W	Word (2 bytes)
L	Longword (4 bytes)
❍	Valid specification
×	Invalid specification: The assembler regards instructions with this combination as the specification being omitted.
Δ	The assembler regards them as extended instructions.

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 1)

Data Move Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**	
MOV	#imm,Rn	O	Δ	Δ	B	*1
MOV	@(disp,PC),Rn	×	O	O	L	
MOV	symbol,Rn	×	O	O	L	
MOV	Rn,Rm	×	×	O	L	
MOV	Rn,@Rm	O	O	O	L	
MOV	@Rn,Rm	O	O	O	L	
MOV	Rn,@–Rm	O	O	O	L	
MOV	@Rn+,Rm	O	O	O	L	
MOV	R0,@(disp,Rn)	O	O	O	L	
MOV	Rn,@(disp,Rm)	×	×	O	L	*2
MOV	@(disp,Rn),R0	O	O	O	L	
MOV	@(disp,Rn),Rm	×	×	O	L	*3
MOV	Rn,@(R0,Rm)	O	O	O	L	
MOV	@(R0,Rn),Rm	O	O	O	L	
MOV	R0,@(disp,GBR)	O	O	O	L	
MOV	@(disp,GBR),R0	O	O	O	L	
MOVA	#imm,R0	×	×	Δ	L	
MOVA	@(disp,PC),R0	×	×	O	L	
MOVA	symbol,R0	×	×	O	L	
MOVT	Rn	×	×	O	L	
SWAP	Rn,Rm	O	O	×	W	
XTRCT	Rn,Rm	×	×	O	L	

Notes: 1. In size selection mode, the assembler selects the operation size according to the imm value.
2. In this case, Rn must be one of R1 to R15.
3. In this case, Rm must be one of R1 to R15.

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 2)

Arithmetic Operation Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**	
ADD	Rn,Rm	×	×	O	L	
ADD	#imm,Rn	×	×	O	L	
ADDC	Rn,Rm	×	×	O	L	
ADDV	Rn,Rm	×	×	O	L	
CMP/EQ	#imm,R0	×	×	O	L	
CMP/EQ	Rn,Rm	×	×	O	L	
CMP/HS	Rn,Rm	×	×	O	L	
CMP/GE	Rn,Rm	×	×	O	L	
CMP/HI	Rn,Rm	×	×	O	L	
CMP/GT	Rn,Rm	×	×	O	L	
CMP/PZ	Rn	×	×	O	L	
CMP/PL	Rn	×	×	O	L	
CMP/STR	Rn,Rm	×	×	O	L	
DIV1	Rn,Rm	×	×	O	L	
DIV0S	Rn,Rm	×	×	O	L	
DIV0U	(no operands)	×	×	×	—	
EXTS	Rn,Rm	O	O	×	W	
EXTU	Rn,Rm	O	O	×	W	
MAC	@Rn+,@Rm+	×	O	×	W	
MULS	Rn,Rm	×	O	O	L	*
MULU	Rn,Rm	×	O	O	L	*
NEG	Rn,Rm	×	×	O	L	
NEGC	Rn,Rm	×	×	O	L	
SUB	Rn,Rm	×	×	O	L	
SUBC	Rn,Rm	×	×	O	L	
SUBV	Rn,Rm	×	×	O	L	

Note: The object code generated when W is specified is the same as that generated when L is specified.

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 3)

Logic Operation Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
AND	Rn,Rm	×	×	O	L
AND	#imm,R0	×	×	O	L
AND	#imm,@(R0,GBR)	O	×	×	B
NOT	Rn,Rm	×	×	O	L
OR	Rn,Rm	×	×	O	L
OR	#imm,R0	×	×	O	L
OR	#imm,@(R0,GBR)	O	×	×	B
TAS	@Rn	O	×	×	B
TST	Rn,Rm	×	×	O	L
TST	#imm,R0	×	×	O	L
TST	#imm,@(R0,GBR)	O	×	×	B
XOR	Rn,Rm	×	×	O	L
XOR	#imm,R0	×	×	O	L
XOR	#imm,@(R0,GBR)	O	×	×	B

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 4)

Shift Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
ROTL	Rn	×	×	O	L
ROTR	Rn	×	×	O	L
ROTCL	Rn	×	×	O	L
ROTCR	Rn	×	×	O	L
SHAL	Rn	×	×	O	L
SHAR	Rn	×	×	O	L
SHLL	Rn	×	×	O	L
SHLR	Rn	×	×	O	L
SHLL2	Rn	×	×	O	L
SHLR2	Rn	×	×	O	L
SHLL8	Rn	×	×	O	L
SHLR8	Rn	×	×	O	L
SHLL16	Rn	×	×	O	L
SHLR16	Rn	×	×	O	L

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 5)

Branch Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
BF	symbol	×	×	×	—
BT	symbol	×	×	×	—
BRA	symbol	×	×	×	—
BSR	symbol	×	×	×	—
JMP	@Rn	×	×	×	—
JSR	@Rn	×	×	×	—
RTS	(no operands)	×	×	×	—

Table 11.11 SH-1 Executable Instruction and Operation Size Combinations (Part 6)

System Control Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
CLRT	(no operands)	×	×	×	—
CLRMAC	(no operands)	×	×	×	—
LDC	Rn,SR	×	×	O	L
LDC	Rn,GBR	×	×	O	L
LDC	Rn,VBR	×	×	O	L
LDC	@Rn+,SR	×	×	O	L
LDC	@Rn+,GBR	×	×	O	L
LDC	@Rn+,VBR	×	×	O	L
LDS	Rn,MACH	×	×	O	L
LDS	Rn,MACL	×	×	O	L
LDS	Rn,PR	×	×	O	L
LDS	@Rn+,MACH	×	×	O	L
LDS	@Rn+,MACL	×	×	O	L
LDS	@Rn+,PR	×	×	O	L
NOP	(no operands)	×	×	×	—
RTE	(no operands)	×	×	×	—
SETT	(no operands)	×	×	×	—
SLEEP	(no operands)	×	×	×	—
STC	SR,Rn	×	×	O	L
STC	GBR,Rn	×	×	O	L
STC	VBR,Rn	×	×	O	L
STC	SR,@–Rn	×	×	O	L
STC	GBR,@–Rn	×	×	O	L
STC	VBR,@–Rn	×	×	O	L
STS	MACH,Rn	×	×	O	L
STS	MACL,Rn	×	×	O	L
STS	PR,Rn	×	×	O	L
STS	MACH,@–Rn	×	×	O	L
STS	MACL,@–Rn	×	×	O	L
STS	PR,@–Rn	×	×	O	L
TRAPA	#imm	×	×	O	L

(b) SH-2 Executable Instruction and Operation Size Combinations:
Table 11.12 lists the combination of executable instructions added to SH-2 from SH-1 and the operation size.

Table 11.12 SH-2 Executable Instruction and Operation Size Combinations (Part 1)

Arithmetic Operation Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
MAC	@Rn+,@Rm+	×	O	O	W
MUL	Rn,Rm	×	×	O	L
DMULS	Rn,Rm	×	×	O	L
DMULU	Rn,Rm	×	×	O	L
DT	Rn	×	×	×	—

Table 11.12 SH-2 Executable Instruction and Operation Size Combinations (Part 2)

Branch Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
BF/S	symbol	×	×	×	—
BT/S	symbol	×	×	×	—
BRAF	Rn	×	×	×	—
BSRF	Rn	×	×	×	—

(c) SH-2E Executable Instruction and Operation Size Combinations:
Table 11.13 lists the combination of executable instructions added to SH-2E from SH-2 and the operation size.
Symbol meanings:

FRm,FRn	Floating-point register
FR0	FR0 floating-point register
FPUL	FPU communication register
FPSCR	FPU status control register
S	Single precision (4 bytes)

Table 11.13 SH-2E Executable Instruction and Operation Size Combinations (Part 1)

Data Move Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FLDI0	FRn	×	×	×	O	S
FLDI1	FRn	×	×	×	O	S
FMOV	@Rm,FRn	×	×	×	O	S
FMOV	FRn,@Rm	×	×	×	O	S
FMOV	@Rm+,FRn	×	×	×	O	S
FMOV	FRn,@-Rm	×	×	×	O	S
FMOV	@(R0,Rm),FRn	×	×	×	O	S
FMOV	FRm,@(R0,Rm)	×	×	×	O	S
FMOV	FRm,FRn	×	×	×	O	S

Table 11.13 SH-2E Executable Instruction and Operation Size Combinations (Part 2)

Arithmetic Operation Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FABS	FRn	×	×	×	O	S
FADD	FRm,FRn	×	×	×	O	S
FCMP/EQ	FRm,FRn	×	×	×	O	S
FCMP/GT	FRm,FRn	×	×	×	O	S
FDIV	FRm,FRn	×	×	×	O	S
FMAC	FR0,FRm,FRn	×	×	×	O	S
FMUL	FRm,FRn	×	×	×	O	S
FNEG	FRn	×	×	×	O	S
FSUB	FRm,FRn	×	×	×	O	S

Table 11.13 SH-2E Executable Instruction and Operation Size Combinations (Part 3)

System Control Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FLDS	FRm,FPUL	×	×	×	O	S
FLOAT	FPUL,FRn	×	×	×	O	S
FSTS	FPUL,FRn	×	×	×	O	S
FTRC	FRm,FPUL	×	×	×	O	S
LDS	Rm,FPUL	×	×	O	×	L
LDS	@Rm+,FPUL	×	×	O	×	L
LDS	Rm,FPSCR	×	×	O	×	L
LDS	@Rm+,FPSCR	×	×	O	×	L
STS	FPUL,Rn	×	×	O	×	L
STS	FPUL,@-Rn	×	×	O	×	L
STS	FPSCR,Rn	×	×	O	×	L
STS	FPSCR,@-Rn	×	×	O	×	L

(d) SH-3 Executable Instruction and Operation Size Combinations:

Table 11.14 lists the combination of executable instructions added to SH-3 from SH-2 and the operation size.

Symbol meanings:

Rn_BANK	Bank general register
SSR	Saved status register
SPC	Saved program counter

Table 11.14 SH-3 Executable Instruction and Operation Size Combinations (Part 1)

Data Move Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
PREF	@Rn	×	×	×	—

Table 11.14 SH-3 Executable Instruction and Operation Size Combinations (Part 2)

Shift Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
SHAD	Rn,Rm	×	×	O	L
SHLD	Rn,Rm	×	×	O	L

Table 11.14 SH-3 Executable Instruction and Operation Size Combinations (Part 3)

System Control Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
CLRS	(no operands)	×	×	×	—
SETS	(no operands)	×	×	×	—
LDC	Rm,SSR	×	×	O	L
LDC	Rm,SPC	×	×	O	L
LDC	Rm,Rn_BANK	×	×	O	L
LDC	@Rm+,SSR	×	×	O	L
LDC	@Rm+,SPC	×	×	O	L
LDC	@Rm+,Rn_BANK	×	×	O	L
STC	SSR,Rn	×	×	O	L
STC	SPC,Rn	×	×	O	L
STC	Rm_BANK,Rn	×	×	O	L
STC	SSR,@–Rn	×	×	O	L
STC	SPC,@–Rn	×	×	O	L
STC	Rm_BANK,@–Rn	×	×	O	L
LDTLB	(no operands)	×	×	×	—

(e) SH-3E Executable Instruction and Operation Size Combinations:

Table 11.15 lists the combination of executable instructions added to SH-3E from SH-3 and the operation size.

Symbol meanings:

FRm,FRn Floating-point register

FR0 FR0 floating-point register (when only FR0 can be specified)

FPUL FPU communication register

FPSCR FPU status control register

S Single precision (4 bytes)

Table 11.15 SH-3E Executable Instruction and Operation Size Combinations (Part 1)

Data Move Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FLDI0	FRn	×	×	×	O	S
FLDI1	FRn	×	×	×	O	S
FMOV	@Rm,FRn	×	×	×	O	S
FMOV	FRm,@Rn	×	×	×	O	S
FMOV	@Rm+,FRn	×	×	×	O	S
FMOV	FRm,@–Rn	×	×	×	O	S
FMOV	@(R0,Rm),FRn	×	×	×	O	S
FMOV	FRn,@(R0,Rn)	×	×	×	O	S
FMOV	FRm,FRn	×	×	×	O	S

Table 11.15 SH-3E Executable Instruction and Operation Size Combinations (Part 2)

Arithmetic Operation Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FABS	FRn	×	×	×	O	S
FADD	FRm,FRn	×	×	×	O	S
FCMP/EQ	FRm,FRn	×	×	×	O	S
FCMP/GT	FRm,FRn	×	×	×	O	S
FDIV	FRm,FRn	×	×	×	O	S
FMAC	FR0,FRm,FRn	×	×	×	O	S
FMUL	FRm,FRn	×	×	×	O	S
FNEG	FRn	×	×	×	O	S
FSQRT	FRn	×	×	×	O	S
FSUB	FRm,FRn	×	×	×	O	S

Table 11.15 SH-3E Executable Instruction and Operation Size Combinations (Part 3)

System Control Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**Default when Omitted**
FLDS	FRm,FPUL	×	×	×	O	S
FLOAT	FPUL,FRn	×	×	×	O	S
FSTS	FPUL,FRn	×	×	×	O	S
FTRC	FRm,FPUL	×	×	×	O	S
LDS	Rm,FPUL	×	×	O	×	L
LDS	@Rm+,FPUL	×	×	O	×	L
LDS	Rm,FPSCR	×	×	O	×	L
LDS	@Rm+,FPSCR	×	×	O	×	L
STS	FPUL,Rn	×	×	O	×	L
STS	FPUL,@–Rn	×	×	O	×	L
STS	FPSCR,Rn	×	×	O	×	L
STS	FPSCR,@–Rn	×	×	O	×	L

(f) SH-4 Executable Instruction and Operation Size Combinations:
Table 11.16 lists the combination of executable instructions added to SH-4 from SH-3E and the operation size.
Symbol meanings:
DRm,DRn Double-precision floating-point register

XDm,XDn	Extended double-precision floating-point register
FVm,FVn	Single-precision floating-point vector register
XMTRX	Single-precision floating-point extended register matrix
DBR	Debug vector base register
SGR	Save general register 15
D	Double precision (8 bytes)

Table 11.16 SH-4 Executable Instruction and Operation Size Combinations (Part 1)

Data Move Instructions		Operation Sizes					
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**D**	**Default when Omitted**
FMOV	DRm,DRn	×	×	×	×	O	D
FMOV	DRm,@Rn	×	×	×	×	O	D
FMOV	DRm,@-Rn	×	×	×	×	O	D
FMOV	DRm,@(R0,Rn)	×	×	×	×	O	D
FMOV	@Rm,DRn	×	×	×	×	O	D
FMOV	@Rm+,DRn	×	×	×	×	O	D
FMOV	@(R0,Rm),DRn	×	×	×	×	O	D
FMOV	DRm,XDn	×	×	×	×	O	D
FMOV	XDm,DRn	×	×	×	×	O	D
FMOV	XDm,XDn	×	×	×	×	O	D
FMOV	XDm,@Rn	×	×	×	×	O	D
FMOV	XDm,@-Rn	×	×	×	×	O	D
FMOV	XDm,@(R0,Rn)	×	×	×	×	O	D
FMOV	@Rm,XDn	×	×	×	×	O	D
FMOV	@Rm+,XDn	×	×	×	×	O	D
FMOV	@(R0,Rm),XDn	×	×	×	×	O	D

Table 11.16 SH-4 Executable Instruction and Operation Size Combinations (Part 2)

Arithmetic Operation Instructions		Operation Sizes					
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**D**	**Default when Omitted**
FABS	DRn	×	×	×	×	O	D
FADD	DRm,DRn	×	×	×	×	O	D
FCMP/EQ	DRm,DRn	×	×	×	×	O	D
FCMP/GT	DRm,DRn	×	×	×	×	O	D
FDIV	DRm,DRn	×	×	×	×	O	D
FIPR	FVm,FVn	×	×	×	O	×	S
FMUL	DRm,DRn	×	×	×	×	O	D
FNEG	DRn	×	×	×	×	O	D
FSQRT	DRn	×	×	×	×	O	D
FSUB	DRm,DRn	×	×	×	×	O	D
FTRV	XMTRX,FVn	×	×	×	O	×	S

Table 11.16 SH-4 Executable Instruction and Operation Size Combinations (Part 3)

System Control Instructions		Operation Sizes					
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**S**	**D**	**Default when Omitted**
FCNVDS	DRm,FPUL	×	×	×	×	O	D
FCNVSD	FPUL,DRn	×	×	×	×	O	D
FLOAT	FPUL,DRn	×	×	×	×	O	D
FRCHG	(no operands)	×	×	×	×	×	—
FSCHG	(no operands)	×	×	×	×	×	—
FTRC	DRm,FPUL	×	×	×	×	O	D
LDC	Rm,DBR	×	×	O	×	×	L
LDC	@Rm+,DBR	×	×	O	×	×	L
OCBI	@Rn	×	×	×	×	×	—
OCBP	@Rn	×	×	×	×	×	—
OCBWB	@Rn	×	×	×	×	×	—
STC	DBR,Rn	×	×	O	×	×	L
STC	DBR,@-Rn	×	×	O	×	×	L
STC	SGR,Rn	×	×	O	×	×	L
STC	SGR,@-Rn	×	×	O	×	×	L

(g) SH2-DSP, SH3-DSP Executable Instruction and Operation Size Combinations:

Table 11.17 shows the executable instruction and operation size combinations for the SH2-DSP and SH3-DSP instructions added to those of the SH-2 and SH-3, respectively.

Symbol meanings:

MOD	Modulo register
RS	Repeat start register
RE	Repeat end register
DSR	DSP status register
A0	DSP data register (A0, X0, X1, Y0, or Y1 can be specified.)

Table 11.17 SH2-DSP, SH3-DSP Executable Instruction and Operation Size Combinations (Part 1)

Repeat Control Instructions		**Operation Sizes**			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
LDRS	@(disp,PC)	×	×	O	L
LDRS	symbol	×	×	O	L
LDRE	@(disp,PC)	×	×	O	L
LDRE	symbol	×	×	O	L
SETRC	Rn	×	×	×	—
SETRC	#imm	×	×	×	—

Table 11.17 SH2-DSP, SH3-DSP Executable Instruction and Operation Size Combinations (Part 2)

System Control Instructions		Operation Sizes			
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**
LDC	Rn,MOD	×	×	O	L
LDC	Rn,RS	×	×	O	L
LDC	Rn,RE	×	×	O	L
LDC	@Rn+,MOD	×	×	O	L
LDC	@Rn+,RS	×	×	O	L
LDC	@Rn+,RE	×	×	O	L
LDS	Rn,DSR	×	×	O	L
LDS	Rn,A0	×	×	O	L
LDS	@Rn+,DSR	×	×	O	L
LDS	@Rn+,A0	×	×	O	L
STC	MOD,Rn	×	×	O	L
STC	RS,Rn	×	×	O	L
STC	RE,Rn	×	×	O	L
STC	MOD,@–Rn	×	×	O	L
STC	RS,@–Rn	×	×	O	L
STC	RE,@–Rn	×	×	O	L
STS	DSR,Rn	×	×	O	L
STS	A0,Rn	×	×	O	L
STS	DSR,@–Rn	×	×	O	L
STS	A0,@–Rn	×	×	O	L

(2) Notes on Delayed Branch Instructions

The unconditional branch instructions are delayed branch instructions. The microprocessors execute the delay slot instruction (the instruction directly following a branch instruction in memory) before executing the delayed branch instruction.

If an instruction inappropriate for a delay slot is specified, the assembler issues error 150 or 151.

Table 11.18 shows the relationship between the delayed branch instructions and the delay slot instructions.

Table 11.18 Relationship between Delayed Branch Instructions and Delay Slot Instructions

		Delayed Branch									
Delay Slot		**BF/S**	**BT/S**	**BRAF**	**BSRF**	**BRA**	**BSR**	**JMP**	**JSR**	**RTS**	**RTE**
BF		×	×	×	×	×	×	×	×	×	×
BT		×	×	×	×	×	×	×	×	×	×
BF/S		×	×	×	×	×	×	×	×	×	×
BT/S		×	×	×	×	×	×	×	×	×	×
BRAF		×	×	×	×	×	×	×	×	×	×
BSRF		×	×	×	×	×	×	×	×	×	×
BRA		×	×	×	×	×	×	×	×	×	×
BSR		×	×	×	×	×	×	×	×	×	×
JMP		×	×	×	×	×	×	×	×	×	×
JSR		×	×	×	×	×	×	×	×	×	×
RTS		×	×	×	×	×	×	×	×	×	×
RTE		×	×	×	×	×	×	×	×	×	×
TRAPA		×	×	×	×	×	×	×	×	×	×
LDC	Rn,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
	@Rn+,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
MOV	@(disp,PC),Rn	×	×	×	×	×	×	×	×	×	×
	symbol,Rn	×	×	×	×	×	×	×	×	×	×
MOVA	@(disp,PC),R0	×	×	×	×	×	×	×	×	×	×
	symbol,R0	×	×	×	×	×	×	×	×	×	×
Extended	MOV.L #imm,Rn	×	×	×	×	×	×	×	×	×	×
instructions	MOV.W #imm,Rn	×	×	×	×	×	×	×	×	×	×
	MOVA #imm,R0	×	×	×	×	×	×	×	×	×	×
Any other instruction		O	O	O	O	O	O	O	O	O	O

Symbol meanings:

O Normal, i.e., the assembler generates the specified object code.

× Error 150 or 151
The instruction specified is inappropriate as a delay slot instruction.
The assembler generates object code with a NOP instruction (H'0009).

Notes: 1. Operates normally when the CPU type is SH-1, SH-2, SH-2E, or SH2-DSP.
Any other CPU type will cause error 150 or 151 to occur.

Note: If the delayed branch instruction and the delay slot instruction are coded in different sections, the assembler does not check the validity of the delay slot instruction.

(3) Notes on Address Calculations

When the operand addressing mode is PC relative with displacement, i.e., @(disp,PC), the value of PC must be taken into account in coding. The value of PC can vary depending on certain conditions.

(a) Normal Case

The value of PC is the first address in the currently executing instruction plus 4 bytes.

Example:

(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

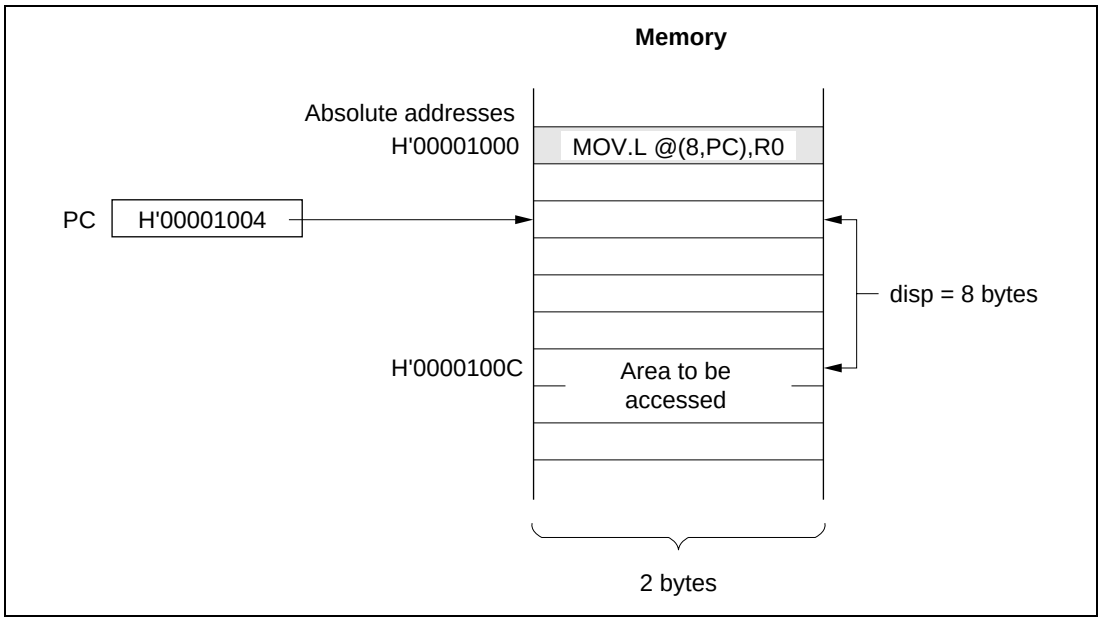


Figure 11.1 Address Calculation Example (Normal Case)

(b) During the Delay Slot Instruction

The value of PC is destination address for the delayed branch instruction plus 2 bytes.

Example:

(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

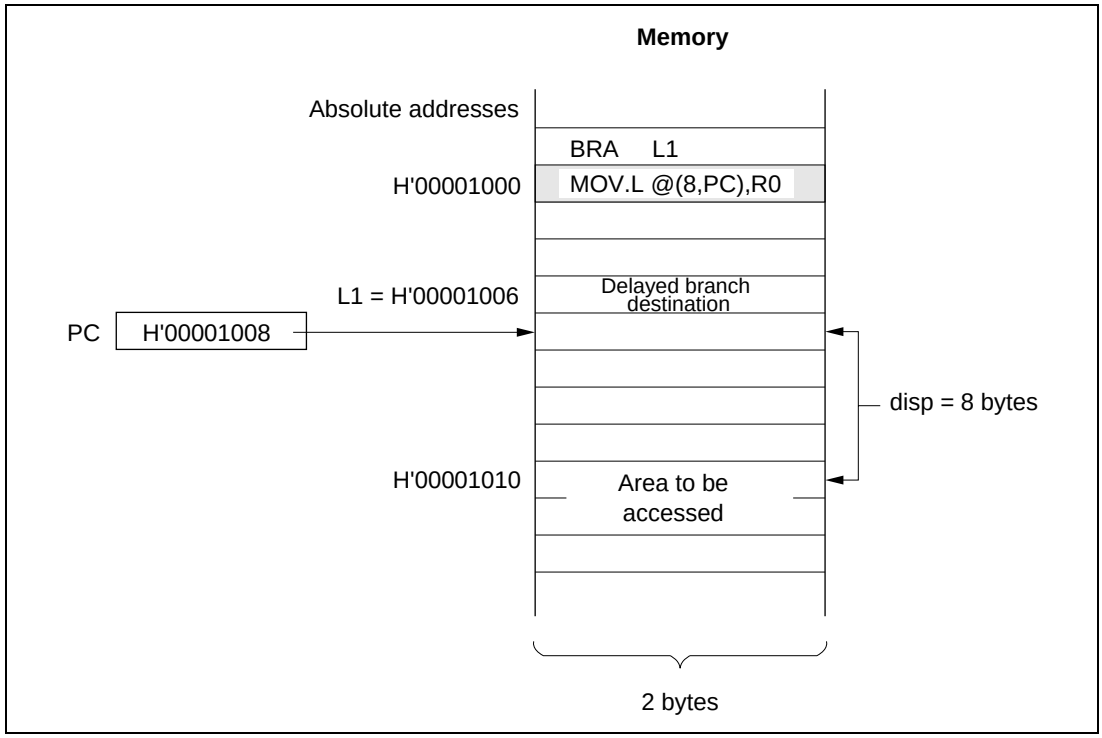


Figure 11.2 Address Calculation Example (When the Value of PC Differs Due to a Branch)

Supplement: When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

HITACHI

(c) During the Execution of Either a MOV.L @(disp,PC),Rn or a MOVA @(disp,PC),R0

When the value of PC is not a multiple of 4, microprocessors correct the value by discarding the lower 2 bits when calculating addresses.

Example 1:

When the microprocessor corrects the value of PC
(Consider the state when a MOV instruction is being executed at address H'00001002.)

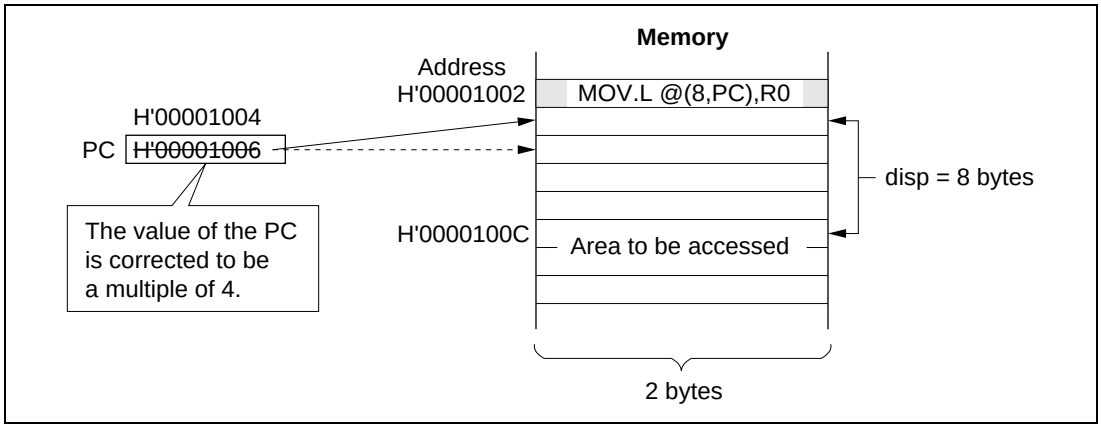


Figure 11.3 Address Calculation Example (When Microprocessor Corrects the Value of PC)

Example 2:

When the microprocessor does not correct the value of PC
(Consider the state when a MOV instruction is being executed at address H'00001000.)

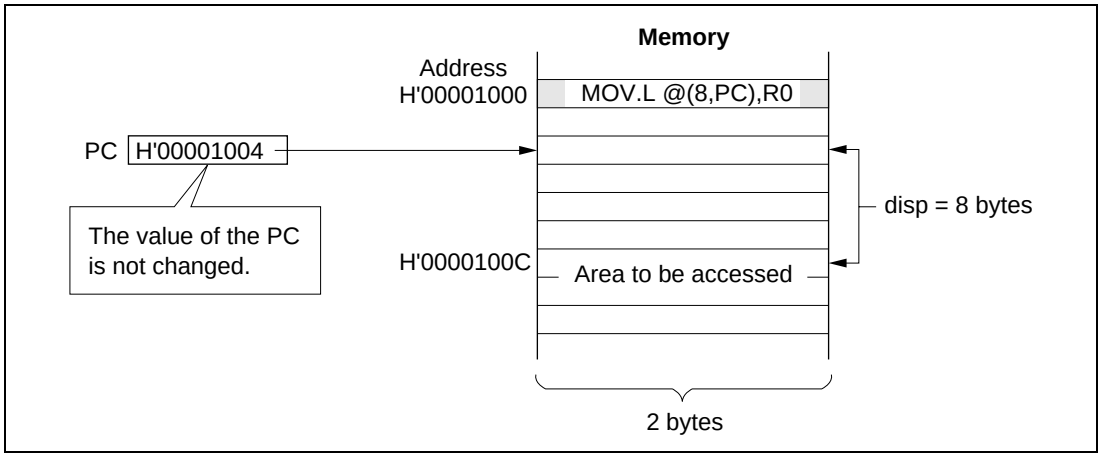


Figure 11.4 Address Calculation Example (When Microprocessor Does Not Correct the Value of PC)

Supplement: When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

11.3 DSP Instructions

11.3.1 Program Contents

(1) Source Statements

The SH2-DSP and SH3-DSP instructions are classified into two types: executable instructions and DSP instructions. The DSP instructions manipulate DSP registers. The instruction set and description format of DSP instructions are different from those of the executable instructions. For the DSP instructions, many operations can be included in one statement. The DSP instruction operation is as follows:

— DSP operation:

Specifies operations between DSP registers.

PABS, PADD, PADDC, PAND, PCLR, PCMP, PCOPY, PDEC, PDMSB, PINC, PLDS, PMULS, PNEG, POR, PRND, PSHA, PSHL, PSTS, PSUB, PSUBC, PXOR

— X data transfer operation:

Specifies data transfer between a DSP register and X data memory.

MOVX, NOPX

— Y data transfer operation:

Specifies data transfer between a DSP register and Y data memory.

MOVY, NOPY

— Single data transfer operation:

Specifies data transfer between a DSP register and memory.

MOVS

(2) Parallel Operation Instructions

Parallel operation instructions specify DSP operations as well as data transfer between a DSP register and X or Y data memory at the same time. The instruction size is 32 bits. The description format is as follows:

[<label>][Δ<DSP operation part>][Δ<data transfer part>][<comment>]

(a) DSP Operation Part Description Format:

[<condition>Δ]<DSP operation>Δ<operand>[Δ...]

- Condition:
  Specifies how parallel operation instruction is executed as follows:
  DCT: The instruction is executed when the DC bit is 1.
  DCF: The instruction is executed when the DC bit is 0.
- DSP operation:
  Specifies DSP operation.
  PADD and PMULS, and PSUB and PMULS can be specified in combination.

(b) Data Transfer Part Description Format:

[<X data transfer operation>[Δ<operand>]] [Δ<Y data transfer operation>[Δ<operand>]]

Be sure to specify X data transfer and Y data transfer in this order. Inputting an instruction is not required when the data move instruction is NOPX or NOPY.

Example:

```
LABEL1: PADD A0,M0,A0 PMULS X0,Y0,M0  MOVX.W @R4+,X0 MOVY.W @R6+,Y0  ;DSP Instruction
Label          DSP operation part              Data transfer part          Comment

        DCT PINC X1,A1  MOVX.W @R4,X0 MOVY.W @R6+, Y0
        DSP operation part        Data transfer part

        PCMP X1, M0    MOVX.W @R4, X0 ;Y Memory transfer is omitted
      DSP operation part  Data transfer part          Comment
```

(3) Data Transfer Instructions

Two types of data move instructions are available: combination of X data memory transfer and Y data memory transfer, and single data transfer. The description formats are as follows:

(a) Combination of X Data Memory Transfer and Y Data Memory Transfer Instructions:

```
[<label>][Δ<X data transfer operation>[Δ<operand>]]
  [Δ<Y data transfer operation>[Δ<operand>]][<comment>]
```

Be sure to specify X data memory transfer and Y data memory transfer in this order. Inputting an instruction is not required when the data move instruction is NOPX or NOPY. Note that both X data memory and Y data memory cannot be omitted, unlike the parallel operation instruction.

Example:

```
LABEL2:  MOVX.W @R4,X0                    ;Data move instruction
                                          (Y data memory transfer is omitted)
         MOVX.W @R4,X0  MOVY.W @R6+, Y0
```

(b) Single Data Transfer Instruction:

```
[<label>][Δ<single data transfer operation>Δ<operand>][<comment>]
```

Specifies the MOVS instruction.

Example:

```
LABEL3:  MOVS.W @-R2,A0      ;Single data transfer
```

(4) Coding of Source Statements Across Multiple Lines

For the DSP instructions, many operations can be included in one statement, and therefore, source statements become long and complicated. To make programs easy to read, source statements for DSP instructions can be written across multiple lines by separating between an operand and an operation, in addition to separating at a comma between operands.

Write source statements across multiple lines using the following procedure.

— Insert a new line between an operand and an operation.

— Insert a plus sign (+) in the first column of the new line.

— Continue writing the source statement following the plus sign.

Spaces and tabs can be inserted following the plus sign.

Example:

```
                PADD        A0,M0,X0
+               PMULS       A1,Y1,M0
+               MOVX        @R4,X0
+               MOVY        @R6,Y1
; A single source statement is written across four lines.
```

11.3.2 DSP Instructions

(1) DSP Operation Instructions

Table 11.19 lists DSP instructions in mnemonic.

Table 11.19 DSP Instructions in Mnemonic

Specifications	Mnemonic
DSP arithmetic operation instructions	PADD, PSUB, PCOPY, PDMSB, PINC, PNEG, PMULS, PADDC, PSUBC, PCMP, PDEC, PABS, PRND, PCLR, PLDS, PSTS
DSP logic operation instructions	POR, PAND, PXOR
DSP shift operation instructions	PSHA, PSHL

(a) Operation Size:

For the DSP operation instructions, operation size cannot be specified.

(b) Addressing Mode:

Table 11.20 lists addressing modes for the DSP operation instructions.

Table 11.20 Addressing Modes for DSP Operation Instructions

Addressing Mode	Description Format
DSP register direct	Dp (DSP register name)
Immediate data	#imm

- DSP register direct

  Table 11.21 lists registers that can be specified in DSP register direct addressing mode. For Sx, Sy, Dz, Du, Se, Sf, and Dg, refer to table 11.23, DSP Operation Instructions.

Table 11.21 Registers that Can Be Specified in DSP Register Direct Addressing Mode

		DSP Register							
		A0	**A1**	**M0**	**M1**	**X0**	**X1**	**Y0**	**Y1**
Dp	Sx	Yes	Yes			Yes	Yes		
	Sy			Yes	Yes			Yes	Yes
	Dz	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Du	Yes	Yes			Yes		Yes	
	Se		Yes			Yes	Yes	Yes	
	Sf		Yes			Yes		Yes	Yes
	Dg	Yes	Yes	Yes	Yes				

- Immediate data

  Immediate data can be specified for the first operand of the PSHA and PSHL instructions. The following items can be specified:

  — Value type

    Constants, symbols, or expressions can be specified.

  — Symbol types

    Symbols including relative and externally defined symbols can be specified as immediate data.

  — Value range

    Table 11.22 lists the specifiable value ranges.

Table 11.22 Ranges of Immediate Data

Immediate Data	Range
PSHA	H'FFFFFFE0 to H'00000020 (–32 to 32)
PSHL	H'FFFFFFF0 to H'00000010 (–16 to 16)

(c) Combination of Multiple DSP Operation Instructions:

The PADD instruction and the PMULS instruction, or the PSUB instruction and the PMULS instruction can be specified in combination. Each of these two combinations is basically one DSP instruction. The PADD (or PSUB) operand and a PMULS operand are separately described so that programs can be read easily.

Example:

```
PADD A0,M0,A0  PMULS X0,Y0,M0  NOPX  MOVY.W @R6+, Y0
PSUB A1,M1,A1  PMULS X1,Y1,M1  MOVX @R4+,X0  NOPY
```

Note: Warning 701 is displayed if the same register is specified as the destination registers when multiple DSP operation instructions are specified in combination.

Example:

PADD A0,M0,A0 PMULS X0,Y0,A0 → Warning 701

(d) Conditional DSP Operation Instructions:

Conditional DSP operation instructions specify if the program is executed according to the DC bit of the DSR register.
DCT: When the DC bit is 1, the instruction is executed.
DCF: When the DC bit is 0, the instruction is executed.
Conditional DSP operation instructions are the following:

PADD, PAND, PCLR, PCOPY, PDEC, PDMSB, PINC, PLDS, PNEG, POR, PSHA, PSHL, PSTS, PSUB, PXOR

(e) DSP Operation Instruction List:

Table 11.23 lists DSP operation instructions. For the registers that can be specified as Sx, Sy, Dz, Du, Se, Sf, and Dg, refer to table 11.21, Registers that Can Be Specified in DSP Register Direct Addressing Mode.

Table 11.23 DSP Operation Instructions

Mnemonic	Addressing Mode	Mnemonic	Addressing Mode
PABS	Sx, Dz		
PABS	Sy, Dz		
PADD	Sx, Sy, Dz		
PADD	Sx, Sy, Du	PMULS	Se, Sf, Dg
PADDC	Sx, Sy, Dz		
PAND	Sx, Sy, Dz		
PCLR	Dz		
PCMP	Sx, Sy		
PCOPY	Sx, Dz		
PCOPY	Sy, Dz		
PDEC	Sx, Dz		
PDEC	Sy, Dz		
PDMSB	Sx, Dz		
PDMSB	Sy, Dz		
PINC	Sx, Dz		
PINC	Sy, Dz		
PLDS	Dz, MACH		
PLDS	Dz, MACL		
PMULS	Se, Sf, Dg		
PNEG	Sx, Dz		
PNEG	Sy, Dz		
POR	Sx, Sy, Dz		
PRND	Sx, Dz		
PRND	Sy, Dz		
PSHA	#imm, Dz		
PSHA	Sx, Sy, Dz		
PSHL	#imm, Dz		
PSHL	Sx, Sy, Dz		
PSTS	MACH, Dz		

Table 11.23 DSP Operation Instructions (cont)

Mnemonic	Addressing Mode	Mnemonic	Addressing Mode
PSTS	MACL, Dz		
PSUB	Sx, Sy, Dz		
PSUB	Sx, Sy, Du	PMULS	Se, Sf, Dg
PSUBC	Sx, Sy, Dz		
PXOR	Sx, Sy, Dz		

(2) Data Transfer Instructions

(a) Mnemonics:

Two types of data transfer instructions are available: dual memory transfer instructions and single memory transfer instructions.

Dual memory transfer instructions specify data transfer, at the same time, between X memory and a DSP register, and between Y memory and a DSP register.

Single memory transfer instructions specify data transfer between arbitrary memory and a DSP register. Table 11.24 lists data transfer instructions in mnemonic.

Table 11.24 Data Transfer Instructions in Mnemonic

Classification		Mnemonic
Dual memory transfer	X memory transfer	NOPX MOVX
	Y memory transfer	NOPY MOVY
Single memory transfer		MOVS

(b) Operation Size:

NOPX and NOPY instructions:	Operation size cannot be specified.
MOVX and MOVY instructions:	Only word size (.W) can be specified. If omitted, word size is assumed.
MOVS instruction:	Word size (.W) or longword size (.L) can be specified. If omitted, longword size is assumed.

(c) Addressing Mode:

Table 11.25 lists addressing modes that can be specified for the data transfer instructions.

Table 11.25 Addressing Modes of Data Transfer Instructions

Addressing mode	Description
DSP register direct	Dz
Register indirect	@Az
Register indirect with post-increment	@Az+
Register indirect with index/post-increment	@Az+Iz
Register indirect with pre-decrement	@-Az

Register indirect with index/post-increment is a special addressing mode for the DSP data transfer instructions. In this mode, after referring to the contents indicated by register Az, register Az contents are incremented by the value of the Iz register.

(d) Registers that Can Be Specified in Addressing Modes:

Table 11.26 lists registers that can be specified in the DSP register direct, register indirect, register indirect with post-increment, register indirect with index/post-increment, and register indirect with pre-decrement addressing modes. For Dx, Dy, Ds, Da, Ax, Ay, As, Ix, Iy, and Is, refer to table 11.27, Data Transfer Instructions.

Table 11.26 Registers that Can Be Specified in Addressing Modes for Data Transfer Instructions

		General Registers								DSP Registers									
		R2	R3	R4	R5	R6	R7	R8	R9	A0	A1	M0	M1	X0	X1	Y0	Y1	A0G	A1G
Dz	Dx													Yes	Yes				
	Dy															Yes	Yes		
	Ds									Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Da									Yes	Yes								
Az	Ax			Yes	Yes														
	Ay					Yes	Yes												
	As	Yes	Yes	Yes	Yes														
Iz	Ix							Yes											
	Iy								Yes										
	Is							Yes											

Note: Warning 703 is displayed if the destination register for the DSP instruction and the destination register for the data transfer instruction are the same register in the same statement.

Example:

PADD A0,M0,Y0 NOPX MOVY.W @R6+,Y0 → Warning 703

(e) Data Transfer Instruction List:

Table 11.27 lists data transfer instructions. For registers that can be specified for Dx, Dy, Ds, Da, Ax, Ay, As, Ix, Iy, and Is, refer to table 11.26, Registers that Can Be Specified in Addressing Modes for Data Transfer Instructions.

Table 11.27 Data Transfer Instructions

Classification	Mnemonic	Addressing Mode
X data transfer instructions	NOPX	
	MOVX.W	@Ax, Dx
	MOVX.W	@Ax+, Dx
	MOVX.W	@Ax+Ix, Dx
	MOVX.W	Da, @Ax
	MOVX.W	Da, @Ax+
	MOVX.W	Da, @Ax+Ix
Y data transfer instructions	NOPY	
	MOVY.W	@Ay, Dy
	MOVY.W	@Ay+, Dy
	MOVY.W	@Ay+Iy, Dy
	MOVY.W	Da, @Ay
	MOVY.W	Da, @Ay+
	MOVY.W	Da, @Ay+Iy
Single data transfer instructions	MOVS.W	@-As, Ds
	MOVS.W	@As, Ds
	MOVS.W	@As+, Ds
	MOVS.W	@As+Is, Ds
	MOVS.W	Ds, @-As
	MOVS.W	Ds, @As
	MOVS.W	Ds, @As+
	MOVS.W	Ds, @As+Is
	MOVS.L	@-As, Ds
	MOVS.L	@As, Ds
	MOVS.L	@As+, Ds
	MOVS.L	@As+Is, Ds
	MOVS.L	Ds, @-As
	MOVS.L	Ds, @As
	MOVS.L	Ds, @As+
	MOVS.L	Ds, @As+Is

11.4 Assembler Directives

The assembler directives are instructions that the assembler interprets and executes. The underscores indicate the default. Table 11.28 lists the assembler directives provided by this assembler.

Table 11.28 Assembler Directives

Type	Mnemonic	Function
Target CPU	.CPU	Specifies the target CPU.
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.
	.FREG	Defines a floating-point register name.
Data and data area reservation	.DATA	Reserves integer data.
	.DATAB	Reserves an integer data block.
	.SDATA	Reserves string literal data.
	.SDATAB	Reserves an string literal data block.
	.SDATAC	Reserves string literal data (with length).
	.SDATAZ	Reserves string literal data (with zero terminator).
	.FDATA	Reserves floating-point data.
	.FDATAB	Reserves a floating-point data block.
	.XDATA	Reserves fixed-point data.
	.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).
	.FRES	Reserves floating-point data area.

Table 11.28 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.
Object modules	.OUTPUT	Controls object module and debugging information output.
	.DEBUG	Controls the output of symbolic debugging information.
	.ENDIAN	Selects big endian or little endian.
	.LINE	Changes line number.
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.

.CPU

Description Format: Δ.CPUΔ<target CPU>

The label field is not used.

Specification	Target CPU
SH1	Assembles program for SH-1
SH2	Assembles program for SH-2
SH2E	Assembles program for SH-2E
SH3	Assembles program for SH-3
SH3E	Assembles program for SH-3E
SH4	Assembles program for SH-4
SHDSP	Assembles program for SH2-DSP
SH3DSP	Assembles program for SH3-DSP

Description: .CPU specifies the target CPU for which the source program is assembled.

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 SHCPU environment variable.

If the directive is not specified, the CPU selected by the environment variable SHCPU becomes valid.

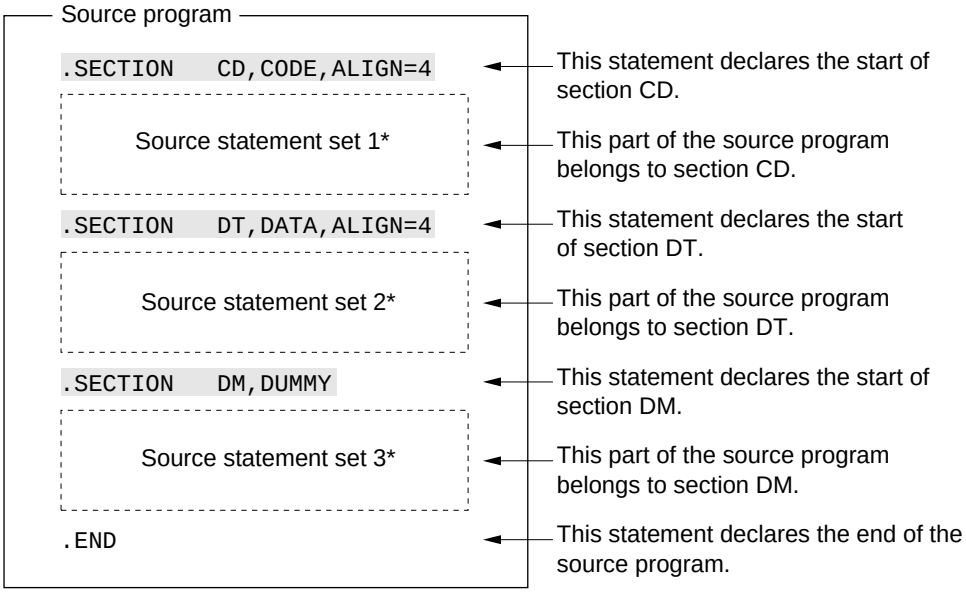
Example:
```
.CPU   SH2                    ; Assembles program for the SH-2.
.SECTION A,CODE,ALIGN=4
MOV.L R0,R1
MOV.L R0,R2
```

.SECTION

Description Format: Δ.SECTIONΔ<section name> [,<section attribute> [,<section type>]]

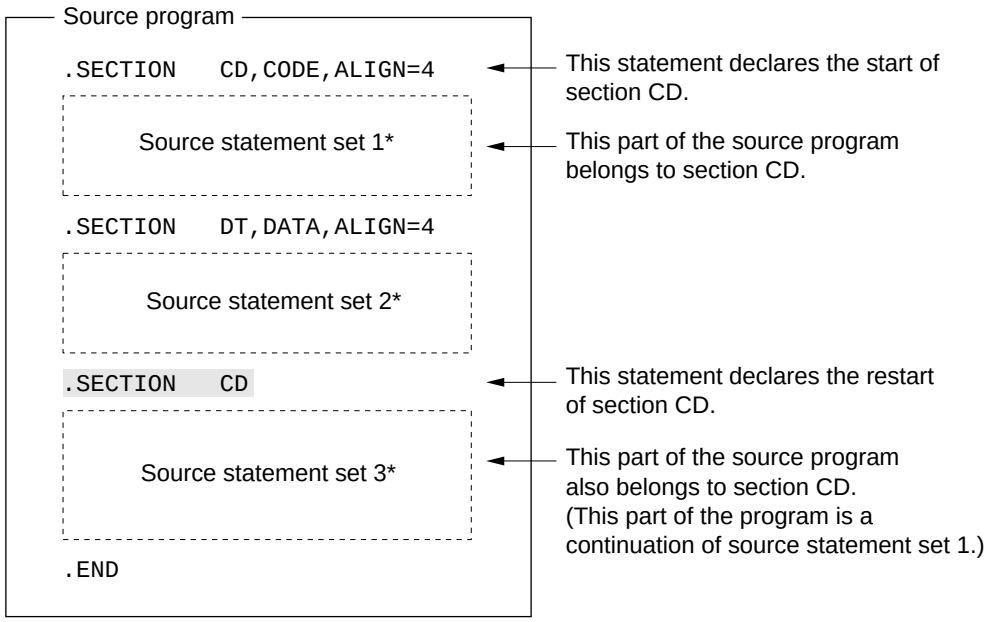
The label field is not used.
<section attribute>={ CODE | DATA | STACK | DUMMY }
<section type>={LOCATE= <start address>|
ALIGN=<boundary alignment value>}

Description: .SECTION is the section declaration assembler directive.
A section is a part of a program, and the optimizing linkage editor regards it as a unit of processing. 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 optimizing 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=4 is assumed.
The following describes section declaration using the simple examples shown below.



Note: This example assumes that the .SECTION directive does not appear in any of the source statement sets 1 to 3.

It is possible to redeclare (and thus restart,) a section that was previously declared in the same file. 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.

When using the .SECTION directive to restart a section, the second and third operands must be omitted. (The original specifications when first declaring the section remain valid.)

Use LOCATE = <start address> as the third operand when starting an absolute address section. The start address is the absolute address of the start of that section.

The start address 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 start address are from H'00000000 to H'FFFFFFFF. (From 0 to 4,294,967,295 in decimal.)
Use ALIGN = <boundary alignment value> to start a relative address section. The linkage editor will adjust the start address of the section to be the multiple of the boundary alignment value. The boundary alignment 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, e.g. 2^0, 2^1, 2^2, ..., 2^{31}. For code sections, the values must be 4 or larger powers of 2, e.g. 2^2, 2^3, 2^4, ..., 2^{31}.
The assembler provides a default section for the following cases:

- The use of executable, extended, or DSP 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.

The default section is the following section:

- Section name: P
- Section type: Code section
  Relative address section (with a boundary alignment value of 4)

Example:

```
          .ALIGN     4                ;This section of the program belongs to the default section P.
          .DATA.L    H'11111111       ;The default section P is a code section, and is a relative
            ~                         ;address section with a boundary alignment value of 4.

     .SECTION CD,CODE,ALIGN=4

           MOV       R0,R1            ;This section of the program belongs to the section CD.
           MOV       R0,R2            ;The section CD is a code section, and is a relative address
             ~                        ;section with a boundary alignment value of 4.

     .SECTION DT,DATA,LOCATE=H'00001000

 X1:       .DATA.L   H'22222222       ;This section of the program belongs to the section DT.
           .DATA.L   H'33333333       ;The section DT is a data section, and is an absolute address
             ~                        ;section with a start address of H00001000.

           .END
```

Note: This example assumes the .SECTION directive does not appear in the parts indicated by "~".

.ORG

Description Format: Δ.ORGΔ<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:

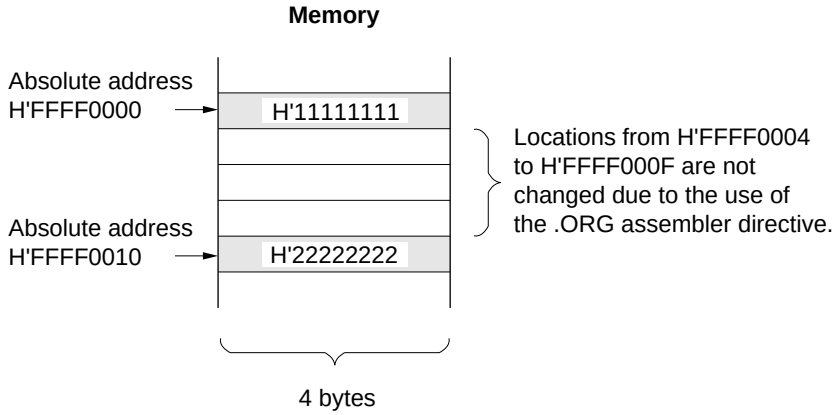
- The specification must be a constant value or an address within the section, and,
- Forward reference symbols must not appear in the specification.

The values allowed for the location-counter value are from H'00000000 to H'FFFFFFFF. (From 0 to 4,294,967,295 in decimal.)
When the location-counter value is specified with a constant value, the following condition must be satisfied:
<location-counter value> ≥ <section start address> (when compared as unsigned values)
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'FFFF0000
.DATA.L    H'11111111
.ORG       H'FFFF0010    ; This statement sets the value of the location
                         ; counter.
.DATA.L    H'22222222    ; The integer data H'22222222 is stored at
                         ; absolute address H'FFFF0010.
~
```

Explanatory Figure for the Coding Example



.ALIGN

Description Format: Δ.ALIGNΔ<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, e.g. 2^0, 2^1, 2^2, ..., 2^{31}.
When .ALIGN is used in a relative section the following must be satisfied:

Boundary alignment value specified by .SECTION ≥ 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'09.

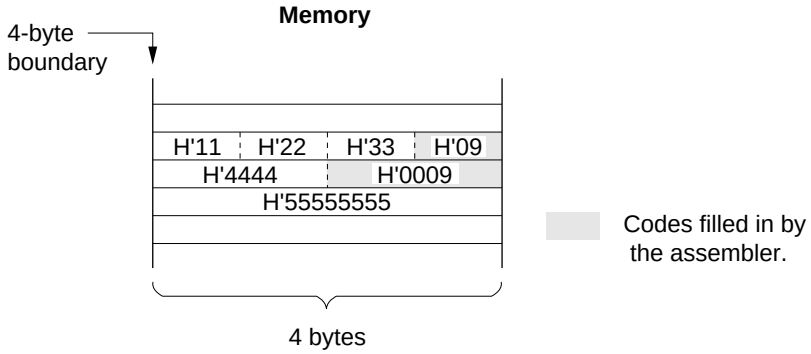
When .ALIGN is used in a data, dummy, or stack section, the assembler only adjusts the value of the location counter, and does not fill in any object code in memory.

Note: This object code is not displayed in the assemble listing.

Example:
```
.SECTION   P,CODE,ALIGN=4
.DATA.B    H'11
.DATA.B    H'22
.DATA.B    H'33

.ALIGN     2              ; This statement adjusts the value of the location
.DATA.W    H'4444         ; counter to be a multiple of 2.

.ALIGN     4              ; This statement adjusts the value of the location
.DATA.L    H'55555555     ; counter to be a multiple of 4.
~
```

Explanatory Figure for the Coding Example

This example assumes that the byte-sized integer data H'11 is originally located at the 4-byte boundary address. The assembler will insert the filler data as shown in the figure below.



HITACHI

.EQU

Description Format: <symbol>[:]Δ.EQUΔ<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'FFFFFFFF.
(From -2,147,483,648 to 4,294,967,295 in decimal.)

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/EQ   #X1,R0    ;This is the same as CMP/EQ #10,R0.
      BT       LABEL1
      CMP/EQ   #X2,R0    ;This is the same as CMP/EQ #20,R0.
      BT       LABEL2

      ~
```

.ASSIGN

Description Format: <symbol>[:]Δ.ASSIGNΔ<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'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)
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/EQ   #X1,R0      ;This is the same as CMP/EQ #1,R0.
        BT       LABEL1
        CMP/EQ   #X2,R0      ;This is the same as CMP/EQ #2,R0.
        BT       LABEL2

        ~

X1:     .ASSIGN  3
X2:     .ASSIGN  4
        CMP/EQ   #X1,R0      ; This is the same as CMP/EQ #3,R0.
        BT       LABEL3
        CMP/EQ   #X2,R0      ; This is the same as CMP/EQ #4,R0.
        BF       LABEL4
        ~
```

.REG

Description Format: <symbol>[:]Δ.REGΔ<register name>
or
<symbol>[:]Δ.REGΔ(<register name>)

Description: .REG defines the alias of a register name.
The alias of a register name defined with .REG can be used in exactly the same manner as the original register name.
The alias of a register name defined with .REG cannot be redefined.
The alias of a register name can only be defined for the general registers (R0 to R15, and SP).
Definitions with the .REG directive are valid from the point of the definition forward in the program.
Symbols defined with .REG have the following limitations:
They cannot be used as externally referenced or externally defined symbols.

Example:

```
~

MIN:  .REG      R10
MAX:  .REG      R11
      MOV       #0,MIN    ;This is the same as MOV #1,R10.
      MOV       #99,MAX   ;This is the same as MOV #99,R11.

      CMP/HS    MIN,R1
      BF        LABEL
      CMP/HS    R1,MAX
      BF        LABEL

~
```

.FREG

Description Format: <symbol>[:]Δ.FREGΔ<floating-point register name>
or
<symbol>[:]Δ.FREGΔ(<floating-point register name>)

Description: .FREG defines the alias of a floating-point register name.
The alias of a floating-point register name defined with .FREG can be used in exactly the same manner as the original register name.
The alias of a floating-point register name defined with .FREG cannot be redefined.
The alias can only be defined for the floating-point registers FRm (m = 0 to 15), DRn (n = 0, 2, 4, 6, 8, 10, 12, 14), XDn (n = 0, 2, 4, 6, 8, 10, 12, 14), and FVi (i = 0, 4, 8, 12).
Definitions with the .FREG are valid from the point of the definition forward in the program.
Symbols defined with .FREG have the following limitations:
They cannot be used as externally referenced or externally defined symbols.
.FREG is valid only when SH-2E, SH-3E, or SH-4 is selected as the CPU type.

Example:

```
        ~

MAX: .FREG     FR11
     FMOV      FR1,MAX   ; This is the same as
                         ; FMOV FR1,FR11.

     FCMP/EQ   MAX,FR2   ; This is the same as
                         ; FCMP/EQ FR11,FR2.
     BF        LABEL

        ~
```

.DATA

Description Format: [<symbol>[:]]Δ.[.<operation size>]Δ<integer data>[,...]

<operation size>: { B | W | L }

Description: .DATA reserves integer data in memory.
The specifier determines the size of the reserved data.
The longword 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 and the range of integer data are as follows:

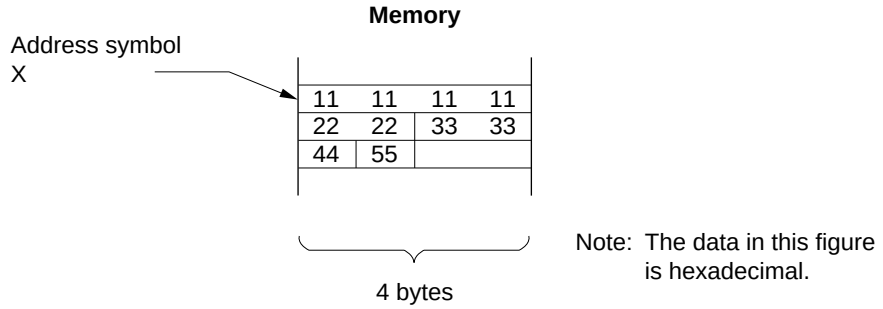
Operation Size	Integer Data Range*	
B (Byte)	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (–128 to –1)
W (Word, 2 bytes)	H'00000000 to H'0000FFFF H'FFFF8000 to H'FFFFFFFF	(0 to 65,535) (–32,768 to –1)
L (Longword, 4 bytes)	H'00000000 to H'7FFFFFFF H'80000000 to H'FFFFFFFF	(0 to 4,294,967,295) (–2,147,483,648 to –1)

Note: Numbers in parentheses are decimal.

Example:

```
~

        .ALIGN    4
X:      .DATA.L   H'11111111 ;
        .DATA.W   H'2222,H'3333; These statements reserve integer
                                ; data.
        .DATA.B   H'44,H'55     ;

~
```

Explanatory Figure for the Coding Example



HITACHI

.DATAB

Description Format: [<symbol>[:]]Δ.DATAB[.<operation range>]Δ<block count>,<integer data>

<operation size>: { B | W | L }

Description: .DATAB reserves the specified number of integer data items consecutively in memory.
The operation size determines the size of the reserved data.
The longword 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.

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	Block Size Range*	
B (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)

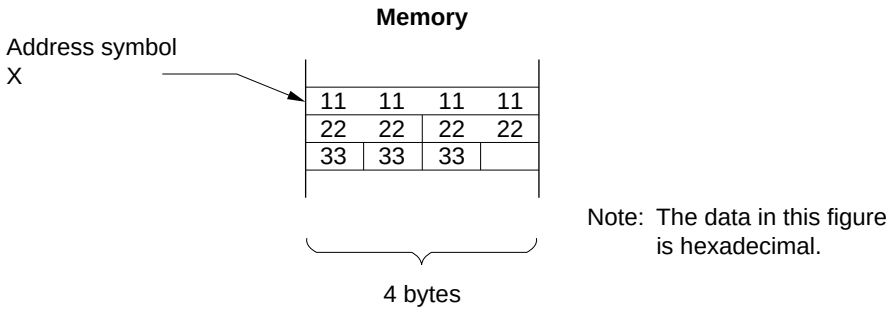
Operation Size	Integer Data Range*	
B	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (–128 to –1)
W	H'00000000 to H'0000FFFF H'FFFF8000 to H'FFFFFFFF	(0 to 65,535) (–32,768 to –1)
L	H'00000000 to H'7FFFFFFF H'80000000 to H'FFFFFFFF	(0 to 4,294,967,295) (–2,147,483,648 to –1)

Note: Numbers in parentheses are decimal.

Example:

```
        ~

        .ALIGN      4
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



.SDATA

Description Format: [<symbol>[:]]Δ.SDATAΔ"<string literal>"[,...]

Description: .SDATA reserves string literal data in memory.
A control character can be appended to a string literal.
The syntax for this notation is as follows:

"<string literal>"<ASCII code for a control character>

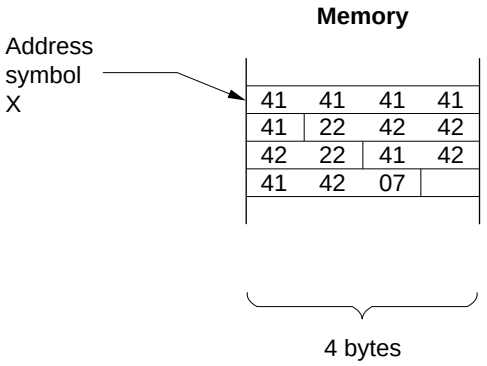
The ASCII code for a control character must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example:

```
~

        .ALIGN   4
X:      .SDATA   "AAAAA"        ; This statement reserves string
                                ; literal data.
        .SDATA   """BBB"""      ; The string literal in this example
                                ; includes double quotation marks.
        .SDATA   "ABAB"<H'07>   ; The string literal in this example
                                ; has a control character 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 string literals 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.)
A control character can be appended to a string literal.
The syntax for this notation is as follows:

"<string literal>"<ASCII code for a control character>

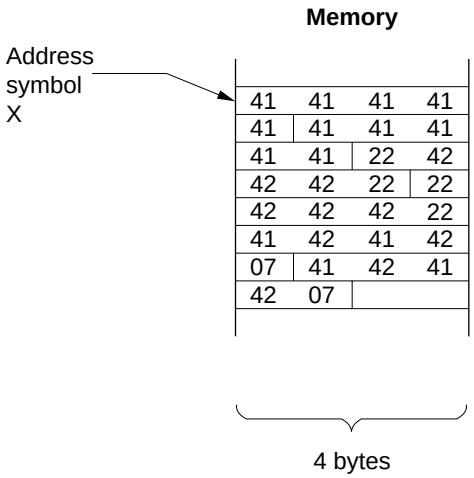
The ASCII code for a control character must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example:

```
        ~
        .ALIGN    4
X:
        .SDATAB   2,"AAAAA"          ; This statement reserves two
                                     ; string literal data blocks.
        .SDATAB   2,""""BBB""""      ; The string literal in this
                                     ; example includes double quotation
                                     ; marks.
        .SDATAB   2,"ABAB"<H'07>     ; The string literal in this
                                     ; example has a control character
                                     ; 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 with length is a string literal with an inserted leading byte that indicates the length of the string.
The length indicates the size of the string literal (not including the length) in bytes.
A control character can be appended to a string literal.
The syntax for this notation is as follows:

"<string literal>"<control code>

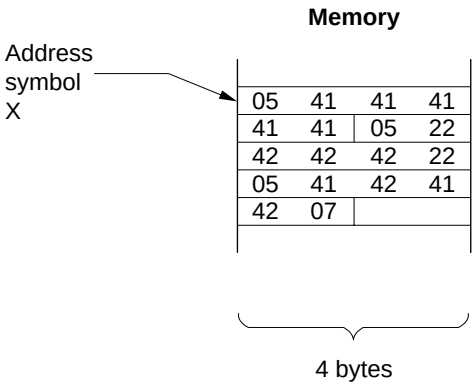
The ASCII code for a control character must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example:

```
~

        .ALIGN 4
X:
        .SDATAC     "AAAAA"  ; This statement reserves character
                             ; string data (with length).
        .SDATAC     """BBB"""  ; The string literal in this example
                             ; includes double quotation marks.
        .SDATAC   "ABAB"<H'07>; The string literal in this example
                             ; has a control character 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. A control character can be appended to a string literal. The syntax for this notation is as follows:

"<string literal>"<ASCII code for a control character>

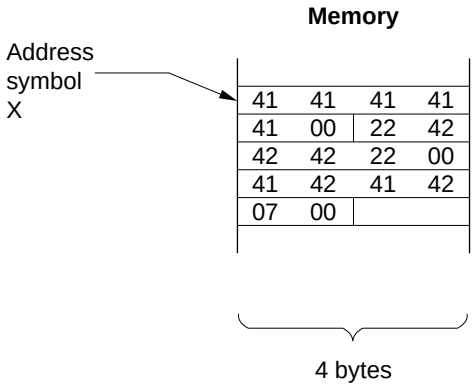
The ASCII code for a control character must be specified as follows:

- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

Example:

```
        ~
        .ALIGN    4
X:      .SDATAZ   "AAAAA"          ; This statement reserves character
                                   ; string data (with zero terminator).
        .SDATAZ   """BBB"""        ; The string literal in this example
                                   ; includes double quotation marks.
        .SDATAZ   "ABAB"<H'07>     ; The string literal in this example
                                   ; has a control character 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.

.FDATA

Description Format: [<symbol>[:]]Δ.FDATA[.<operation size>]Δ<floating-point data>[,...]

Operands: Enter the values to be reserved as data.

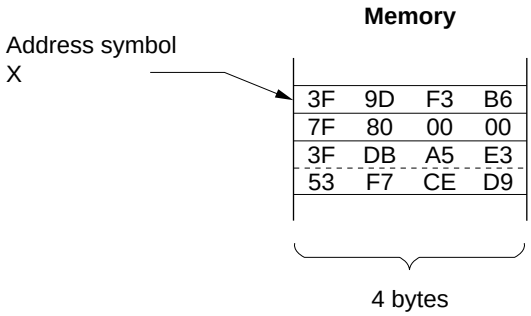
Description: .FDATA reserves floating-point data in memory.
The operation size determines the size of the reserved data.
Single precision is used when the operation size is omitted.
Operation size is as follows:

Operation Size	Data Size
S	Single precision (4 bytes)
D	Double precision (8 bytes)

Example:

```
        ~
        .ALIGN      4
X:
        .FDATA.S    F'1.234         ; This statement reserves a 4-byte
                                    ; area 3F9DF3B6 (F'1.234S).
        .FDATA.S    H'7F800000.S    ; This statement reserves a 4-byte
                                    ; area 7F800000 (H'7F800000.S).
        .FDATA.D    F'4.32D-1       ; This statement reserves an 8-byte
                                    ; area 3FDBA5E353F7CED9
                                    ; (F'4.32D-1).

        ~
```

Explanatory Figure for the Coding Example



.FDATAB

Description Format: [<symbol>[:]]Δ.FDATAB[.<operation size>]Δ<block count>,
<floating-point data>

Description: .FDATAB reserves the specified number of floating-point data items consecutively in memory.

The operation size determines the size of the reserved data.

Single precision 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, externally defined symbols, and relative symbols must not appear in specification.

The range of values that can be specified as the block count varies with the operation size.

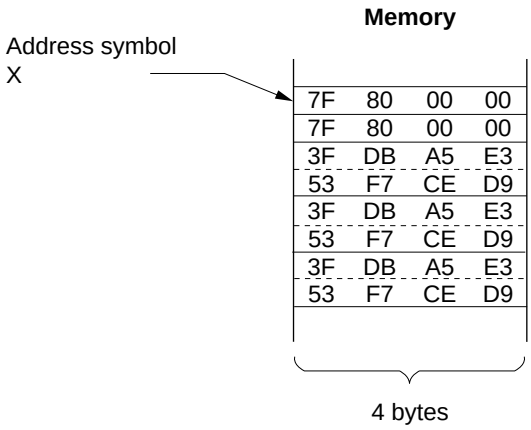
Operation Size	**Block Size Range***
S (single precision, 4 bytes)	H'00000001 to H'3FFFFFFF (1 to 1,073,741,823)
D (double precision, 8 bytes)	H'00000001 to H'1FFFFFFF (1 to 536,870,911)

Note: Numbers in parentheses are decimal.

Example:

```
        ~

        .ALIGN      4
X:
        .FDATAB.S   2,H'7F800000.S  ; This statement reserves two blocks
                                    ; of 4-byte areas 7F800000
                                    ; (H'7F800000.S).
        .FDATAB.D   3,F'4.32D-1     ; This statement reserves three
                                    ; blocks of 8-byte areas
                                    ; 3FDBA5E353F7CED9 (F'4.32D-1).

        ~
```

Explanatory Figure for the Coding Example



HITACHI

.XDATA

Description Format: [<symbol>[:]]Δ.XDATA[.<operation size>]Δ<fixed-point data>[,...]

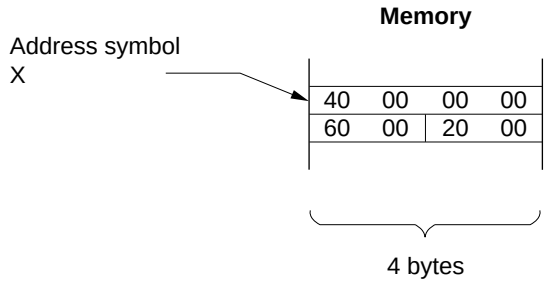
Description: .XDATA reserves fixed-point data in memory.
The operation size determines the size of the reserved data.
The longword size is used when the operation range is omitted.
The operation size is as follows:

Operation Size	Data Size
W	Word (2 bytes)
L	Longword (4 bytes)

Example:

```
        ~

        .ALIGN     4
X:
        .XDATA.L   0.5          ; This statement reserves 4-byte
                                ; area (H'40000000).
        .XDATA.W   0.75,0.25    ; This statement reserves 2-byte
                                ; areas (H'6000) and (H'2000).

        ~
```

Explanatory Figure for the Coding Example



.RES

Description Format: [<symbol>[:]]Δ.RES[.<operation size>]Δ<area count>

Description: .RES reserves data areas in memory.
The operation size determines the size of one area.
The longword size is used when the specifier 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.

The range of values that can be specified as the area count varies with the operation size.
The data size and the range of area count are as follows:

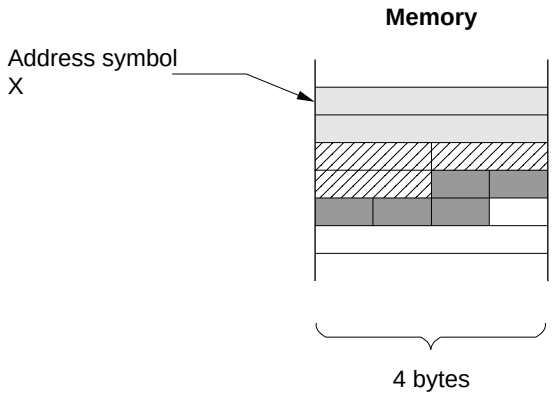
Operation Size	Area Count Range*	
B (byte)	H'00000001 to H'FFFFFFFF	(1 to 4,294,967,295)
W (word, 2 bytes)	H'00000001 to H'7FFFFFFF	(1 to 2,147,483,647)
L (longword, 4 bytes)	H'00000001 to H'3FFFFFFF	(1 to 1,073,741,823)

Note: Numbers in parentheses are decimal.

Example:

```
        ~
        .ALIGN  4
X:
        .RES.L  2   ; This statement reserves two-longword-size
                    ; areas.
        .RES.W  3   ; This statement reserves three-word-size
                    ; areas.
        .RES.B  5   ; This statement reserves five-byte-size
                    ; areas.

        ~
```

Explanatory Figure for the Coding Example



.SRES

Description Format: [<symbol>[:]]Δ.SRESΔ<string literal area size>[,...]

Description: .SRES reserves string literal data areas.
The string literal area size must be specified as follows:

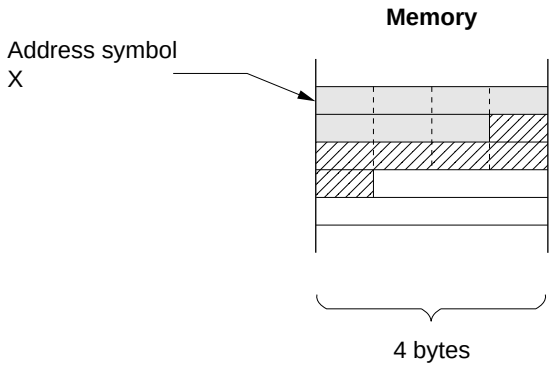
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

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:

```
        ~

        .ALIGN    4
X:
        .SRES     7         ; This statement reserves a 7-byte area.
        .SRES     6         ; This statement reserves a 6-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.
A string literal with length is a string literal with an inserted leading byte that indicates the length of the string.
The length indicates the size of the string literal (not including the length) in bytes.
The string literal area size must be specified as follows:

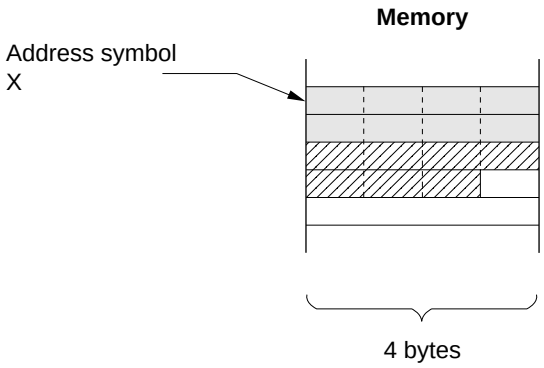
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

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:

```
        ~
        .ALIGN  4
X:
        .SRESC  7       ; This statement reserves 7 bytes plus 1 byte
                        ; for the count.
        .SRESC  6       ; This statement reserves 6 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).
A string literal with zero termination is a string literal with an appended trailing byte (with the value H'00) that indicates the end of the string.
The string literal area size must be specified as follows:

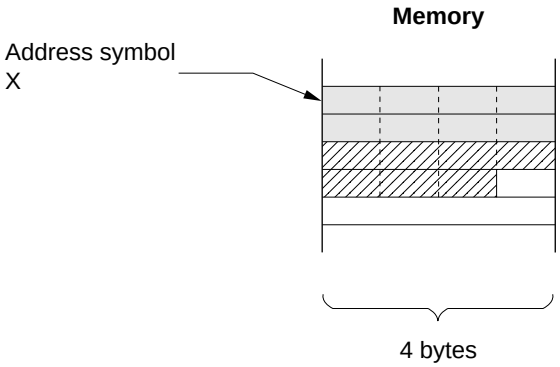
- The specification must be a constant value, and,
- Forward reference symbols must not appear in the specification.

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:

```
~

        .ALIGN  4
X:
        .SRESZ  7   ; This statement reserves 7 bytes plus 1 byte
                    ; for the terminating byte.
        .SRESZ  6   ; This statement reserves 6 bytes plus 1 byte
                    ; for the terminating byte.

~
```

Explanatory Figure for the Coding Example



.FRES

Description Format: [<symbol>[:]]Δ.FRES[.<operation size>]Δ<area count>

<operation size> = { S | D }

Description: .FRES reserves floating-point data areas in memory.
The operation size determines the size of the reserved data.
Single precision is used when the specifier is omitted.
The area count must be specified as follows:
The specification must be a constant value,

- The specification must be a constant value, and,
- Forward reference symbols, externally referenced symbols, and relative symbols must not appear in the specification.

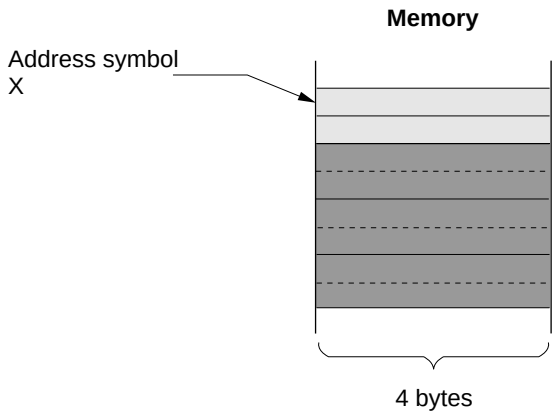
Operation size is as follows:

Operation Size	Data Size
S	Single precision (4 bytes)
D	Double precision (8 bytes)

Example:

```
        ~

        .ALIGN    4
X:
        .FRES.S   2       ; This statement reserves two areas.
        .FRES.D   3       ; This statement reserves three areas.

        ~
```

Explanatory Figure for the Coding Example



.EXPORT

Description Format: Δ.EXPORTΔ<symbol>[,...]

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.

- Constant symbols (other than those defined with the .ASSIGN directive)
- Absolute address symbols (other than address symbols in a dummy section)
- Relative address symbols

To reference a symbol as an externally referenced symbol, it is necessary to declare it to be an externally defined symbol, and also to declare it to be an externally referenced symbol.
Externally referenced symbols are declared in the file in which they are referenced using either the .IMPORT or the .GLOBAL directive.

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.
        ~

        .ALIGN      4
        .DATA.L     X       ; This statement references X.

        ~
```

.IMPORT

Description Format: Δ.IMPORTΔ<symbol>[,<symbol>...]

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 file.
Symbols defined in the current file cannot be declared to be externally referenced symbols.
To reference a symbol as an externally referenced symbol, it is necessary to declare it to be an externally referenced symbol, and also to declare it to be an externally defined symbol.
Externally defined symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL directive.

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.
        ~

        .ALIGN    4            ;
        .DATA.L   X            ; This statement references X.

        ~
```

.GLOBAL

Description Format: Δ.GLOBALΔ<symbol>[,<symbol>...]

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 current file from other files. An externally referenced symbol declaration is required to reference symbols defined in another file.
A symbol defined within the current file is declared to be an externally defined symbol by a .GLOBAL declaration.
A symbol that is not defined within the current file is declared to be an externally referenced symbol by a .GLOBAL declaration.
The following can be declared to be externally defined symbols.

- Constant symbols (other than those defined with the .ASSIGN assembler directive)
- Absolute address symbols (other than address symbols in a dummy section)
- Relative address symbols

To reference a symbol as an externally referenced symbol, it is necessary to declare it to be an externally defined symbol, and also to declare it to be an externally referenced symbol.
Externally defined symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL directive.
Externally referenced symbols are declared in the file in which they are referenced using either the .IMPORT or the .GLOBAL directive.

Example: (In this example, a symbol defined in file A is referenced from file B.)
File A:

```
        .GLOBAL     X           ; This statement declares X to be an
                                ; externally defined symbol.
        ~

X:      .EQU        H'10000000  ; This statement defines X.

        ~
```

File B:

```
        .GLOBAL     X           ; This statement declares X to be an
                                ; externally referenced symbol.
        ~

        .ALIGN      4           ;
        .DATA.L     X           ; This statement references X.

        ~
```

.OUTPUT

Description Format: Δ.OUTPUTΔ<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.

Output Specifier	Output Control
obj	An object module is output.
noobj	No object module is output.

(2) Output of debugging information

Controls the output of the debugging information.

Output Specifier	Output Control
dbg	Debugging information is output in the object module.
nodbg	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.

Example:

```
    ～
.OUTPUT   OBJ
.OUTPUT   NODBG     ← OK
    ～
```

```
    ～
.OUTPUT   OBJ
.OUTPUT   NOOBJ     ← Error
    ～
```

Specifications concerning debugging information output are only valid when an object module is output.

The assembler gives priority to command line option specifications concerning the object module and debugging information output.

Example 1: Note: This example and its description assume that no command line options concerning object module or debugging information output were specified.

```
.OUTPUT  OBJ          ; An object module is output.
                      ; No debugging information is output.
~
```

Example 2:

```
.OUTPUT  OBJ,DBG      ; Both an object module and debugging
                      ; information is 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: Δ.DEBUGΔ<output specifier>

<output specifier>= { ON | OFF }
The label field is not used.

Description: .DEBUG controls the output of symbolic debugging information.
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.

Output Specifier	Output Control
on	Symbolic debugging information is output.
off	No symbolic debugging information is output.

Example:

```
~

.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 debug
                  ; information.

~
```

Supplement: The term "symbolic debugging information" refers to the parts of debugging information concerned with symbols.

.ENDIAN

Description Format: Δ.ENDIANΔ<endian>

<endian>:{ BIG | LITTLE }
The label field is not used.

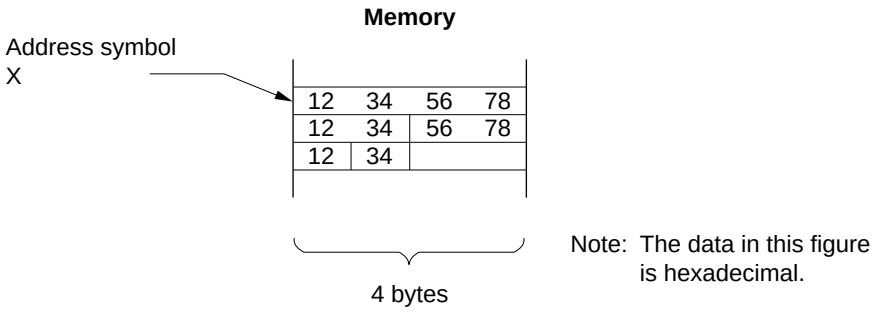
Description: .ENDIAN selects the big endian or little endian.
Enter a .ENDIAN directive at the beginning of the source program.
The assembler gives priority to command line option specifications concerning endian selection.

Endian	Output Control
BIG	Assembles program in big endian
LITTLE	Assembles program in little endian

Example: 1. When the big endian is selected

```
    .CPU      SH3
    .ENDIAN   BIG          ; This statement selects the big endian.

    ~

X:  .DATA.L   H'12345678   ;
    .DATA.W   H'1234,H'5678 ; These statements reserve integer
                           ; data.
    .DATA.B   H'12,H'34    ;

    ~
```

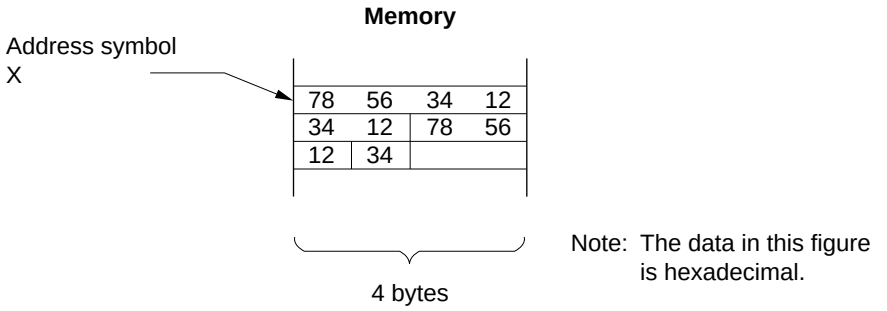
Explanatory Figure for the Coding Example



Example: 2. When the little endian is selected

```
    .CPU      SH3
    .ENDIAN   LITTLE        ; This statement selects the little endian.

    ~

X:
    .DATA.L   H'12345678    ;
    .DATA.W   H'1234,H'5678 ; These statements reserve integer
                            ; data.
    .DATA.B   H'12,H'34     ;

              ~
```

Explanatory Figure for the Coding Example



.LINE

Description Format: Δ.LINEΔ ["<file name>",]<line number>

The label field is not used.

Description: .LINE changes the file name and line number referred to at error message output or at debugging.
The line number and the file name specified with a .LINE directive is valid until the next .LINE.
The compiler (version 3.0 or later) generates .LINE corresponding to the line in the C source file when the debugging option is specified and the assembly source program is output.
If the file name is omitted, the file name is not changed, but only the line number is changed.

Example:

```
shc  –code=asmcode -debug test.c
```

C source program (test.c)

```
int    func()
{
   int   i,j;

   j=0;
   for (i=1;i<=10;i++){
       j+=i;
   }
   return(j);
 }
```

→

Assembly source program (test.src)

```
       .EXPORT    _func
       .SECTION   P,CODE,ALIGN=4
       .LINE      "/asm/test.c",1
_func:                  ; function: func
                        ; frame size=0
       .LINE      "/asm/test.c",5
       MOV        #0,R5
       .LINE      "/asm/test.c",6
       MOV        #10,R6
       MOV        #1,R4
L212:
       .LINE      "/asm/test.c",7
       ADD        R4,R5
       ADD        #1,R4
       .LINE      "/asm/test.c",6
       CMP/GT     R6,R4
       BF         L212
       .LINE      "/asm/test.c",10
       RTS
       .LINE      "/asm/test.c",9
       MOV        R5,R0
       .END
```

.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*[1]	Assembler Action
(1)	list	An assemble listing is output.*[2]
	nolist	No assemble listing is output.*[2]
(2)	src	A source program listing is output in the assemble listing.*[3]*[4]
	nosrc	No source program listing is output in the assemble listing.*[3]*[4]
(3)	cref	A cross-reference listing is output in the assemble listing.*[3]*[4]
	nocref	No cross-reference listing is output in the assemble listing.*[3]*[4]
(4)	sct	A section information listing is output in the assemble listing.*[3]*[5]
	nosct	No section information listing is output in the assemble listing.*[3]*[6]

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
.PRINT  NOSRC
   ~
```

← OK

```
   ~
.PRINT  LIST
.PRINT  NOLIST
   ~
```

← Error

The output specifiers concerned with the source program listing, the cross-reference listing, and the section information listing are only valid when an assemble listing is output.
The assembler gives priority to command line option specifications concerning assemble listing output.

Example 1:

Note: This example and its description assume that no command line options concerning assemble listing output are specified.

```
.PRINT      LIST     ; All types of assemble listing are output.

~
```

Example 2:

```
.PRINT      LIST,NOSRC,NOCREF
                     ; Only a section information listing is output.

~
```

.LIST

Description Format: Δ.LISTΔ<output specifier>[,...]

Δ<output specifier>={ ON |OFF | COND | NOCOND | DEF | NODEF |
CALL | NOCALL | EXP | NOEXP |
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	on	off	Source statements	The source statements following this directive
b	cond	nocond	Failed condition*1	Condition-failed .AIF or .AIFDEF directive statements
	def	nodef	Definition*1	Macro definition statements .AREPEAT and .AWHILE definition statements .INCLUDE directive statements .ASSIGNA and .ASSSIGNC directive statements
	call	nocall	Call*1	Macro call statements, .AIF, AIFDEF, and .AENDI directive statements
	exp	noexp	Expansion*1	Macro expansion statements .AREPEAT and .AWHILE expansion statements
c	code	nocode	Object code lines*1	The object code lines exceeding the source statement lines

Note: This specification is valid when the **show** or **noshow** option is not specified.

The specification of the .LIST directive is only valid when an assemble listing is output.
The assembler gives priority to command line option specifications concerning source program listing output.

.LIST directive statements themselves are not output on the source program listing.

Example: This example assumes that the command line specifies nothing about the source program listing.

```
          .LIST   NOCOND,NODEF
          .MACRO  SHLRN   COUNT,Rd

SHIFT     .ASSIGNA \COUNT

          .AIF \&SHIFT GE 16
           SHLR16  \Rd
SHIFT     .ASSIGNA \&SHIFT-16
          .AENDI

          .AIF \&SHIFT GE 8
           SHLR8   \Rd
SHIFT     .ASSIGNA \&SHIFT-8
          .AENDI

          .AIF \&SHIFT GE 4
           SHLR2   \Rd
           SHLR2   \Rd
SHIFT     .ASSIGNA \&SHIFT-4
          .AENDI

          .AIF \&SHIFT GE 2
           SHLR2   \Rd
SHIFT     .ASSIGNA \&SHIFT-2
          .AENDI

          .AIF \&SHIFT GE 1
           SHLR    \Rd
          .AENDI
          .ENDM

          SHLRN    23,R0
```

This statement controls source program listing output.

These statements define a general-Purpose multiple-bit shift procedure as a macro instruction.

Macro call

Source Listing Output of Coding Example:

The .LIST directive suppresses the output of the macro definition, .ASSIGNA and .ASSIGNC directive statements, and .AIF and .AIFDEF condition-failed statements.

```
31                                  31
32                                  32              SHLRN    23,R0
33                                     M
35                                     M
36                                     M            .AIF 23  GE 16
37 00000000 4029                       C             SHLR16  R0
39                                     M            .AENDI
40                                     M
41                                     M            .AIF 7   GE 8
45                                     M
46                                     M            .AIF 7   GE 4
47 00000002 4009                       C             SHLR2   R0
48 00000004 4009                       C             SHLR2   R0
50                                     M            .AENDI
51                                     M
52                                     M            .AIF 3   GE 2
53 00000006 4009                       C             SHLR2   R0
55                                     M            .AENDI
56                                     M
57                                     M            .AIF 1   GE 1
58 00000008 4001                       C             SHLR    R0
59                                     M            .AENDI
```

.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	When Not Specified
LIN=<line count>	The specified value is set to the number of lines per page.	20 to 255	60
COL=<column count>	The specified value is set to the number of columns per line.	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.

Example: Note: This example and its description assume that no command line options concerning the assemble listing line count and/or column count are specified.

```
~
.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: Δ.HEADINGΔ"<string literal>"

The label field is not used.

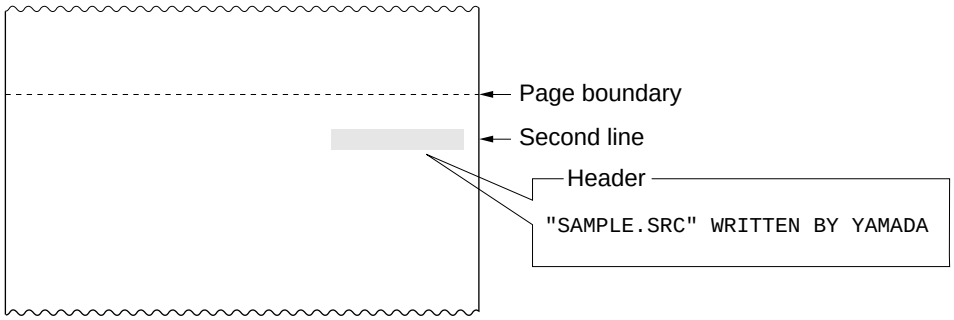
Description: .HEADING sets the header for the source program listing.
A string literal of up to 60 characters can be specified as the header.
The .HEADING directive can be used any number of times in a given source program.
The range of validity for a given use of the .HEADING directive is as follows:

- 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.

Example:

```
~
.HEADING  """SAMPLE.SRC""  WRITTEN BY YAMADA"
~
```

Explanatory Figure for the Coding Example



.PAGE

Description Format: Δ.PAGE

The label field is not used.

Description: .PAGE inserts a new page in the source program listing at an arbitrary point. 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.

Example:

```
~
MOV         R0,R1
RTS
MOV         R0,R2
.PAGE       ;A new page is specified here since the section changes at  this point.
.SECTION    DT,DATA,ALIGN=4
.DATA.L     H'11111111
.DATA.L     H'22222222
.DATA.L     H'33333333
~
```

Explanatory Figure for the Coding Example

Source program listing

```
   18  00000022  6103               18            MOV       R0,R1
   19  00000024  000B               19            RTS
   20  00000026  6203               20            MOV       R0,R2
```

New page

```
*** SuperH RISC engine ASSEMBLER Ver. 5.0 ***      06/01/00 10:15:30
PROGRAM NAME =

   22  00000000                     22            .SECTION  DT,DATA,ALIGN
   23  00000000  11111111           23            .DATA.L   H'11111111
   24  00000004  22222222           24            .DATA.L   H'22222222
   25  00000008  33333333           25            .DATA.L   H'33333333
```

.SPACE

Description Format: Δ.SPACE[Δ<line count>]

The label field is not used.

Description: .SPACE outputs the specified number of blank lines to the source program listing.
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.
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.

Example:

```
.SECTION    DT1,DATA,ALIGN=4
.DATA.L     H'11111111
.DATA.L     H'22222222
.DATA.L     H'33333333
.DATA.L     H'44444444          ;Inserts five blank lines at the point
.SPACE  5                       ; where the section changes.
.SECTION    DT2,DATA,ALIGN=4
~
```

Explanatory Figure for the Coding Example

Source program listing

```
*** SuperH RISC engine ASSEMBLER Ver. 5.0 ***      06/01/00 10:15:30
PROGRAM NAME =

    1  00000000                       1            .SECTION   DT1,DATA,ALIGN=4
    2  00000000  11111111             2            .DATA.L    H'11111111
    3  00000004  22222222             3            .DATA.L    H'22222222
    4  00000008  33333333             4            .DATA.L    H'33333333
    5  0000000C  44444444             5            .DATA.L    H'44444444









    7  00000000                       7            .SECTION   DT2,DATA,ALIGN=4
```

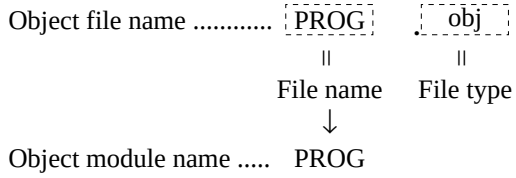
.PROGRAM

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 upper-case 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:



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: Δ.RADIXΔ<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.
When there is no radix specification with the .RADIX directive in a program, integer constants with no radix specification are interpreted as decimal constants.
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.

Radix Specifier	Integer Constant with no Radix
B	Binary
Q	Octal
D	Decimal
H	Hexadecimal

Example: 1.

```
        ~
        .RADIX  D
X:      .EQU    100  ;This 100 is decimal.
        ~
        .RADIX  H
Y:      .EQU    64   ;This 64 is hexadecimal.
        ~
```

2.

```
        ~
        .RADIX  H

Z:      .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>
Label: 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=4
        START:
                ~
                .END      START     ;Declares the end of the source program.
                                    ;Symbol START becomes the entry point.
```

11.5 File Inclusion Function

The file inclusion function allows source files to be included into other source files at assembly. 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=4
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=4
      MOV #ON,R0

       ~
```

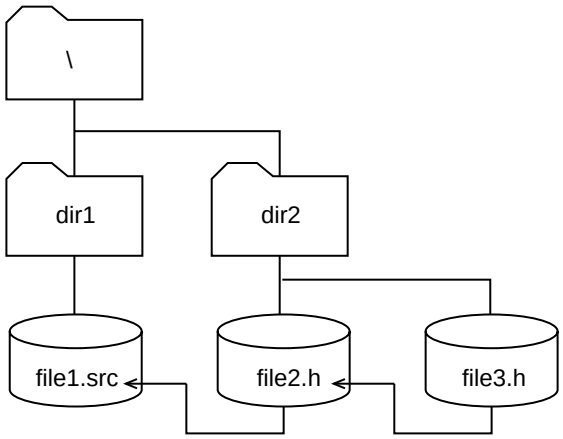
.INCLUDE

Description Format: Δ.INCLUDE Δ"<file name>"

The label field is not used.

Description: .INCLUDE is the file inclusion assembler directive. If no file type is specified, only the file name is used as specified (the assembler does not assume any default file type).
The file name can include the directory. 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).
Include files can include other files. The nesting depth for file inclusion is limited to 30 levels.
The current directory for the .INCLUDE directive in a source file is the directory where the assembler is invoked. The current directory for the .INCLUDE directive in an include file is the directory where the include file exits.
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 (\)
- Inputs source file \dir1\file1.src
- Makes file2.h included in file1.src
- Makes file3.h included in file2.h

The start command is as follows:

```
>asmsh \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 (/).

11.6 Conditional Assembly Function

11.6.1 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 (these variables can be redefined).

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
        ～
       .AIF \&FLAG EQ 1        ; MOV R0,R1 is assembled
        MOV R0,R1              ; when FLAG is 1.
       .AENDI
        ～
```

(b) Character preprocessor variables

Character preprocessor variables are defined by the .ASSIGNC directive (these variables can be redefined).

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"
        ~
       .AIF "\&FLAG" EQ "ON"   ; MOV R0,R1 is assembled
        MOV 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.L       SYM1,R0     ; Replaced with MOV.L 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  R0,R10            ; Assembled when FLAG
 MOV  R1,R11            ; is ON.
 MOV  R2,R12            ;
.AELSE
 MOV  R10,R0            ; Assembled when FLAG
 MOV  R11,R1            ; is not ON.
 MOV  R12,R2            ;
.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     R0,R10          ; Assembled when FLAG is defined with
 MOV     R1,R11          ; the .DEFINE directive before the .AIFDEF
 MOV     R2,R12          ; directive in the program.
.AELSE
 MOV     R10,R0          ; Assembled when FLAG is not defined with
 MOV     R11,R1          ; the .DEFINE directive before the .AIFDEF
 MOV     R12,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:

```
                        ; This example is a division of 64-bit data by 32-bit data.
                        ; R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
TST       R0,R0         ; Zero divisor check
BT        zero_div
CMP/HS    R0,R1         ; Overflow check
BT        over_div
DIV0U                   ; Flag initialization
.AREPEAT  32
 ROTCL    R2            ; These statements are iteratively assembled 32 times.
 DIV1     R0,R1         ;
.AENDR
ROTCL     R2            ; R2 = quotient
```

(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:

```
                                        ;  This example is a multiply and
                                        ; accumulate
                                        ;  operation.
TblSiz:  .ASSIGNA   50                  ;  TblSiz: Data table size
         MOV        A_Tbl1,R1           ;  R1: Start address of data table 1
         MOV        A_Tbl2,R2           ;  R2: Start address of data table 2
         CLRMAC                         ;  MAC register initialization
         .AWHILE    \&TblSiz GT 0       ;  While TblSiz is larger than 0,
         MAC.W      @R1+,@R2+           ;  this statement is iteratively assembled.
TblSiz:  .ASSIGNA   \&TblSiz-1          ;  1 is subtracted from TblSiz.
         .AENDW
         STS        MACL,R0             ;  The result is obtained in R0.
```

11.6.2 Conditional Assembly Directives

This assembler provides the following 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 .AELIF .AELSE .AENDI	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.
	.AIFDEF .AELSE .AENDI	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.
Iterated expansion	.AREPEAT .AENDR	Repeats assembly of a part of a source program (between .AREPEAT and .AENDR) the specified number of times.
	.AWHILE .AENDW	Assembles a part of a source program (between .AWHILE and .AENDW) iteratively while the specified condition is satisfied.
	.EXITM	Terminates .AREPEAT or .AWHILE iterated expansion.
Others	.AERROR	Processes an error during preprocessor expansion.
	.ALIMIT	Specifies the maximum count of .AWHILE expansion.

.ASSIGNA

Description Format: <preprocessor variable>[:] Δ.ASSIGNA Δ<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:

```
                              ; This example generates a general-purpose multiple-bit
                              ; shift instruction which shifts bits to the right by the
                              ; number of SHIFT.
RN:     .REG  R0              ; R0 is set to Rn.
SHIFT:  .ASSIGNA  27          ; 27 is set to SHIFT.

        .AIF \&SHIFT GE 16    ; Condition: SHIFT ≥ 16
         SHLR16  Rn           ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
SHIFT:  .ASSIGNA  \&SHIFT-16  ; 16 is subtracted from SHIFT.
        .AENDI

        .AIF \&SHIFT GE 8     ; Condition: SHIFT ≥ 8
         SHLR8  Rn            ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
SHIFT:  .ASSIGNA  \&SHIFT-8   ; 8 is subtracted from SHIFT.
        .AENDI

        .AIF \&SHIFT GE 4     ; Condition: SHIFT ≥ 4
         SHLR2  Rn            ; When the condition is satisfied, Rn is shifted to the right by 4 bits.
         SHLR2  Rn            ;
SHIFT:  .ASSIGNA  \&SHIFT-4   ; 4 is subtracted from SHIFT.
        .AENDI

        .AIF \&SHIFT GE 2     ; Condition: SHIFT ≥ 2
         SHLR2  Rn            ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
SHIFT:  .ASSIGNA  \&SHIFT-2   ; 2 is subtracted from SHIFT.
        .AENDI

        .AIF \&SHIFT EQ 1     ; Condition: SHIFT = 1
         SHLR     Rn          ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
        .AENDI
```

The expanded results are as follows:

```
        SHLR16  R0      ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
        SHLR8   R0      ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
        SHLR2   R0      ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
        SHLR    R0      ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
```

.ASSIGNC

Description Format: <preprocessor variable>[:] Δ.ASSIGNC Δ"<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:

```
FLAG:  .ASSIGNC  "ON"          ; “ON” is set to FLAG.
       ~

       .AIF "\&FLAG" EQ "ON" ; MOV R0,R1 is assembled
        MOV R0,R1             ; when FLAG is "ON".
       .AENDI
       ~

FLAG:  .ASSIGNC  "\&FLAG "     ; A space (" ") is added to FLAG.
FLAGA: .ASSIGNC  "OFF"         ; "OFF" is added to FLAGA.
FLAG:  .ASSIGNC  "\&FLAG'AND \&FLAGA"
                               ; An apostrophe (') is used to distinguish FLAG and
                               ; AND.
                               ; FLAG finally becomes "ON AND OFF".
       ~
```

.DEFINE

Description Format: <symbol>[:] Δ.DEFINE Δ"<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.L       SYM1,R0       ;  Replaced with MOV.L 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 .DEFINE directive. Add 0 to the beginning of the number to stop replacing such number.

```
A0:     .DEFINE "0"
        MOV.B   #H'A0,R0        ;       Replaced with MOV.B #H'0,R0.
        MOV.B   #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.B   #B'10,R0        ;       Replaced with MOV.H #H'10,R0.
```

.AIF, .AELIF, .AELSE, .AENDI

Description Format: Δ.AIF Δ<term1> Δ<relational operator> Δ<term2>
<Source statements assembled if the AIF condition is satisfied>
[Δ.AELIF Δ<term1> Δ<relational operator> Δ<term2>
<Source statements assembled if the AELIF condition is satisfied>]
[Δ.AELSE
<Source statements assembled if all the conditions are not satisfied>]
.AENDI

The label field is not used.

Description: .AIF, .AELIF, .AELSE, and .AENDI are the assembler directives that 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 condition 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 ("). To specify a double quotation mark in a string literal, enter two double quotation marks in succession.
The following relational operators can be used:

```
EQ: term1 = term2
NE: term1 ≠ term2
GT: term1 > term2
LT: term1 < term2
GE: term1 ≥ term2
LE: term1 ≤ term2
```

Numeric values are handled as 32-bit signed integers. For string literals, only EQ and NE conditions can be used.

Example:

```
~

.AIF \&TYPE EQ 1
 MOV R0,R3              ; These statements
 MOV R1,R4              ; are assembled
 MOV R2,R5              ; when TYPE is 1.
.AELIF \&TYPE EQ 2
 MOV R0,R6              ; These statements
 MOV R1,R7              ; are assembled
 MOV R2,R8              ; when TYPE is 2.
.AELSE
 MOV R0,R9              ; These statements
 MOV R1,R10             ; are assembled
 MOV R2,R11             ; when TYPE is not 1 nor 2.
.AENDI

~
```

.AIFDEF, .AELSE, .AENDI

Description Format: Δ.AIFDEF Δ<replacement symbol>
<statements to be assembled when the specified replacement symbol is defined>
[Δ.AELSE
<statements to be assembled when the specified replacement symbol is not defined>]
.AENDI
The label field is not used.

Description: .AIFDEF, .AELSE, and .AENDI are the assembler directives that select whether or not to assemble source statements according to the replacement symbol definition.
The condition 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.

The replacement symbol can be defined by the .DEFINE directive
When the specified replacement symbol is defined by the command line option or defined before being referenced by theses 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        R0,R3       ; These statements are assembled when
 MOV        R1,R4       ; FLAG is defined by .DEFINE directive.
.AELSE
 MOV       R0,R6        ; These statements are assembled when
 MOV       R1,R7        ; FLAG is not defined by .DEFINE directive.
.AENDI
```

~

.AREPEAT, .AENDR

Description Format: Δ.AREPEAT Δ<count>
<Source statements iteratively assembled>
.AENDR

The label field is not used.

Description: .AREPEAT and .AENDR are the assembler directives that assemble source statements by iteratively expanding them the specified number of times. The condition 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:

```
                              ; This example is a division of 64-bit data by 32-bit data.
                              ; R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
    TST       R0,R0           ; Zero divisor check
    BT        zero_div
    CMP/HS    R0,R1           ; Overflow check
    BT        over_div
    DIV0U                     ; Flag initialization
    .AREPEAT  32
    ROTCL     R2              ; These statements are
    DIV1      R0,R1           ; iterated 32 times.
    .AENDR
    ROTCL     R2              ; R2 = quotient
```

.AWHILE, .AENDW

Description Format: Δ.AWHILE Δ<term1> Δ<relational operator> Δ<term2>
<Source statements iteratively assembled>
Δ.AENDW
The label field is not used.

Description: .AWHILE and .AENDW are the assembler directives that assemble source statements by iteratively expanding them while the specified condition is satisfied.

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 ("). To specify a double quotation mark in a string literal, enter two double quotation marks (" ") in succession.

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 ≠ term2
GT: term1 > term2
LT: term1 < term2
GE: term1 ≥ term2
LE: term1 ≤ term2
```

Numeric values are handled as 32-bit signed integers. For string literals, only EQ and NE conditions can be used.

Example:

```
                                    ; This example is a multiply and accumulate
                                    ; operation.
TblSiz:  .ASSIGNA  50               ; TblSiz: Data table size
         MOV       A_Tbl1,R1        ; R1: Start address of data table 1
         MOV       A_Tbl2,R2        ; R2: Start address of data table 2
         CLRMAC                     ; MAC register initialization
         .AWHILE   \&TblSiz GT 0    ; While TblSiz is larger than 0,
         MAC.W    @R0+,@R1+         ; this statement is iteratively assembled.
TblSiz:  .ASSIGNA  \&TblSiz-1       ; 1 is subtracted from TblSiz.
         .AENDW
         STS       MACL,R0          ; The result is obtained in R0.
```

.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
                                ; specified.
         ADD     R0,R1
         ADD     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   R0,R1  .....When COUNT is 0
        ADD   R2,R3
        ADD   R0,R1 ·····When COUNT is 1
        ADD   R2,R3
```

After this, COUNT becomes 2 and expansion is terminated.

.AERROR

Description Format: Δ.AERROR
The label field is not used.

Description: When the .AERROR directive is assembled, error 667 is generated and the assembler is terminated with an error.
The .AERROR directive can be used to check values such as preprocessor variables.

Example:

```
        ~

.AIF        \&FLG EQ 1
 MOV        R1,R10
 MOV        R2,R11
.AELSE
.AERROR                 ; When \&FLG is not 1, an error is generated.
.AENDI

        ~
```

.ALIMIT

Description Format: Δ.ALIMIT Δ<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:

```
        .ALIMIT    20
          ~

FLG:    .ASSIGNA   0
        .AWHILE    \&FLG EQ 0      ; Expansion is terminated after performed
         NOP                       ; 20 times, and a warning message is output.
        .AENDW

          ~
```

11.7 Macro Function

11.7.1 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 R0, R1, R2,
 MOV R0,R10              ; and R3 is defined as macro instruction SUM.
 ADD R1,R10
 ADD R2,R10
 ADD R3,R10
.ENDM
 ~

 SUM                     ; This statement calls macro instruction SUM.
                         ; Macro body   MOV  R0,R10
                         ;              ADD  R1,R10
                         ;              ADD  R2,R10
                         ;              ADD  R3,R10
                         ; 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 R0,\ARG1           ; ARG1 is referenced in the macro body.
 ADD R1,\ARG1
 ADD R2,\ARG1
 ADD R3,\ARG1
.ENDM
        ~

 SUM R10                ; This statement calls macro instruction SUM
                        ; specifying macro parameter R10.
                        ; The argument in the macro body is
                        ; replaced with the macro parameter, and
                        ;       MOV  R0,R10
                        ;       ADD  R1,R10
                        ;       ADD  R2,R10
                        ;       ADD  R3,R10 is expanded.
```

11.7.2 Macro Function Directives

This assembler provides the following macro function directives.

Table 11.29 Macro Function Directives

Directive	Description
.MACRO	Defines a macro instruction.
.ENDM	
.EXITM	Terminates macro instruction expansion. Refer to section 11.6.2, .EXITM.

.MACRO, .ENDM

Description Format: Δ.MACROΔ<macro name>[Δ<argument>[,...]]
Δ.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:

```
 ~
.MACRO  SUM                ; Processing to obtain the sum of R0, R1, R2,
 MOV R0,R10                ; and R3 is defined as macro instruction SUM.
 ADD R1,R10
 ADD R2,R10
 ADD R3,R10
.ENDM
 ~

 SUM                       ; This statement calls macro instruction SUM
                           ; Macro body  MOV  R0,R10
                           ;             ADD  R1,R10
                           ;             ADD  R2,R10
                           ;             ADD  R3,R10 is expanded.
```

11.7.3 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 a macro body 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      #1,R1             ; Argument P1 is referenced.
.SDATA   "R1"              ; Argument P is referenced.
```

(2) Preprocessor variable reference

Preprocessor variables can be referenced in macro bodies.

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   RES_STR STR, Rn
           MOV.L   #str\@,\Rn
           BRA     end_str\@
           NOP
str\@     .SDATA   "\STR"
          .ALIGN   2
end_str\@
          .ENDM
           RES_STR "ONE",R0
           RES_STR "TWO",R1
```

Different symbols are generated each time RES_STR is expanded.

Expanded results are as follows:

```
           MOV.L   #str00000,R0
           BRA     end_str00000
           NOP
str00000  .SDATA   "ONE"
          .ALIGN   2
end_str00000
           MOV.L   #str00001,R1
           BRA     end_str00001
           NOP
str00001  .SDATA   "TWO"
          .ALIGN   2
end_str00001
```

(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       #"\",R0)        ; \ is expanded as an ASCII character.
.ENDM
```

Expanded results are as follows:

```
MOV     #"\",R0             ; \ is expanded as an ASCII character.
```

(5) Comments in macros

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 (to avoid repeating the same comment in the listing file), 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 PUSH Rn
MOV.L    \Rn,@-R15          \;   \Rn is a register.
.ENDM
PUSH     R0
```

Expanded results are as follows (the comment is not expanded):

```
MOV.L    R0,@-R15
```

(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.

11.7.4 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=9       ;  Macro instruction SUM and arguments
                                    ;  FROM and TO are defined.
       MOV    R\FROM,R10
COUNT  .ASSIGNA       \FROM+1
       .AWHILE \&COUNT LE \TO        Macro body is coded
       MOV     R\&COUNT,R10          using arguments.
COUNT  .ASSIGNA        \&COUNT+1
       .AENDW
       .ENDM

       SUM     0,5                   Both will be expanded
       SUM     TO=5                  into the same statements.
```

Expanded results are as follows (the arguments in the macro body are replaced with macro parameters):

```
       MOV     R0,R10
       MOV     R1,R10
       MOV     R2,R10
       MOV     R3,R10
       MOV     R4,R10
       MOV     R5,R10
```

11.7.5 String Literal Manipulation Functions

This assembler provides the following string literal manipulation functions.

Function	Description
.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.

String literals are specified by enclosing the desired characters with double quotation marks ("). To specify a double quotation mark in a string literal, enter two double quotation marks in succession.

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
.ALIGN 4                    .
.SRES   .LEN("\P1")
.ENDM
 ~
RESERVE_LENGTH ABCDEF
RESERVE_LENGTH ABC
```

Expanded results are as follows:

```
.ALIGN  4
.SRES   6                   ; "ABCDEF" has six characters.
.ALIGN  4
.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 of the found string (with 0 indicating the start of the string) in decimal with no radix. .INSTR is replaced with –1 if string literal 2 does not appear in string literal 1.

String literals are specified by enclosing the desired characters with double quotation marks ("). To specify a double quotation mark in a string literal, enter two double quotation marks in succession.

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("ABCDEFG","\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 "ABCDEFG"
.DATA.W  -1         ; "ABCDEFG" 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 (").
String literals are specified by enclosing the desired characters in double quotation marks ("). To specify a double quotation mark in a string literal, enter two double quotation marks in succession.
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("ABCDEFG",\P1,\P2)
.ENDM
 ~

RESERVE_STR 2,2
RESERVE_STR ,3              ; Macro parameter P1 is omitted.
```

Expanded results are as follows:

```
.SDATA  "CD"
.SDATA  "ABC"
```

11.8 Automatic Literal Pool Generation Function

11.8.1 Overview of Automatic Literal Pool Generation

To move 2-byte or 4-byte constant data (referred to below as a "literal") to a register, a literal pool (a collection of literals) must be reserved and referred to in PC relative addressing mode. For literal pool location, the following must be considered:

- Is data stored within the range that can be accessed by data move instructions?
- Is 2-byte data aligned to a 2-byte boundary and is 4-byte data aligned to a 4-byte boundary?
- Can data be shared by several data move instructions?
- Where in the program should the literal pool be located?

The assembler automatically generates from a single instruction a .DATA directive and a PC relative MOV or MOVA instruction, which moves constant data to a register.

For example, this function enables program (a) below to be coded as (b):

(a)

```
        MOV.L   DATA1,R0
        MOV.L   DATA2,R1

        ~

        .ALIGN 4
DATA1   .DATA.L H'12345678
DATA2   .DATA.L 500000
```

(b)

```
        MOV.L   #H'12345678,R0
        MOV.L   #500000,R1

        ~
```

11.8.2 Extended Instructions Related to Automatic Literal Pool Generation

The assembler automatically generates a literal pool corresponding to an extended instruction (MOV.W #imm, Rn; MOV.L #imm, Rn; or MOVA #imm, R0) and calculates the PC relative displacement value.

An extended instruction source statement is expanded to an executable instruction and literal data as shown in table 11.30.

Table 11.30 Extended Instructions and Expanded Results

Extended Instruction	Expanded Result
MOV.W #imm, Rn	MOV.W @(disp, PC), Rn and 2-byte literal data
MOV.L #imm, Rn	MOV.L @(disp, PC), Rn and 4-byte literal data
MOVA #imm, R0	MOVA @(disp, PC), R0 and 4-byte literal data

11.8.3 Size Mode for Automatic Literal Pool Generation

Automatic literal pool generation has two modes: size specification mode and size selection mode. In size specification mode, a data transfer instruction (extended instruction) whose operation size is specified is used to generate a literal pool. In size selection mode, when a transfer instruction without size specification is written, the assembler automatically checks the imm operand value and selects a suitable-size transfer instruction.

Table 11.31shows data transfer instructions and size mode.

Table 11.31 Data Transfer Instructions and Size Mode

Data Move Instruction	Size Specification Mode	Size Selection Mode
MOV #imm, Rn	Executable instruction	Selected by assembler
MOV.B #imm, Rn	Executable instruction	Executable instruction
MOV.W #imm, Rn	Extended instruction	Extended instruction
MOV.L #imm, Rn	Extended instruction	Extended instruction

(1) Size Specification Mode

In this mode, a data transfer instruction without size specification (MOV #imm,Rn) is handled as a normal executable instruction. This mode is used when **auto_literal** is not specified as the command line option.

(2) Size Selection Mode

In this mode, when a data transfer instruction without size specification (MOV #imm,Rn) is written, the assembler checks the imm operand value and automatically generates a literal pool if necessary. The imm value is checked for the signed value range.

This mode is used when **auto_literal** is specified as the command line option.

Table 11.32 shows the instructions selected depending on imm value range.

Table 11.32 Instructions Selected in Size Selection Mode

imm Specification	imm Value Range*	Selected Instruction
Constant value Constants symbols defined before reference Absolute address symbol defined before reference	H'FFFFFF80 to H'0000007F (–128 to 127)	MOV.B #imm, Rn
	H'FFFF8000 to H'FFFFFF7F (–32,768 to –129) H'00000080 to H'00007FFF (128 to 32,767)	MOV.W #imm, Rn Expansion result: [MOV.W @(disp, PC), Rn and 2-byte literal data]
	H'80000000 to H'FFFF7FFF (–2,147,483,648 to –32,769) H'00008000 to H'7FFFFFFF (32,768 to 2,147,483,647)	MOV.L #imm, Rn Expansion result: [MOV.L @(disp, PC), Rn and 4-byte literal data]
Relative address symbol Externally referenced symbol Constants symbols defined after reference Absolute address symbol defined after reference	Does not depend on imm value	MOV.L #imm, Rn Expansion result: [MOV.L @(disp, PC), Rn and 4-byte literal data]

Note: The values in parentheses ( ) are decimal.

11.8.4 Literal Pool Output

The literal pool is output to one of the following locations:

- After an unconditional branch and its delay slot instruction
- Where a .POOL directive has been specified by the programmer

Note that this location can be selected by command line option **literal**. The assembler outputs the literal corresponding to an extended instruction to the nearest output location following the extended instruction. The assembler gathers the literals to be output as a literal pool.

Note

When a label is specified in a delay slot instruction, no literal pool will be output to the location following the delay slot.

(1) Literal Pool Output after Unconditional Branch

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
        MOV.L  #H'FFFF0000,R0
        MOV.W  #H'FF00,R1
        MOV.L  #CD1_START,R2
        MOV    #H'FF,R3
        RTS
        MOV    R0,R10
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
   1 0000F000             1       .SECTION CD1,CODE,LOCATE=H'0000F000
   2 0000F000             2  CD1_START
   3 0000F000 D003        3       MOV.L   #H'FFFF0000,R0
   4 0000F002 9103        4       MOV.W   #H'FF00,R1
   5 0000F004 D203        5       MOV.L   #CD1_START,R2
   6 0000F006 E3FF        6       MOV     #H'FF,R3
   7 0000F008 000B        7       RTS
   8 0000F00A 6A03        8       MOV     R0,R10
   9                              **** BEGIN-POOL ****
  10 0000F00C FF00                DATA FOR SOURCE-LINE 4
  11 0000F00E 0000                ALIGNMENT CODE
  12 0000F010 FFFF0000            DATA FOR SOURCE-LINE 3
  13 0000F014 0000F000            DATA FOR SOURCE-LINE 5
  14                              ****  END-POOL  ****
  15                         9       .END
```

(2) Literal Pool Output to the .POOL Location

If literal pool output location after unconditional branches is not available within the valid displacement range (because the program has a small number of unconditional branches), the assembler outputs error 402. In this case, a .POOL directive must be specified within the valid displacement range.

The valid displacement range is as follows:

— Word-size operation: 0 to 511 bytes

— Longword-size operation: 0 to 1023 bytes

When a literal pool is output to a .POOL location, a branch instruction is also inserted to jump over the literal pool.

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START
        MOV.L   #H'FFFF0000,R0
        MOV.W   #H'FF00,R1
        MOV.L   #CD1_START,R2
        MOV     #H'FF,R3
        .POOL
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
  1 0000F000             1       .SECTION CD1,CODE,LOCATE=H'0000F000
  2 0000F000             2  CD1_START:
  3 0000F000 D003        3       MOV.L   #H'FFFF0000,R0
  4 0000F002 9103        4       MOV.W   #H'FF00,R1
  5 0000F004 D203        5       MOV.L   #CD1_START,R2
  6 0000F006 E3FF        6       MOV     #H'FF,R3
  7 0000F008             7       .POOL
  8                        **** BEGIN-POOL ****
  9 0000F008 A006              BRA TO END-POOL
 10 0000F00A 0009              NOP
 11 0000F00C FF00              DATA FOR SOURCE-LINE 4
 12 0000F00E 0000              ALIGNMENT CODE
 13 0000F010 FFFF0000          DATA FOR SOURCE-LINE 3
 14 0000F014 0000F000          DATA FOR SOURCE-LINE 5
 15                       ****  END-POOL  ****
 16                     8       .END
```

11.8.5 Literal Sharing

When the literals for several extended instructions are gathered into a literal pool, the assembler makes the extended instructions share identical immediate value.

The following operand forms can be identified and shared:

(1) Symbol

(2) Constant

(3) Symbol ± Constant

In addition to the above, expressions that are determined to have the same value at assembly processing may be shared.

However, extended instructions having different operation sizes do not share literal data even when they have the same immediate value.

An example of literal data sharing among extended instructions is shown on below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
        MOV.L   #H'FFFF0000,R0
        MOV.W   #H'FF00,R1
        MOV.L   #H'FFFF0000,R2
        MOV     #H'FF,R3
        RTS
        MOV     R0,R10
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
   1 0000F000             1       .SECTION CD1,CODE,LOCATE=H'0000F000
   2 0000F000             2  CD1_START:
   3 0000F000 D003        3       MOV.L   #H'FFFF0000,R0
   4 0000F002 9103        4       MOV.W   #H'FF00,R1
   5 0000F004 D202        5       MOV.L   #H'FFFF0000,R2
   6 0000F006 E3FF        6       MOV     #H'FF,R3
   7 0000F008 000B        7       RTS
   8 0000F00A 6A03        8       MOV     R0,R10
   9                         **** BEGIN-POOL ****
  10 0000F00C FF00              DATA FOR SOURCE-LINE 4
  11 0000F00E 0000              ALIGNMENT CODE
  12 0000F010 FFFF0000          DATA FOR SOURCE-LINE 3,5
  13                         ****  END-POOL  ****
  14                        9       .END
```

11.8.6 Literal Pool Output Suppression

When a program has too many unconditional branches, the following problems may occur:

- Many small literal pools are output
- Literals are not shared

In these cases, suppress literal pool output as shown below.

```
        ~
       <delayed branch instruction>
       <delay slot instruction>
       .NOPOOL
        ~
```

Source program

```
        ~
CASE1:
       MOV.L   #H'FFFF0000,R0   ---------- Extended instruction 1
       RTS
       NOP
       .NOPOOL                  ---------- No literal pool is output here
CASE2:
       MOV.L   #H'FFFF0000,R0   ---------- Extended instruction 2
       RTS
       NOP
                                ---------- Literal pool is output here
        ~
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
        ~
 20 0000F000             20 CASE1:
 21 0000F000 D002        21      MOV.L   #H'FFFF0000,R0
 22 0000F002 000B        22      RTS
 23 0000F004 0009        23      NOP
 24                      24      .NOPOOL
 25 0000F006             25 CASE2:
 26 0000F006 D001        26      MOV.L   #H'FFFF0000,R0
 27 0000F008 000B        27      RTS
 28 0000F00A 0009        28      NOP
 29                         **** BEGIN-POOL ****
 30 0000F00C FFFF0000           DATA FOR SOURCE-LINE 21,26
 31                         **** END-POOL  ****
        ~
```

11.8.7 Notes on Automatic Literal Pool Generation

(1) If an error occurs for an extended instruction

a. Extended instructions must not be specified in delay slots (error 151).

b. Extended instructions must not be specified in relative sections having a boundary alignment value of less than 2 (error 152).

c. MOV.L #imm, Rn or MOVA #imm, R0 must not be specified in relative sections having a boundary alignment value of less than 4 (error 152).

(2) If an error occurs when a .POOL directive is written

.POOL directives must not be written after unconditional delayed branches (error 522).

(3) If an error occurs when a .NOPOOL directive is written

.NOPOOL directives are valid only when written after delay slot instructions. If written at other locations, the .NOPOOL directive causes error 521.

(4) If the displacement of an executable instruction exceeds the valid range when an extended instruction is expanded

The assembler generates a literal pool and outputs error 402 for the instruction having a displacement outside the valid range.

Solution: Move the literal pool output location (for example, by the .NOPOOL directive), or change the location or addressing mode of the instruction causing the error.

(5) If the literal pool output location cannot be found

If the assembler cannot find a literal pool output location satisfying the following conditions in respect to the extended instruction,

— Same file

— Same section

— The nearest output location following the extended instruction

the assembler outputs, at the end of the section which includes the extended instruction, the literal pool and a BRA instruction with a NOP instruction in the delay slot to jump around the literal pool, and outputs warning 876.

(6) If the displacement from the extended instruction exceeds the valid range

If the displacement of the literal pool from the extended instruction exceeds the valid range, error 402 is generated.

Solution: Output the literal pool within the valid range (for example, using the .POOL directive.)

(7) Differences between size specification mode and size selection mode

Version 2.0 of the assembler can only use the size specification mode, but the size selection mode is added to the assembler version 3.1 or higher. If the source program created before for version 2.0 is assembled in the size selection mode by version 3.1 or higher, the imm values of data transfer instructions without size suffix will differ in the range of H'00000080 to H'000000FF (128 to 255) from these assembled by version 2.0.

An example of source listing output in the size specification mode and size selection mode is shown on below.

Example:

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
        MOV.L   #H'FF,R0
        MOV.W   #H'FF,R1
        MOV.B   #H'FF,R2
        MOV     #H'FF,R3
        RTS
        MOV     R0,R10
        .END
```

↓ ↓ ↓ ↓

Automatic literal pool output in size specification mode (source listing)

```
 1 0000F000             1        .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000 D003        2        MOV.L   #H'FF,R0
 3 0000F002 9103        3        MOV.W   #H'FF,R1
 4 0000F004 E2FF        4        MOV.B   #H'FF,R2
 5 0000F006 E3FF        5        MOV     #H'FF,R3
 6 0000F008 000B        6        RTS
 7 0000F00A 6A03        7        MOV     R0,R10
 8                         ***** BEGIN-POOL *****
 9 0000F00C 00FF           DATA FOR SOURCE-LINE 3
10 0000F00E 0000           ALIGNMENT CODE
11 0000F010 000000FF       DATA FOR SOURCE-LINE 2
12                         ***** END-POOL *****
13                     8        .END
```

The contents of R3 is H'FFFFFFFF.

Automatic literal pool output in size selection mode (source listing)

```
 1 0000F000             1        .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000 D003        2        MOV.L   #H'FF,R0
 3 0000F002 9103        3        MOV.W   #H'FF,R1
 4 0000F004 E2FF        4        MOV.B   #H'FF,R2
 5 0000F006 9301        5        MOV     #H'FF,R3
 6 0000F008 000B        6        RTS
 7 0000F00A 6A03        7        MOV     R0,R10
 8                         ***** BEGIN-POOL *****
 9 0000F00C 00FF           DATA FOR SOURCE-LINE 3,5
10 0000F00E 0000           ALIGNMENT CODE
11 0000F010 000000FF       DATA FOR SOURCE-LINE 2
12                         ***** END-POOL *****
13                     8        .END
```

The contents of R3 is H'000000FF.

11.9 Automatic Repeat Loop Generation Function

11.9.1 Overview of Automatic Repeat Loop Generation Function

In the SH-DSP, the start and end addresses of the repeat loop are set in the RS and RE registers by the LDRS and LDRE instructions. The address settings differ depending on the number of instructions in the repeat loop. When setting the address, consider the relationship between the address and the number of instructions in the repeat loop shown in table 11.33.

Table 11.33 Repeat Loop Instructions and Address Setting

Register Name	One Instruction	Two Instructions	Three Instructions	Four or more Instructions
RS	s_addr0+8	s_addr0+6	s_addr0+4	s_addr
RE	s_addr0+4	s_addr0+4	s_addr0+4	e_addr3+4

s_addr0: Address of the instruction one instruction before the repeat loop start address
s_addr: Repeat loop start address
e_addr3: Address of the instruction three instructions before the repeat loop end address

The automatic repeat loop generation automatically generates the PC relative instructions LDRS and LDRE, and the SETRC instruction from a single extended instruction. The LDRS and LDRE instructions transfer the repeat loop start and end to the RS and RE registers addresses based on the number of instructions in the repeat loop, and the SETRC instruction specifies the repetition count.

For example, program A can be written as program B when using the automatic repeat loop generation.

Program A:

```
            LDRS s_addr0+6
            LDRE s_addr0+4
            SETRC #10
 s_addr0:   NOP
            PADD A0,M0,A0 ;   Repeat loop start address
            PCMP X1,M0    ;   Repeat loop end address
```

Program B:

```
            REPEAT s_addr,e_addr,#10
            NOP
  s_addr:   PADD A0,M0,A0 ;   Repeat loop start address
  e_addr:   PCMP X1,M0    ;   Repeat loop end address
```

11.9.2 Extended Instructions of Automatic Repeat Loop Generation Function

The assembler automatically generates necessary instructions from extended instructions (REPEAT s_label,e_label,#imm, REPEAT s_label,e_label,Rn, and REPEAT s_label,e_label) and calculates the PC relative displacement.

Table 11.34 lists the source statement of each extended instruction and its expanded results of two or three executable instructions.

Table 11.34 Extended Instructions and Expanded Results

Extended Instruction	Expanded Results
REPEAT s_label,e_label,#imm	LDRS @(disp,PC), LDRE@(disp,PC), and SETRC #imm
REPEAT s_label,e_label,Rn	LDRS @(disp,PC), LDRE@(disp,PC), and SETRC Rn
REPEAT s_label,e_label	LDRS @(disp,PC) and LDRE@(disp,PC)

11.9.3 REPEAT Description

Description Format

```
[<symbol>[:]] ΔREPEAT Δ<start address>,<end address>[,<repeat count>]
```

Statement Elements

1. Start and end addresses
   Enter the labels of the start and end addresses of the repeat loop.
2. Repeat count
   Enter the repeat count as an immediate value or as a general register name.

Description

1. REPEAT automatically generates the executable instructions LDRS and LDRE to repeat the instructions in the range from the start address to the end address inclusive.
2. When the repeat count is specified, REPEAT generates a SETRC instruction. When the repeat count is omitted, SETRC is not generated.

11.9.4 Coding Examples

To Repeat Four or More Instructions (Basic Example):

```
            REPEAT RptStart,RptEnd,#5
            PCLR Y0
            PCLR A0
 RptStart:  MOVX @R4+,X1 MOVY @R6+,Y1
            PADD A0,Y0,Y0 PMULS X1,Y1,A0
       DCT  PCLR A0
            AND R0,R4
 RptEnd:    AND R0,R6
```

This program repeats execution of five instructions from RptStart to RptEnd five times.

The above program has the same meaning as the following:

```
            LDRS RptStart
            LDRE RptEnd3+4
            SETRC #5
            PCLR Y0
            PCLR A0
 RptStart:  MOVX @R4+,X1 MOVY @R6+,Y1
 RptEnd3:   PADD A0,Y0,Y0 PMULS X1,Y1,A0;        The label is not actually generated.
       DCT  PCLR A0
            AND R0,R4
 RptEnd:    AND R0,R6
```

To Repeat One Instruction: Specify the same labels as the start and end addresses.

```
            REPEAT Rpt,Rpt,R0
            MOVX @R4+,X1 MOVY @R6,Y1
Rpt:        PADD A0,Y0,Y0 PMULS X1,Y1,A0 MOVX @R4+,X1 MOVY @R6+,Y1
```

The above program has the same meaning as the following:

```
            LDRS RptStart0+8
            LDRE RptStart0+4
            SETRC R0
RptStart0:  MOVX @R4+,X1 MOVY @R6,Y1    ;The label is not actually generated.
Rpt:        PADD A0,Y0,Y0 PMULS X1,Y1,A0 MOVX @R4+,X1 MOVY @R6+,Y1
```

To Repeat Two Instructions:

```
            REPEAT RptStart,RptEnd,#10
            PCLR Y0
RptStart:   MOVX @R4+,X1 MOVY @R6+,Y1
RptEnd:     PADD A0,Y0,Y0 PMULS X1,Y1,A0
```

The above program has the same meaning as the following:

```
            LDRS RptStart0+6
            LDRE RptStart0+4
            SETRC #10
RptStart0:  PCLR Y0  ;The label is not actually generated.
RptStart:   MOVX @R4+,X1 MOVY @R6+,Y1
RptEnd:     PADD A0,Y0,Y0 PMULS X1,Y1,A0
```

To Repeat Three Instructions:

```
            REPEAT RptStart,RptEnd,R0
            PCLR Y0
 RptStart:  MOVX @R4+,X1 MOVY @R6+,Y1
            PMULS X1,Y1,A0
 RptEnd:    PADD A0,Y0,Y0
```

The above program has the same meaning as the following:

```
            LDRE RptStart0+4
            LDRS RptStart0+4
            SETRC R0
 RptStart0: PCLR Y0  ;The label is not actually generated.
 RptStart:  MOVX @R4+,X1 MOVY @R6+,Y1
            PMULS X1,Y1,A0
 RptEnd:    PADD A0,Y0,Y0
```

When Repeat Count is Omitted: When the repeat count is omitted, the assembler does not generate SETRC. To separate the LDRS and LDRE from the SETRC, omit the repeat count.

```
           REPEAT RptStart,RptEnd
           ; The LDRS and LDRE are expanded here.
           MOV #10,R0
 OuterLoop:
           SETRC #16
           PCLR Y0
           PCLR A0
 RptStart: MOVX @R4+,X1 MOVY @R6+,Y1
           PADD A0,Y0,Y0 PMULS X1,Y1,A0
      DCT  PCLR A0
           AND R0,R4
 RptEnd:   AND R0,R6
           DT R0
           BF OuterLoop
```

11.9.5 Notes on the REPEAT Extended Instruction

Start and End Addresses: Only labels in the same section or local labels in the same local block can be specified as the start and end addresses.

The start address must be at a higher address than the REPEAT extended instruction. The end address must be at a higher address than the start address.

Instructions Inside Loops:

- If one of the following assembler directives that reserve data or data area or a .ORG directive is used inside a loop, the assembler outputs a warning message and counts the directive as one of the instructions to be repeated. If a .ALIGN directive is used inside a loop to adjust the boundary alignment, the assembler outputs a warning message and counts the directive as one of the instructions to be repeated.

  The following are the directives which cause this action:
  .DATA, .DATAB, .SDATA, .SDATAB, .SDATAC, .SDATAZ, .FDATA, .FDATAB, .XDATA, .RES, .SRES, .SRESC, .SRESZ, .FRES, .ALIGN, and .ORG
- The assembler prevents automatic generation of literal pools within a loop. Therefore, even when an unconditional branch is used in a loop, no literal pool is generated. If a .POOL directive is used in a loop, the assembler outputs a warning message and ignores the .POOL directive.

Instruction Immediately before Loop: If three or fewer instructions are to be repeated, the instruction immediately before the loop must be an executable instruction or a DSP instruction. Therefore, when three or fewer instructions are to be repeated and if one of the following is located immediately before the start address of the loop, the assembler outputs an error.

- Assembler directives that reserve a data item or a data area or .ORG directive

  .DATA, .DATAB, .SDATA, .SDATAB, .SDATAC, .SDATAZ, .FDATA, .FDATAB, .XDATA, .RES, .SRES, .SRESC, .SRESZ, .FRES, or .ORG
- Literal pool generated by the automatic literal pool generation

  If an unconditional branch instruction and a delay slot instruction are located immediately before a loop, or if a .POOL directive is located immediately before a loop, a literal pool may be automatically generated. To prevent literal pool generation before a loop, use a .NOPOOL directive immediately after the delay slot instruction.
- One alignment byte generated by a .ALIGN directive

  When a .ALIGN directive is used at an odd address immediately before a loop, one alignment byte may be generated (for example, .ALIGN 4 is specified when the location counter value is 3). In this case, the contents of the byte before a loop is not an executable instruction, and an error message is output. If two or more alignment bytes are generated before a loop, their contents consist of a NOP instruction and the program can be correctly executed.

Others:

- One or more executable or DSP instructions must be located between a REPEAT extended instruction and the start address. Otherwise, the assembler outputs an error message.
- A REPEAT extended instruction must not be located between another REPEAT extended instruction and its end address. If REPEAT extended instructions are nested, the assembler outputs an error message; the first REPEAT is valid, and the other REPEAT instructions are ignored.

Section 12 Compiler Error Messages

12.1 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: I, W, E, F, or (–)) Error Message
Meaning of the error message.

Error levels are classified into the following five types:

- (I): Information error (Continues compiling processing and outputs the object program.)
- (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.)
- (–): Internal error (Aborts compiling processing.)

12.2 Error Messages

C0001 (I) "String literal" in comment
A comment has string literal.

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 condition for **if** or **switch** statement.

C0005 (I) Precision lost
Precision may be lost when assigning with type conversion a right hand side value to the left hand side value.

C0006 (I) Conversion in argument
A function parameter expression is converted into a parameter type specified in the prototype declaration.

C0008 (I) Conversion in return

A return statement expression is converted into a value type that should be returned from a function.

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.

C0012 (I) Unused variable "variable name"

An unused variable exists.

C0015 (I) No return value

A return statement is not returning a value in a function that should return a type other than the **void** type, or a return statement does not exist.

C0100 (I) Function "function name" not optimized

A function which is too large cannot be optimized.

C0200 (I) No prototype function

There is no prototype declaration.

C1000 (W) Illegal pointer assignment

A pointer is assigned to a pointer with a different data 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 (characters following a backslash) is used in a character constant or string literal.

C1007 (W) Long character constant

A character constant consists of two or more characters.

C1008 (W) Identifier too long

An identifier's length exceeds 8189 characters. Characters beyond 8190 are ignored.

C1010 (W) Character constant too long

A character constant consists of four or more characters.

C1012 (W) Floating point constant overflow

The value of a floating-point constant exceeds the limit. Assumes the internally represented value corresponding to +∞ or –∞ depending on the sign of the result.

C1013 (W) Integer constant overflow

The integer value exceeds the limit of unsigned long integer constant. Assumes a value ignoring the overflown 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 in a return statement 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) Register saving pragma conflicts in interrupt function "function name"

Invalid **#pragma** that controls saving or recovery of register contents corresponding to an interrupt function indicated by a "function name". The **#pragma** specification is ignored.

C1022 (W) First operand of "operator" is not lvalue

The first operand "operator" cannot be the lvalue.

C1023 (W) Can not convert Japanese code "code" to output type

Some Japanese codes cannot be converted into the specified output codes.

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 +∞ or –∞ depending on the sign of the operands.

C1201 (W) Ineffective floating point operation

Invalid floating-point operations such as ∞ - ∞ 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) "double=float" option ignored

double=float and **cpu=sh4** options are specified at the same time. The compiler ignores the **double=float** option and assumes that **fpu=single** option is specified.

C1400 (W) Function “function name” in #pragma inline is not expanded

A function specified using the **#pragma inline** could not be expanded. Ignores the **#pragma inline** specification.

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.

C2002 (E) Missing ","

A comma (,) is not used to delimit two arguments in a **#define** directive.

C2003 (E) Missing ")"

A right parenthesis ()) does not follow a name in a defined expression that determines whether the name is defined by a **#define** directive.

C2004 (E) Missing ">"

A right angle bracket (>) does not follow a file name in an **#include** directive.

C2005 (E) Cannot open include file "file name"

The file specified by an **#include** directive cannot be opened.

C2006 (E) Multiple #define's

The same macro name is redefined by **#define** directives.

C2008 (E) Processor directive #elif mismatches

There is no **#if**, **#ifdef**, **#ifndef**, or **#elif** directive corresponding to an **#elif** directive.

C2009 (E) Processor directive #else mismatches

There is no **#if**, **#ifdef**, or **#ifndef** directive corresponding to an **#else** directive.

C2010 (E) Macro parameters mismatch

The number of macro call arguments and the number of macro definition arguments are different.

C2011 (E) Line too long

After macro expansion, a source program line exceeds the compiler limit.

C2012 (E) Keyword as a macro name

A preprocessor keyword is used as a macro name in a **#define** or **#undef** directive.

C2013 (E) Processor directive #endif mismatches

There is no **#if**, **#ifdef**, or **#ifndef** directive corresponding to an **#endif** directive.

C2014 (E) Missing #endif

There is no **#endif** directive corresponding to an **#if**, **#ifdef**, or **#ifndef** directive, and the end of file is detected.

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 the limit.

C2017 (E) Missing "

A closing double quotation mark (") does not follow a file name in an **#include** directive.

C2018 (E) Illegal #line

The line count specified by a **#line** directive exceeds the limit.

C2019 (E) File name too long

The length of a file name exceeds the limit.

C2020 (E) System identifier "name" redefined

The name of the defined symbol is the same as that of intrinsic function.

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

A combination of type specifiers is illegal.

C2103 (E) Bad self reference structure

A struct 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 struct 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 type 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 type 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 struct or union is declared as a function type.

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, bool, enum** 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 struct, 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 struct, 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 type with a function type return value is specified.

C2126 (E) Illegal array size

The value specifying the number of array elements is not an integer or out of range of 1 to 2147483647.

C2127 (E) Missing array size

The number of elements in an array is not specified.

C2128 (E) Illegal pointer declaration for "*"

A type specifier other than **const** or **volatile** is specified following an asterisk (*), which indicates a pointer declaration.

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 as either the initial value of a struct, 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 or 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 struct, union, or array is greater than the number of struct 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 struct.

C2142 (E) Illegal void type

void is used illegally. **void** can only be used in the following cases:

(1) To specify a type assigned to a pointer

(2) To specify a function return 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.

C2144 (E) Type mismatch

Variables or functions with the same name which have an **extern** storage class are assigned to different data types.

C2145 (E) Const/volatile specified for incomplete type

An incomplete type is specified as a **const** or **volatile** type.

C2200 (E) Index not integer

An array index expression type is not an integer.

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 struct or union.

C2204 (E) Illegal member reference for "->"

The expression to the left-hand side of the -> operator is not a pointer to a struct or union.

C2205 (E) Undefined member name

An undeclared member name is used to reference a struct or union.

C2206 (E) Modifiable lvalue required for "operator"

The operand 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 on an expression that is not scalar.

C2208 (E) Pointer required for "*"

The unary operator * is used on an expression that is not pointer or on an expression of a pointer for **void**.

C2209 (E) Arithmetic type required for "operator"

The unary operator + or – is used on a non-arithmetic expression.

C2210 (E) Integer required for "~"

The unary operator ~ is used on 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 array, struct, or union is specified in a cast operator, or the operand of a cast operator is **void**, struct, or union and cannot be converted.

C2213 (E) Arithmetic type required for "operator"

The binary operator *, /, *=, or /= is used in an expression that is not an arithmetic expression.

C2214 (E) Integer required for "operator"

The binary operator <<, >>, &, |, ^, %, <<=, >>=, &=, |=, ^=, or %= is used in an expression that is not an integer expression.

C2215 (E) Illegal type for "+"

The combination of operand types used with the binary operator + is not allowed. Only the following type combinations are allowed for the binary operator +:

(1) Two arithmetic operands
(2) Pointer and integer

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 operands

(2) Two pointers assigned to the same data type

(3) The first operand is a pointer and the second operand is an integer.

C2218 (E) Scalar required

The first operand of the conditional operator ?: is not a scalar.

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 one of the following six combinations is allowed for the second and third operands when using the ?: operator:

(1) Two arithmetic operands

(2) Two **void** operands

(3) Two pointers assigned to the same data type

(4) A pointer and an integer constant whose value is zero or another pointer assigned to **void** that was converted from an integer constant whose value is zero

(5) A pointer and another pointer assigned to **void**

(6) Two struct 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 an assignment operator =, *=, /=, %=, +=, –=, <<=, >>=, &=, ^=, or | =.

C2221 (E) Illegal type for "operator"

The operand of the suffix operator ++ or -- is a pointer assigned to function type, **void** type, or to a data type other than scalar type.

C2222 (E) Type not compatible for "="

The operand types for the assignment operator = do not match. Only the following five types of combinations are allowed for the operands of the assignment operator =:

(1) Two arithmetic operands

(2) Two pointers assigned to the same data type

(3) The left operand is a pointer and the right operand is an integer constant whose value is zero or another pointer assigned to **void** that was converted from an integer constant whose value is zero.

(4) A pointer and another pointer assigned to **void**

(5) Two struct or union variables with the same data type

C2223 (E) Incomplete tag used in expression

An incomplete tag name is used for a struct 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 operands

(2) Two pointers assigned to the same data type

(3) A pointer and an integer constant whose value is zero or another pointer assigned to **void**

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 operands

(2) Two pointers assigned to the same data type

C2230 (E) Illegal function call

An expression which is not a function type or a pointer assigned 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 assigned to a function type, **void** type, or to a data type other than scalar type.

C2233 (E) Illegal array reference

An expression used as an array is an array or a pointer assigned to a data type other than a function or **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 with a floating-point type.

C2236 (E) Illegal cast in constant

In a constant expression, an attempt is made to cast a pointer with a **char** or **short** type.

C2237 (E) Illegal constant expression

In a constant expression, a pointer constant is cast with an integer and the result is manipulated.

C2238 (E) Lvalue or function type required for "&"

The unary operator & is not used in the lvalue or an expression other than function type.

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 integer constant expression.

C2307 (E) Multiple case labels

Two or more **case** labels with the same value are used 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

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 an integer.

C2312 (E) Missing (

The control expression (that determines statement execution) does not follow 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

A control expression (that determines statement execution) for an **if** statement is not a scalar.

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.

C2400 (E) Illegal character "character"

An illegal character is detected.

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 detected.

C2405 (E) Null character constant

There are no characters in a character constant (i.e., no characters are specified between two quotation marks).

C2406 (E) Out of float

The number of significant digits in a floating-point constant exceeds 17.

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 the limit of 255.

C2500 (E) Illegal token "phrase"

An illegal token sequence is used.

C2501 (E) Division by zero

An integer is divided by zero in a constant expression.

C2600 (E) String literal(s)

An error message specified by string literal **#error** is output to the list file if **nolistfile** option is not specified.

C2650 (E) Invalid pointer reference

The specified address does not match the boundary alignment value.

C2700 (E) Function "function name" in #pragma interrupt already declared

A function specified in an interrupt function declaration **#pragma interrupt** has been declared as a normal function.

C2701 (E) Multiple interrupt for one function

An interrupt function declaration **#pragma interrupt** has been declared more than once for the same function.

C2702 (E) Multiple #pragma interrupt options

The same type of interrupt is declared more than once.

C2703 (E) Illegal #pragma interrupt declaration

An interrupt function declaration **#pragma interrupt** is specified incorrectly.

C2704 (E) Illegal reference to interrupt function

The interrupt function is incorrectly referenced.

C2705 (E) Illegal parameter in interrupt function

Parameter types to be used for an interrupt function do not match.

C2706 (E) Missing parameter declaration in interrupt function

There is no declaration for a variable to be used for an optional specification of an interrupt function.

C2707 (E) Parameter out of range in interrupt function

The parameter value tn of an interrupt function exceeds the limit of 256.

C2709 (E) Illegal section name declaration

The **#pragma section** specification is illegal.

C2710 (E) Section name too long

The specified section name exceeds the limit of 31 characters.

C2711 (E) Section name table overflow

The number of section specified in one file exceeds the limit of 64.

C2712 (E) GBR based displacement overflow

The variable declared in **#pragma gbr_base** or **#pragma gbr_base1** overflows.

C2713 (E) Illegal #pragma interrupt function type

The function type specified **#pragma interrupt** is illegal.

C2800 (E) Illegal parameter number in in-line function

Parameters to be used for an intrinsic function do not match.

C2801 (E) Illegal parameter type in in-line function

There are different parameter types in an intrinsic function.

C2802 (E) Parameter out of range in in-line function

A parameter exceeds the range that can be specified by an intrinsic function.

C2803 (E) Invalid offset value in in-line function

An argument for an intrinsic function is incorrectly specified.

C2804 (E) Illegal in-line function

An intrinsic function that cannot be used by the specified **cpu** option exists.

C2805 (E) Function "function name" in #pragma inline/inline_asm already declared

The function indicated by a "function name" exists before the **#pragma** specification.

C2806 (E) Multiple #pragma for one function

Two or more **#pragma** directives are incorrectly specified for one function.

C2807 (E) Illegal #pragma inline/inline_asm declaration

The **#pragma inline** or **#pragma inline_asm** is specified illegally.

C2808 (E) Illegal option for #pragma inline_asm

The **code=machinecode** option is specified in addition to the **#pragma inline_asm** specification declaration.

C2809 (E) Illegal #pragma inline/inline_asm function type

An identifier type that specifies **#pragma inline** or **#pragma inline_asm** is illegal.

C2810 (E) Global variable "variable name" in #pragma gbr_base/gbr_base1 already declared

A variable definition indicated by "variable name" exists before **#pragma** specification.

C2811 (E) Multiple #pragma for one global variable

Two or more **#pragma** directives are incorrectly specified for one variable.

C2812 (E) Illegal #pragma gbr_base/gbr_base1 declaration

The **#pragma gbr_base** or **#pragma gbr_base1** specification is illegal.

C2813 (E) Illegal #pragma gbr_base/gbr_base1 global variable type

An identifier type that specifies **#pragma gbr_base** or **#pragma gbr_base1** is illegal.

C2814 (E) Function "function name" in #pragma noregsave/noregalloc/regsave already declared

The function indicated by a "function name" exists before the **#pragma** specification declaration.

C2815 (E) Illegal #pragma noregsave/noregalloc/regsave declaration

The **#pragma noregsave**, **#pragma noregalloc**, or **#pragma regsave** specification is illegal.

C2816 (E) Illegal #pragma noregsave/noregalloc/regsave function type

An identifier type that specifies **#pragma noregsave**, **#pragma noregalloc**, or **#pragma regsave** is illegal.

C2817 (E) Symbol "identifier" in #pragma abs16 already declared

A name indicated by an "identifier" exists before the **#pragma** specification declaration.

C2818 (E) Multiple #pragma for one symbol

More than one **#pragma** is incorrectly specified for one identifier.

C2819 (E) Illegal #pragma abs16 declaration

The **#pragma abs16** specification is illegal.

C2820 (E) Illegal #pragma abs16 symbol type

An identifier type that specifies **#pragma abs16** is illegal.

C2821 (E) Global variable "variable name" in #pragma global_register already declared

The variable that specifies **#pragma global_register** has already been specified.

C2822 (E) Illegal register "register" in #pragma global_register

The register that specifies **#pragma global_register** is illegal.

C2823 (E) Illegal #pragma global_register declaration

The specification of **#pragma global_register** is illegal.

C2824 (E) Illegal #pragma global_register type

A variable that cannot specify **#pragma global_register** exists.

C3000 (F) Statement nest too deep

The nesting level of an **if**, **while**, **for**, **do**, and **switch** statements exceeds the limit.

C3006 (F) Too many parameters

The number of parameters in a function declaration or a function call exceeds the limit.

C3007 (F) Too many macro parameters

The number of parameters in a macro definition or a macro call exceeds the limit.

C3008 (F) Line too long

After a macro expansion, the length of a line exceeds the limit.

C3009 (F) String literal too long

The length of string literal exceeds 32766 characters. The length of string literal is 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 the limit.

C3014 (F) For nest too deep

The nesting level of a **for** statement exceeds the limit.

C3015 (F) Symbol table overflow

The number of symbols to be generated by the compiler exceeds the limit.

C3016 (F) Internal label overflow
The number of internal labels to be generated by the compiler exceeds the limit.

C3017 (F) Too many case labels
The number of **case** labels in one **switch** statement exceeds the limit.

C3018 (F) Too many goto labels
The number of **goto** labels defined in one function exceeds the limit.

C3019 (F) Cannot open source file "file name"
A source file cannot be opened.

C3020 (F) Source file input error "file name"
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 the limit.

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
An array has more than six dimensions.

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.

C3031 (F) Data size overflow
The size of an array or a structure exceeds the limit of 2147483647 bytes.

C3100 (F) Misaligned pointer access
There has been an attempt to refer or specify using a pointer that has an invalid alignment.

C3201 (F) Object size overflow
The object file size exceeds the limit of 4 Gbytes.

C3203 (F) Assembly source line too long

The assembly source line is too long to output.

C3204 (F) Illegal stack access

The size of a stack to be used in a function (including a local variable area, register save area, and parameter push area to call other functions) or a parameter area to call the function exceeds 2 Gbytes.

C3300 (F) Cannot open internal file

An error occurred due to one of the following causes:

(1) An intermediate file internally generated by the compiler cannot be opened.

(2) A file that has the same file name as the intermediate file already exists.

(3) The number of characters in a path name for a list file specification exceeds the limit of 128 characters.

(4) A file which the compiler uses internally cannot be opened.

C3301 (F) Cannot close internal file

An intermediate file internally generated by the compiler cannot be closed. Make sure the compiler is correctly installed.

C3302 (F) Cannot input internal file

An intermediate file internally generated by the compiler cannot be read. Make sure the compiler is correctly installed.

C3303 (F) Cannot output internal file

An intermediate file internally generated by the compiler cannot be written. 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 accessed.

C3305 (F) Invalid command parameter "option name"

An invalid compiler option is specified.

C3306 (F) Interrupt in compilation

An interrupt generated by a (CNTL) + C command (from a standard input terminal) is detected during compilation.

C3307 (F) Compiler version mismatch

File versions in the compiler do not match the other file versions. Refer to the Install Guide for the installation procedure, and reinstall the compiler.

C3308 (F) Cannot create file "file name"

The compiler cannot create necessary files.

C3320 (F) Command parameter buffer overflow

The command line specification exceeds 4096 characters.

C3321 (F) Illegal environment variable

An error occurred due to one of the following causes:

(1) **SHC_LIB** was not specified.

(2) An execution file path name of the compiler was not specified for **SHC_LIB**.

(3) A file name was specified incorrectly when **SHC_LIB** was specified or the number of characters in a path name exceeds the limit of 118 characters.

(4) Other than SH1, SH2, SH2E, SHDSP, SH3, SH3E, or SH4 is set for the environment variable **SHCPU**.

C4000-C4999 (—) Internal error

An internal error occurred during compilation. Report the error occurrence to your local Hitachi dealer.

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) Unrecognized token**
(I) Unrecognized token (macro)

C5008 **(E) Missing closing quote**
(I) Missing closing quote (macro)

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**
(F) #include statement
(E) #line statement

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 "entity" (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 "entity" 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 "entity" 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 "entity" has no member "member name"**

C5136 **(E) Entity-kind " entity" 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 "entity" may not be initialized**

C5146 **(E) Too many initializer values**

C5147 **(E) Declaration is incompatible with "entity" (declared at line "line number")**

C5148 **(E) Entity-kind "entity" 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 "entity" 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**
(I): Constant
(W): Address 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 "entity" 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 "entity" 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 "entity" (declared at line "line number")**

C5257 **(E) Const entity-kind "entity" 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 "entity" 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**
(W) non-static variable
(E) static variable, typedef, enum member

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 "entity" 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 "entity" is intended**

C5300 **(E) A pointer to a bound function may only be used to call the function**

C5302 **(E) Entity-kind "entity" has already been defined**

C5304 **(E) No instance of entity-kind "entity" 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 "entity" matches the argument list:**

C5309 **(E) More than one instance of constructor " entity" 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 "entity" 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 "entity" requires a definition**

C5330 **(E) Entity-kind "entity" 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 "entity" (declared at line "line number")**

C5338 **(E) More than one instance of overloaded function "entity" has "C" linkage**

C5339 **(E) Class "type" has more than one default constructor**

C5341 **(E) "operatorxxxx" 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 "entity" 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 "entity" provides no initializer for:**

C5367 **(E) Implicitly generated constructor for class "type" cannot initialize:**

C5368 **(W) Entity-kind "entity" defines no constructor to initialize the following:**

C5369 **(E) Entity-kind "entity" has an uninitialized const or reference member**

C5370 **(W) Entity-kind "entity" 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 "entity" matches the argument list**

C5386 **(E) No instance of entity-kind "entity" 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 "entity" 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 "entity" has an operator newxxxx() but no default operator deletexxxx()**

C5400 **(I) Entity-kind "entity" has a default operator deletexxxx() but no operator newxxxx()**

C5401 **(E) Destructor for base class "type" is not virtual**

C5403 **(E) Entity-kind "entity" 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 "entity" returns incomplete type "type"**

C5410 **(E) Protected entity-kind "entity" 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 "entity" is missing**

C5442 **(E) Too few arguments for entity-kind "entity"**

C5443 **(E) Too many arguments for entity-kind "entity"**

C5445 **(E) Entity-kind "entity 1" is not used in declaring the parameter types of entity-kind "entity 2"**

C5449 **(E) More than one instance of entity-kind "entity" 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 "entity"**

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 "entity" (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 "entity " has no tag member named "name 2"**

C5473 **(E) Entity-kind "entity" 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 "entity" redeclared "inline" after being called**

C5481 **(E) Invalid storage class for a template declaration**

C5484 **(E) Invalid explicit instantiation declaration**

C5485 **(E) Entity-kind "entity" is not an entity that can be instantiated**

C5486 **(E) Compiler generated entity-kind "entity" cannot be explicitly instantiated**

C5487 **(E) Inline entity-kind "entity" cannot be explicitly instantiated**

C5488 **(E) Pure virtual entity-kind "entity" cannot be explicitly instantiated**

C5489 **(E) Entity-kind "entity" cannot be instantiated -- no template definition was supplied**

C5490 **(E) Entity-kind "entity" cannot be instantiated -- it has been explicitly specialized**

C5493 **(E) No instance of entity-kind "entity" 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 "entity" 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 "entity" 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 "entity" (declared at line "line number")**

C5542 **(F) Could not create instantiation request file "name"**

C5543 **(E) Non-arithmetic operation not allowed in nontype template argument**

C5544 **(E) Use of a local type to declare a nonlocal variable**

C5545 **(E) Use of a local type to declare a function**

C5546 **(E) Transfer of control bypasses initialization of:**

C5548 **(E) Transfer of control into an exception handler**

C5549 **(I) Entity-kind "entity" is used before its value is set**

C5550 **(W) Entity-kind "entity" was set but never used**

C5551 **(E) Entity-kind "entity" cannot be defined in the current scope**

C5552 **(W) Exception specification is not allowed**

C5553 **(W) External/internal linkage conflict for entity-kind "entity" (declared at line "line number")**

C5554 **(W) Entity-kind "entity" 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 "entity 1" does not match "entity 2" -- virtual function override intended?**

C5611 **(W) Overloaded virtual function "entity 1" is only partially overridden in entity-kind "entity 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**

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 "entity" (declared at line "line number") cannot be inlined**

C5679 **(I) Entity-kind "entity" 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 "entity 1" within class "entity 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) Redeclaration of entity-kind "entity" 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 "entity" 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 "entity 1" conflicts with using-declaration of entity-kind "entity 2"**

C5735 **(E) Using-declaration of entity-kind "entity 1" conflicts with entity-kind "entity 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 "entity" ignored**

C5742 **(E) Entity-kind "entity" has no actual member "name 2"**

C5750 **(E) Entity-kind "entity" (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 "entity"**

C5753 **(E) A template-id is not allowed**

C5754 **(E) A class-qualified name is not allowed**

C5755 **(E) Entity-kind "entity" may not be redeclared in the current scope**

C5756 **(E) Qualified name is not allowed in namespace member declaration**

C5757 **(E) Entity-kind "entity" is not a type name**

C5761 **(E) Typename may only be used within a template**

C5766 **(W) Exception specification for virtual entity-kind "entity 1" is incompatible with that of overridden entity-kind "entity 2"**

C5767 **(W) Conversion from pointer to smaller integer**

C5768 **(W) Exception specification for implicitly declared virtual entity-kind "entity 1" is incompatible with that of overridden entity-kind "entity 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 "entity"**

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 "entity"**

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 "entity" 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 "entity" 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 "entity 1" must precede the first use of entity-kind "entity 2"**

C5790 **(E) Explicit specialization is not allowed in the current scope**

C5791 **(E) Partial specialization of entity-kind "entity" is not allowed**

C5792 **(E) Entity-kind "entity" is not an entity that can be explicitly specialized**

C5793 **(E) Explicit specialization of entity-kind "entity" must precede its first use**

C5794 **(W) Template parameter "template parameter" may not be used in an elaborated type specifier**

C5795 **(E) Specializing entity-kind "entity" 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 "entity" (declared at line "line number"):**

C5806 **(W) Omission of exception specification is incompatible with entity-kind "entity" (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 "entity" has a const member**

C5810 **(E) Uninitialized base class "type" has a const member**

C5811 **(E) Const entity-kind "entity" 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 "entity 1" and entity-kind "entity 2" could be used**

C5825 **(W) Virtual inline entity-kind "entity" was never defined**

C5826 **(W) Entity-kind "entity" 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 "entity" 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 "entity"**

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 "entity 1" is not used in template argument list of entity-kind "entity 2"**

C5843 **(E) The type of partial specialization template parameter entity-kind "entity" 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 "entity"**

C5846 **(E) This partial specialization would have been made the instantiation of entity-kind "entity" 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 **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 "entity"**

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 "entity" is a pure virtual function**

C5859 **(E) Pure virtual entity-kind "entity" 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)**

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 **(E) Embedded C++ does not support templates**

C5876 **(E) Embedded C++ does not support exception handling**

C5877 **(E) Embedded C++ does not support namespaces**

C5878 **(E) Embedded C++ does not support run-time type information**

C5879 **(E) Embedded C++ does not support the new cast syntax**

C5880 **(E) Embedded C++ does not support using-declarations**

C5881 **(E) Embedded C++ does not support "mutable"**

C5882 **(E) Embedded C++ does not support multiple or virtual inheritance**

C5885 **(E) "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 "entity" 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 "entity" 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 **(E) 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) Redeclaration of entity-kind "entity" 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 "entity"**

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**

C5988 **(E) Invalid pragma declaration**

C5989 **(E) "name" has already been specified by other pragma**

C5990 **(E) Pragma may not be specified after definition**

C5991 **(E) Invalid kind of pragma is specified to this symbol**

12.3 Standard Library Error Messages

For some library functions, if an error occurs during the library function execution, an error code is set in the macro **errno** defined in the header file <errno.h> contained in the standard library. Error messages are defined in the error codes so that error messages can be output. The following shows an example of an error message output program.

Example:

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

main()
{
   FILE *fp;

   fp=fopen("file", "w");
   fp=NULL;

   fclose(fp);                          /* error occurred          */

   printf("%s\n", strerror(errno));     /* print error message     */
}
```

Description:

1. Since the file pointer of NULL is passed to the **fclose** function as an actual parameter, an error will occur. In this case, an error code corresponding to **errno** is set.
2. The **strerror** function returns a pointer of the string literal of the corresponding error message when the error code is passed as an actual parameter. An error message is output by specifying the output of the string literal of the **printf** function.

Table 12.1 List of Standard Library Error Messages

Error No.	Error Message/Explanation	Functions to Set Error Code
1100 (ERANGE)	`Data out of range` An overflow occurred.	atan, cos, sin, tan, cosh, sinh, tanh, exp, fabs, frexp, ldexp, modf, ceil, floor, strtol, atoi, fscanf, scanf, sscanf, atol
1101 (EDOM)	`Data out of domain` Results for mathematical parameters are not defined.	acos, asin, atan2, log, log10, sqrt, fmod, pow
1102 (EDIV)	`Division by zero` Division by zero was performed.	divbs, divws, divls, divbu, divwu, divlu
1104 (ESTRN)	`Too long string` The length of string literal exceeds 512 characters.	strtol, strtod, atof, atoi, atol
1106 (PTRERR)	`Invalid file pointer` 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, atol, atoi
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 3 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, sscanf, atof
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

Table 12.1 List of Standard Library Error Messages (cont)

Error No.	Error Message/Explanation	Functions to Set Error Code
1250 (EDBLO)	`Overflow out of double` A double-type decimal value is out of range (overflow).	strtod, fscanf, scanf, sscanf, atof
1260 (EDBLU)	`Underflow out of double` A double-type decimal value is out of range (underflow).	strtod, fscanf, scanf, sscanf, atof
1270 (ELDBLO)	`Overflow out of long double` A long double-type decimal value is out of range (overflow).	fscanf, scanf, sscanf
1280 (ELDBLU)	`Underflow out of long double` A long double-type decimal value is out of range (underflow).	fscanf, scanf, sscanf
1300 (NOTOPN)	`File not open` The file is not open.	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, 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 file, input file was issued for an output function.	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 command line options are not correct.
Check and correct the command line 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 system.

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.

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.

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, DSP instruction, extended instruction, or assembler directive that reserves data is in the stack section.
Remove, from the stack section, the executable instruction, DSP instruction, extended 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

Illegal delay slot instruction placed following delayed branch instruction.
Change the order of the instructions so that the illegal delay slot instruction does not immediately follow a delayed branch instruction.

151 (E) ILLEGAL EXTENDED INSTRUCTION POSITION

Extended instruction placed following a delayed branch instruction.
Place an executable instruction following the delayed branch instruction.

152 (E) ILLEGAL BOUNDARY ALIGNMENT VALUE

Illegal boundary alignment value specified for a section including extended instructions.
Specify 2 or a larger multiple of 2 as a boundary alignment value.

160 (E) REPEAT LOOP NESTING

Another REPEAT is located between a REPEAT and its end address.
Correct the REPEAT location.

161 (E) ILLEGAL START ADDRESS FOR REPEAT LOOP

No executable or DSP instructions are located between a REPEAT and the start address.
Use one or more executable or DSP instructions between the REPEAT and the start address.

162 (E) ILLEGAL DATA BEFORE REPEAT LOOP

Illegal data is found immediately before the loop specified by a REPEAT instruction.
If an assembler directive is located before the loop, correct the directive. If a literal pool is located before the loop, use a .NOPOOL directive to prevent the literal pool output.
When three or fewer instructions are to be repeated, an executable or DSP instruction must be located before the loop.

200 (E) UNDEFINED SYMBOL REFERENCE

Undefined symbol reference.
Define the symbol.

201 (E) ILLEGAL SYMBOL OR SECTION NAME

Reserved word 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 or comment.

304 (E) LACKING OPERANDS

Too few operands.
Correct the operands.

307 (E) ILLEGAL ADDRESSING MODE

Illegal addressing mode in operand.
Correct the operand.

308 (E) SYNTAX ERROR IN OPERAND

Syntax error in operand.
Correct the operand.

309 (E) FLOATING POINT REGISTER MISMATCH

A double-precision floating-point register is specified for a single-precision operation, or a single-precision floating-point register is specified for a double-precision operation.
Correct the operation size or the floating-point register.

350 (E) SYNTAX ERROR IN SOURCE STATEMENT (<mnemonic>)

There are syntax error(s) in the DSP instruction statement.
Correct the source statement.

351 (E) ILLEGAL COMBINATION OF MNEMONICS (<mnemonic>, <mnemonic>)

Illegal combination of DSP operation instruction is specified.
Correct the combination of DSP operation instructions.

352 (E) ILLEGAL CONDITION (<mnemonic>)

Illegal condition for DSP operation instruction is specified.
Delete the condition or change the DSP operation instruction.

353 (E) ILLEGAL POSITION OF INSTRUCTION (<mnemonic>)

The DSP operation instruction is specified in an illegal position.
Specify the DSP operation instruction in the correct position.

354 (E) ILLEGAL ADDRESSING MODE (<mnemonic>)

The addressing mode of the DSP operation instruction is illegal.
Correct the operand.

355 (E) ILLEGAL REGISTER NAME (<mnemonic>)

The register name of the DSP operation instruction is illegal.
Correct the register name.

357 (E) ILLEGAL COMBINATION OF MNEMONICS (<mnemonic>)

An illegal data transfer instruction is specified.
Correct the data transfer instruction.

371 (E) ILLEGAL COMBINATION OF MNEMONICS (<mnemonic>, <mnemonic>)

The combination of data transfer instructions is illegal.
Correct the combination of data transfer instructions.

372 (E) ILLEGAL ADDRESSING MODE (<mnemonic>)

An illegal addressing mode for the data transfer instruction operand is specified.
Correct the operand.

373 (E) ILLEGAL REGISTER NAME (<mnemonic>)

An illegal register name for the data transfer instruction is specified.
Correct the register name.

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

Attempt to perform multiplication, division, or logic operation on relative-address value.
Correct the expression.

406 (E) ILLEGAL OPERAND

An expression is specified at the location where floating-point data must be specified.
Specify floating-point data.

407 (E) MEMORY OVERFLOW

Memory overflow during expression calculation.
Simplify the expression.

408 (E) DIVISION BY ZERO

Attempt to divide by 0.
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 specified as shift value.
Correct the expression.

450 (E) ILLEGAL DISPLACEMENT VALUE

Illegal displacement value. (Negative value is specified.)
Correct the displacement value.

452 (E) ILLEGAL DATA AREA ADDRESS

PC-relative data transfer instruction specifies illegal address for data area.
Access a correct address according to the instruction operation size. (4-byte boundary for MOV.L and MOVA, and 2-byte boundary for MOV.W.)

453 (E) LITERAL POOL OVERFLOW

More than 510 extended instructions exist that have not output literals.
Output literal pools using .POOL.

460 (E) ILLEGAL SYMBOL

A label which does not reference a forward position, an undefined symbol, or a symbol other than a label is specified as an operand of a REPEAT, or the start address comes after (which means at a higher address than) the end address.
Correct the operand.

461 (E) SYNTAX ERROR IN OPERAND

Illegal operand.
Correct the operand.

462 (E) ILLEGAL VALUE IN OPERAND

The distance between a REPEAT and the label is out of range.
Correct the location of the REPEAT or the label.

463 (E) NO INSTRUCTION IN REPEAT LOOP

No instruction is found in a REPEAT loop, or no instruction is found at the end address.
Write an instruction between the start and end addresses, or specify an address storing an instruction as the end address.

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.

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, an import symbol, or relative-address symbol) specified in operand.
Correct the operand.

518 (E) INVALID IMPORT SYMBOL

Symbol declared for import 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 .NOPOOL DIRECTIVE POSITION

.NOPOOL placed at illegal position.
Place .NOPOOL following a delayed branch instruction.

522 (E) ILLEGAL .POOL DIRECTIVE POSITION

.POOL placed following a delayed branch instruction.
Place an executable instruction following the delayed branch instruction.

523 (E) ILLEGAL OPERAND

Illegal .LINE directive operand.
Correct the operand.

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 character string has more than 255 characters.
The character strings 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 5

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.

600 (E) INVALID CHARACTER

Illegal character.
Correct it.

601 (E) INVALID DELIMITER

Illegal delimiter character.
Correct it.

602 (E) INVALID CHARACTER STRING FORMAT

Character string error.
Correct it.

603 (E) SYNTAX ERROR IN SOURCE STATEMENT

Source statement syntax error.
Check and correct 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 in the name field of the .MACRO directive.

612 (E) ILLEGAL MACRO NAME

Macro name error (.MACRO directive).
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 formal parameters repeated in formal parameter declaration in macro definition (.MACRO directive).
Correct the formal parameters.

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

.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 formal parameter in a macro body (between .MACRO and .ENDM directives).
Correct the formal parameter.
When there is an error in a formal parameter 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 missing macro processing exclusion parenthesis.

623 (E) SYNTAX ERROR IN STRING FUNCTION

Syntax error in character string manipulation function.
Check and correct the character string manipulation function.

624 (E) MACRO PARAMETERS MISMATCH

Too many numbers of macro parameters for positional specification in macro call.
Correct the number of macro parameters.

631 (E) END DIRECTIVE MISMATCH

Terminating preprocessor directive does not agree with matching directive.
Check and correct the preprocessor directives.

640 (E) SYNTAX ERROR IN OPERAND

Syntax error in conditional assembly directive operand.
Check and correct the whole source statement.

641 (E) INVALID RELATIONAL OPERATOR

Error in conditional assembly directive 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.
Check and correct 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

After .AREPEAT or .AWHILE expansion, the number of characters in a line exceeds 8,192 characters.
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

Could not open the file specified by .INCLUDE directive.
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.

662 (E) ILLEGAL .END DIRECTIVE POSITION

.END directive appears between .AIF and .AENDI directives.
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) EXPANSION TOO LONG

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) SUCCESSFUL CONDITION .AERROR

Statement including the .AERROR directive was processed in the .AIF condition.
Check and correct the conditional statement so that the .AERROR directive is not processed.

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 character string exceeds 255 characters.
The character strings to specify to the operand of .ASSIGNC directive, .DEFINE directive, and character manipulating functions (.LEN, .INSTR, .SUBSTR) must have 255 or less characters.

700 (W) ILLEGAL VALUE IN OPERAND (<mnemonic>)

The operand value of the DSP operation instruction is out of range.
Correct the operand value.

701 (W) MULTIPLE REGISTER IN DESTINATION (<mnemonic>, <mnemonic>)

The same register is specified as multiple destination operands of the DSP instruction.
Specify the register correctly.

702 (W) ILLEGAL OPERATION SIZE (<mnemonic>)

The operation size of the DSP operation instruction or the data transfer instruction is illegal.
Cancel or correct the operation size.

703 (W) MULTIPLE REGISTER IN DESTINATION (<mnemonic>, <mnemonic>)

The same register is specified as the destination registers of the DSP operation instruction and data transfer instruction.
Specify the register correctly.

800 (W) SYMBOL NAME TOO LONG

A symbol exceeds 32 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.

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 of integer constant.
Correct the notation.
The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.

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.
The assembler ignores the label.

813 (W) SECTION ATTRIBUTE MISMATCH

A different section type is specified on section restart, 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 restart.
The specification of starting section remains valid.

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) ILLEGAL DATA AREA ADDRESS

Illegal allocation of data or data area.
Locate the word data or data area on an even address. Locate the long-word or single-precision data or data area on an address of a multiple of 4. Locate the double-precision data or data area on an address of a multiple of 8.
The assembler corrects the location of the data or data area according to its specified size.

817 (W) ILLEGAL BOUNDARY VALUE

A boundary alignment value less than 4 specified for a code section.
The specification is valid, but if an executable instruction, DSP instruction, or extended instruction is located at an odd address, warning 882 occurs.
Special care must be taken when specifying 1 for code section boundary alignment value.

818 (W) COMMANDLINE OPTION MISMATCH FOR FLOATING DIRECTIVE

When the CPU type is SH-2E or SH-3E, the **round=nearest** or **denormalize=on** option is specified.
Change the specification in the **round** or **denormalize** option.
The assembler creates the object code according to the specification in the **round** or **denormalize** option.

825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION

An executable instruction, DSP instruction, extended instruction, or assembler directive that reserves data is in dummy section.
Remove, from the dummy section, the executable instruction, DSP instruction, extended instruction, or assembler directive that reserves data.
The assembler ignores the executable instruction, DSP instruction, extended instruction, or assembler directive that reserves data in dummy section.

826 (W) ILLEGAL PRECISION

The floating-point constant does not have the same precision specified with the operation size.
Correct the operation size or the precision type of the floating-point constant.
The assembler assumes the precision specified with the operation size.

832 (W) MULTIPLE 'P' DEFINITIONS

Symbol P already defined before 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 the executable instruction.
Correct the value.
The assembler generates object code with a value corrected to be within range.

836 (W) ILLEGAL CONSTANT SIZE

Illegal notation of integer constant.
Correct the notation.
The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.

837 (W) SOURCE STATEMENT TOO LONG

After .AREPEAT or .AWHILE expansion, the number of characters in a line exceeds 8192 characters.
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 character strings and comments, or the **sjis**, **euc**, or **latin1** option is not specified.
Specify the shift JIS code, EUC code, LATIN1 code in character strings or comments. Or specify the **sjis**, **euc**, or **latin1** option.

839 (W) ILLEGAL FIGURE IN OPERAND

Fixed-point data having six or more digits is specified in word size, or that having 11 or more digits is specified in long-word size.
Reduce the digits to the limit.

840 (W) OPERAND OVERFLOW

Floating-point data overflowed.
Modify the value.
The assembler assumes +∞ when the value is positive and –∞ when negative.

841 (W) OPERAND UNDERFLOW

Floating-point data underflowed.
Modify the value.
The assembler assumes +0 when the value is positive and –0 when negative.

842 (W) OPERAND DENORMALIZED

Denormalized numbers are specified for floating-point data.
Check and correct the floating-point data.
The assembler creates the object code according to the specification (sets denormalized numbers).

850 (W) ILLEGAL SYMBOL DEFINITION

Symbol specified in label field.
Remove the symbol.

851 (W) MACRO SERIAL NUMBER OVERFLOW

Macro serial number exceeded 99,999.
Reduce the number of macro calls.

852 (W) UNNECESSARY CHARACTER

Characters appear after the operands.
Correct the operand(s).

854 (W) .AWHILE ABORTED BY .ALIMIT

Expansion count has reached the maximum value specified by .ALIMIT directive, and expansion has been terminated.
Check and correct the condition for iterated expansion.

870 (W) ILLEGAL DISPLACEMENT VALUE

Illegal displacement value.
Either the displacement value is not an even number when the operation size is word, or the displacement value is not a multiple of 4 when the operation size is long word.
Take account of the fact that the assembler corrects the displacement value.
The assembler generates object code with the displacement corrected according to the operation size.
For a word size operation the assembler discards the low order bit of the displacement to create an even number, and for a long-word-size operation the assembler discards the two low order bits of the displacement to create a multiple of 4.

871 (W) PC RELATIVE IN DELAY SLOT

Executable instruction with PC relative addressing mode operand is located following delayed branch instruction.
Take account of the fact that the value of PC is changed by a delayed branch instruction.
The assembler generates object code exactly as specified in the program.

874 (W) CANNOT CHECK DATA AREA BOUNDARY

Cannot check data area boundary for PC-relative data transfer instructions.
Note carefully the data area boundary at linkage process.
The assembler outputs this message when a data transfer instruction is included in a relative-address section, or when an import symbol is used to indicate a data area.

875 (W) CANNOT CHECK DISPLACEMENT SIZE

Cannot check displacement size for PC-relative data transfer instructions.
Note carefully the distance between data transfer instructions and data area at linkage.
The assembler outputs this message when a data transfer instruction is included in a relative-address section, or when an import symbol is used to indicate a data area.

876 (W) ASSEMBLER OUTPUTS BRA INSTRUCTION

The assembler automatically outputs a BRA instruction.
Specify a literal pool output position using .POOL, or check that the program to which a BRA instruction is added can run normally.
When a literal pool output location is not available, the assembler automatically outputs literal pool and a BRA instruction to jump over the literal pool.

880 (W) .END NOT FOUND

No .END in the program.
Insert an .END.

881 (W) ILLEGAL DIRECTIVE IN REPEAT LOOP

An illegal assembler directive was found in a .REPEAT loop.
Delete the directive.
If a directive that reserves a data item or a data area, an .ALIGN directive, or an .ORG directive is used in a .REPEAT loop, the assembler counts the directive as one of the instructions to be repeated.

882 (W) ILLEGAL ADDRESS

The executable instruction and extended instruction are written in an odd address.
Write the executable instruction and extended instruction are written in an even address.

901 (F) SOURCE FILE INPUT ERROR

Source file input error.
Check the hard disk for adequate free space. Create the required free space, e.g. 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 listing file.
Check the hard disk for adequate free space. Create the required free space, e.g. 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, e.g. 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 62,265 sections can be enabled.
When debugging information is not output, up to 65,274 sections can be enabled.
Subdivide the program.

933 (F) ILLEGAL ENVIRONMENT VARIABLE

The specified target CPU is incorrect.
Correct the target CPU.

935 (F) SUBCOMMAND FILE INPUT ERROR

Subcommand file input error.
Check the hard disk for adequate free space. Create the required free space, e.g. by deleting unnecessary files.

950 (F) MEMORY OVERFLOW

Insufficient memory.
Subdivide the source program.

951 (F) LITERAL POOL OVERFLOW

The number of literal pools exceeds 100,000.
Subdivide the source program.

952 (F) LITERAL POOL OVERFLOW

Literal pool capacity overflow.
Insert unconditional branch before overflow.

953 (F) MEMORY OVERFLOW

Insufficient memory.
Subdivide the source program.

954 (F) LOCAL BLOCK NUMBER OVERFLOW

The number of local blocks that are valid in the local label exceeds 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, e.g. by deleting unnecessary files.

957 (F) MEMORY OVERFLOW

Insufficient memory.
Subdivide the source program.

958 (F) MEMORY OVERFLOW

Insufficient memory.
Subdivide the source program.

964 (F) MEMORY OVERFLOW

Insufficient memory (Unable to process the symbol information).
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 code (Error level) Error message

Error details

There are five different error levels, corresponding to different degrees of seriousness.

Error Code	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; main processing is interrupted.
L3000–L3999 P3000–P3999	(F)	Fatal	Processing is interrupted.
L4000 - P4000 -	(-)	Internal	Processing is interrupted.

Error codes beginning with L are optimizing linkage editor output messages.

Error codes beginning with P are prelinker output messages. Output of errors with numbers beginning with 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 **optimization** optimization.

L0002 (I) Symbol “symbol” created by optimization “optimization”

The symbol named **symbol** was created as a result of **optimization** optimization.

L0003 (I) “file”-“symbol” moved to “section” by optimization

As a result of variable_access optimization, the symbol **symbol** in the file **file** was moved.

L0004 (I) “file”-“symbol” deleted by optimization

As a result of symbol_delete optimization, the symbol **symbol** in the file **file** was deleted.

L0005 (I) The offset value from the symbol location has been changed by optimization: “file”- “section”- “symbol±offset”

The offset value has been changed because the range allowed for symbol±offset has been changed by optimization. Check for any problem. To prevent the offset value from changing, do not specify option **goptimize** when assembling “file.”

L0100 (I) No inter-module optimization information in “file”

No inter-module optimization information was found in the file **file**. Inter-module optimization is not performed on “file”. On compiling and assembly, the goptimize option should be specified.

L0101 (I) No stack information in “file”

No stack information was found in file **file**. **file** may possibly be an assembler output file or a SYSROF-> ELF conversion file. The contents of this file will not be contained in the stack information file output by the optimizing linkage editor.

P0200 (I) “instance” no longer needed in “file”

An unused instance **instance** exists in file **file**.

P0201 (I) “instance” assigned to file “file”

The instance **instance** was assigned to file **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 **instance** was assigned to the file **file**.

L1000 (W) Option “option” ignored

The option **option** is invalid, and has been ignored.

L1001 (W) Option “option 1” is ineffective without option “option 2”

option 2 is necessary when specifying **option 1**. **option 1** has been ignored.

L1002 (W) Option “option 1” cannot be combined with option “option 2”

option 1 and **option 2** cannot be specified simultaneously. **option 1** has been ignored.

L1003 (W) Divided output file cannot be combined with option “option”

When **option** is specified, division of the output file cannot be specified. The **option** specification is ignored. The first input file name is used as an output file name.

L1004 (W) Fatal level message cannot be changed to other level : “number”

The level of a fatal level message cannot be changed. The specification of **number** is ignored. Only errors at information/warning/error level can be changed using change_message.

L1005 (W) Subcommand file terminated with end option instead of exit option

There is no processing specification following the end option. The exit option has been assumed.

L1006 (W) Options following exit option ignored

All options following the exit option have been ignored.

L1007 (W) Duplicate option : “option”

Duplicate specifications of **option** were found. Only the last specification is effective.

L1010 (W) Duplicate file specified in option “option” : “file”

option was used to specify the same file twice. The second specification has been ignored.

L1011 (W) Duplicate module specified in option “option” : “module”

option was used to specify the same module twice. The second specification has been 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 has been 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. Specification of **name** has been 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.

L1102 (W) Symbol “symbol” redefined in option “option”

The symbol specified by **option** has already been defined. Processing is continued without change.

L1103 (W) Invalid address value specified in option “option” : “address”
address specified by **option** is invalid. The **address** specification is ignored.

L1110 (W) Entry symbol “symbol” in entry option conflicts
A symbol other than the symbol **symbol** specified by the entry option is specified as the entry symbol at compile or assembly time. The option specification is given priority.

L1120 (W) Section address is not assigned to “section”
There is no specification of the address of **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 the section specified by the start option conflicts with memory alignment requirements. The section address is modified to conform with alignment.

L1130 (W) Section attribute mismatch in rom option : “section 1,section 2”
The attributes and alignment of **section 1** and **section 2** specified by the rom option are different. The larger value is effective as the 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 in a different record type.

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.

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 debug information in the input files. The debug option has been ignored. Check whether the debug 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 has been ignored. If all input files are assembler output files or SYSROF->ELF conversion files, the stack option is invalid.

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"

Debug information in **file** is not dwarf2. The debug information has been deleted.

L1320 (W) Duplicate symbol "symbol" in "file"

The symbol **symbol** is duplicated. The symbol in the first file input is given preference.

L1321 (W) Entry symbol "symbol" in "file" conflicts

Multiple object files containing entry symbol definitions were input. Only the symbol in the first file input is effective.

L1322 (W) Section alignment mismatch : "section"

Sections with the same name but different alignments were input. Only the larger alignment specification is effective.

L1323 (W) Section attribute mismatch : "section"

The same section name but with different attributes was input. If they are an absolute section and relative section, the section is treated as an absolute section. If the read/write attributes are different, both are allowed.

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 too deep; 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 at compile time cannot be checked. To check the consistency of stack size specified by the stack option at compile time, specify the goptimize option.

L1501 (W) Stack size overflow : “stack size”

The stack section size exceeded the **stack size** specified by the stack option at compile time. Either change the option used at compile time, 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 at compile time.

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 both be specified simultaneously.

L2005 (E) Option “option” cannot be specified while processing “process”

option cannot be specified for **process**.

L2006 (E) Option “option 1” is ineffective without option “option 2”

The option **option 1** requires that the option **option 2** be specified.

L2010 (E) Option “option” requires parameter

The option **option** requires that a parameter be specified.

L2011 (E) Invalid parameter specified in option “option” : “parameter”

An invalid parameter was specified for **option**.

L2012 (E) Invalid number specified in option “option” : “value”

An invalid value was specified for **option**. Check the range of valid values.

L2013 (E) Invalid address value specified in option “option” : “address”

The address **address** specified in **option** is invalid. A hexadecimal address between 0 and FFFFFFFF should be specified.

L2014 (E) Illegal symbol/section name specified in “option” : “name”

The section or symbol name specified in **option** uses an illegal character. Only alphanumerics, the underscore (_), and the dollar sign ($) may be used in section/symbol names (the leading character cannot be a number).

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**, **file (type)** that cannot be processed was input.

L2211 (E) Invalid input file type specified while processing “process” : “file (type)”

file (type) that cannot be processed was input during **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”

A section of the same name but with different type (initial values present/absent) was 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 to 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**. **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.

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 and assembly lists is placed in 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 and assembly lists.

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 and assembly lists.

L2340 (E) Symbol name in section “section” is too long

The number of characters in symbols in **section** specified by fsymbol exceeded 8174 characters.

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.

P2500 (E) Cannot find library file : “file”

The 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. Check whether, for the file using **instance**, **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**. Check whether, for the file using **instance**, **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**. Check to 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”

file cannot be written to. 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

A temporary file cannot be written to. 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 of 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 **(cntl)+C** 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 16 Limitations

16.1 Limitations of the Compiler

Table 16.1 shows the limitations of the compiler. Source programs must fall within these limitations.

Table 16.1 Limitations of the Compiler

Classification	Item	Limit
Invoking the compiler	Total number of macro names that can be specified using the **define** option	None
	Length of file name (characters)	*None* (depends on the OS)
Source programs	Length of one line (characters)	32768
	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 a **#include** directive	*None*
	Total number of macro names that can be specified in a **#define** directive	None
	Number of parameters that can be specified using a macro definition or a macro call operation	*None*
	Number of expansions of a macro name	*None*
	Nesting levels of **#if**, **#ifdef**, **#ifndef**, **#else**, or **#elif** directive	*None*
	Total number of operators and operands that can be specified in a **#if** or **#elif** 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
	Size of arrays and structures	2147483647 bytes

Table 16.1 Limitation of the Compiler (cont)

Classification	Item	Limit
Statements	Nesting levels of compound statements	*None*
	Nesting levels of statement in a combination of repeat (**while**, **do**, and **for**) and select (**if** and **switch**) statements	32
	Number of **goto** labels that can be specified in one function	511
	Number of **switch** statements	256
	Nesting levels of **switch** statements	16
	Number of **case** labels in a single **switch** statement	511
	Nesting levels of **for** statements	16
Expressions	Character array length	32766
	Number of parameters that can be specified using a function definition or a function call operation	63[*2]
	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 **open** function	20

Notes: 1. The items in which the limit is changed by this version is shown in italics.
2. For non-static function members, 62.

16.2 Limitations of the Assembler

Table 16.2 shows the limitations of the assembler.

Table 16.2 Limitations of the Compiler

Item	Limit
Length of one line (characters)	*8192*
Character constants	Up to 4
Symbol character arrays	*None*[*2]
Number of symbols	*None*
Number of externally referenced symbols	*None*
Number of externally defined symbols	*None*
Maximum size for a section	Up to H’FFFFFFFF bytes
Number of sections	H'FEF2 bytes (with debugging information) or H'FEFB bytes (without debugging information)
File include	Up to 30 levels of nesting
Character array length	255
Length of file name (characters)	*None* (depends on the OS)

Notes: 1. The items in which the limit is changed by this version is shown in italics.
2. For a preprocessor variable name, DEFINE replacement symbol name, macro name or macro (temporary) argument name, it is limited to 32 characters.

Section 17 Notes on Version Upgrade

17.1 Notes on Version Upgrade

This section describes notes when the version is upgraded from the earlier version (SuperH RISC engine C/C++ Compiler Package Ver. 5.x or lower).

17.1.1 Guaranteed Program Operation

When the version is upgraded and program is developed, operation of the program may change. When the program is created, note the followings and sufficiently test your program.

1. **Programs Depending on Execution Time or 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 with the program execution time and peripherals such as the I/O, or 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 version when two or more side effects are included in one expression.

   Example

```
a[i++]=b[i++];      /* i increment order 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. Operations may change depending on the 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 or Type Inequality**

   When a variable is not initialized or the parameter or return value types do not match between the calling and called functions, an incorrect value is accessed. Operations may change depending on the version.

   Example

   file 1:

```
int f(double d)
{
     :
}
```

   file 2:

```
int g(void)
{
    f(1);
}
```

   The parameter of the caller function is the int type, but the parameter of the callee function 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 versions in mind.

17.1.2 Compatibility with Earlier Version

The following notes cover situations in which the compiler is used to generate a file that is to be linked with files generated by the earlier version or with object files or library files that have been output by the assembler or linkage editor, points to do with the debugger, and several other important points.

1. **Object Format**

   The standard object file format has been changed from SYSROF to ELF. The standard format for debugging information has also been changed to DWARF2.

   When object files (SYSROF) output by the earlier version of the compiler (Ver. 5.x or lower) or assembler (Ver. 4.x or lower) are to be input to the 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- and 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.

2. **Point of Origin for Include Files**

   When an include file specified with a relative directory format was searched for, in the earlier version, the search would start from the compiler's directory. In the new version, the search starts from the directory that contains the source file.

3. **C++ Program**

   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 name of the library function for initial/post processing of the global class object, which is used to set the execution environment, has also been changed. Refer to section 9.2.2, Execution Environment Settings, and modify the name.

4. **Abolition of Common Section (Assembly Program)**

   With the change of the object format, support for a common section has been abolished.

5. **Specification of Entry via .end Directive (Assembly Program)**

   Only an externally defined symbol can be specified with the .end directive.

6. **Inter-module Optimization**

   Object files output by the earlier 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.

17.1.3 Command-line Interface

1. **Rules for Assembler and Optimizing Linkage Editor 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**

   Support for the interactive specification of options has been abolished.

   The inter-module optimizing tool (optlnksh), linkage editor (lnk), librarian (lbr), and object converter (cnvs) of earlier versions have been integrated into an optimizing linkage editor (optlnk). Therefore, command-line specification has changed greatly. Tables 17.1 and 17.2 list the changed commands.

Table 17.1 Changed Linkage Commands

No.	Command Name	V6	V7	Note
1	start	start = section (address)	start = section/address	-
		Abbreviation: st	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)	rename = (before change = after change), (before change = after change), -	The conception of unit has been abolished due to the change in the object format.
		Abbreviation: re	Abbreviation: ren	
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.
12	cpu	Abbreviation: c	Abbreviation: cp	Direct range can be specified.
13	elf/sysrof/sysrofplus	elf/sysrof/sysrofplus	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*

Table 17.1 Changed Linkage Commands (cont)

No.	Command Name	V6	V7	Note
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 format has been abolished.
24	list	list	Abolished	Different from the list option for 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: Invalidated by the change_message option.

Table 17.2 Changed Librarian Commands

No.	Command Name	V2	V7	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 format has been abolished.
10	exit	Abbreviation disabled	Abbreviation enabled	-

17.1.4 Provided Contents

In the SuperH RISC engine C/C++ Compiler Package, the following files have been changed.

1. **Standard Library File**

   To specify any function interface or optimizing option, a standard library generator is provided instead of the conventional standard library files.

2. **Header File**

   Header file _h_c_lib.h has been added to global class object initialization processing routine (_CALL_INIT function) and global class object post processing routine (_CALL_END function). These routines are provided as standard library.

17.1.5 List File Specification

1. **Optimizing Linkage Editor**

   The formats of the conventional linkage map and library lists have been updated.

17.2 Additions and Improvements

17.2.1 Common Additions and Improvements

1. 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: 32,767 -> unlimited
- Number of source program lines: C/C++: 32,767, ASM: 65,535 -> unlimited
- Length of C program string literals: 512 characters -> 32,766 characters
- Length of assembly program line: 255 characters -> 8,192 characters
- Length of subcommand file line: ASM: 300 bytes, optlnk: 512 bytes -> unlimited
- Number of parameters of the optimizing linkage editor ROM option: 64 -> unlimited

2. Hyphens for Directory and File Names

A hyphen (-) can be specified for directory and file names.

3. Specification of Copyright Display

Specifying the logo/nologo option can specify whether or not the copyright output is displayed.

4. Prefix to Error Messages

To support the error-help function in Hitachi Embedded Workshop, a prefix has been added to error messages for the compiler and optimizing linkage editor.

17.2.2 Added and Improved Compiler Functions

1. Added fpscr Option

When the cpu=sh4 option is specified and the fpu option is not specified, whether or not the precision mode in the FPSCR register is guaranteed can be specified before and after a function call.

2. #pragma Extension

A #pragma extension can be described without parenthesis ().

3. Added Intrinsic Functions

The trace function was added.

4. Added Implicit Declarations

_ _HITACHI_ _ and _ _HITACHI_VERSION_ _ are implicitly declared by #define.

5. **Static Label Name**

   The label name for references to a static label by using #pragma inline_asm was changed to _ _$ (name). However, in a linkage list, the name is displayed as _ (name).

6. **Extension and Change of Language Specification**

   — Conditions for errors in the initialization of unions have been tightened.

   Example:

```
union {
    char c [4];
} uu={{'a','b','c'}};
```

   — A structure can now be assigned and declared at the same time.

   Example:

```
struct {
    int a, int b;
} sl;

void test ()
{
    struct S s2=s1;
}
```

   — bool-type data is aligned to four-byte boundaries.

   — Exception processing and template functions are also supported according to the C++ language specification.

   — The C preprocessor supports ANSI/ISO.

17.2.3 Added and Improved Optimizing Linkage Editor Functions

1. **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.

2. **Search Path**

   An environment variable (HLNK_DIR) can be used to specify the several search paths for input files or library files.

3. **Subdividing the Output of Load Modules**

   The output of an absolute load module file can be subdivided.

4. **Changing the Error Level**

   For informational, warning, and error level messages, the error level or the output can be individually changed.

5. **Support for Binary and HEX**

   Binary files can be input and output.

   Intel® HEX-type output can be selected.

6. **Output the Stack Amount Information**

   The stack option can output an information file for the stack analysis tool.

7. **Debugging Information Deletion**

   The strip option can be used to delete only debugging information from either the load module file or the library file.

17.3 Operating Format Converter

17.3.1 Object File Format

The object file format complies with the standard ELF format. The debugging information format also complies with the standard DWARF2 format.

17.3.2 Compatibility with Earlier Versions

1. **Object and Library Files**

   When an object file or library file that has been output by an earlier version of the compiler (Ver. 5.x or lower) or assembler (Ver. 4.x or lower) is to be input to the optimizing linkage editor, it must be converted to 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 file 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 output by earlier versions of the compiler (Ver. 5.x or lower) or assembler.

2. Load Module File

ELF-format load module files can be converted to the format output by the linkage editor of earlier versions (Ver. 6.0 or lower) by using the format converter. Table 17.3 is a list of the object file formats that can be converted.

Table 17.3 Object File Formats that can be Converted from ELF Format

Version Number of Compiler or Assembler		Linkage Editor Specification Option		Object File Format: Object	Object File Format: Debugging Information	Conversion
Compiler	4.x or lower	debug		SYSROF	SYSROF	Enabled
Assembler	3.x or lower	sdebug		SYSROF	SYSROF	Disabled
Compiler Assembler	5.x 4.x	sysrof	debug	SYSROF	SYSROF	Enabled
			sdebug	SYSROF	DWARF1	Disabled
		elf	debug	ELF	DWARF1	Enabled
			sdebug	ELF	DWARF1	Disabled

The format converter outputs a file with a converted object, and the same name as the input file. The input file is saved as <input file name.extension>.bak.

The load module file output by an earlier version of the linkage editor (Ver. 6.0 or lower) cannot be converted to the ELF format.

Load modules with a newly-added feature of the compiler, assembler, or optimizing linkage editor cannot be converted.

17.3.3 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 also be used.

17.3.4 List of Options

In the command line format, uppercase letters indicate the abbreviations. Characters underlined indicate the defaults. When Hitachi Embedded Workshop is used, the option is specified in the option window of the optimizing linkage editor. The format of the dialog menus that correspond to Hitachi Embedded Workshop is as follows:
Tab name [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

A relocatable object file or library file created by an earlier version of the compiler (Ver. 5.x or lower) or assembler (Ver. 4.x or lower) 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 an option in Hitachi Embedded Workshop.

Table 17.4 Options for Converting Object Files or Library Files

	Item	Option	Dialog Menu	Specification
1	Address space[*1] specification	Address_space=<size> <size>:20|24|28|32	-	Address space specification
2	fpu	Fpu	-	With FPU
3	dsp	Dsp	-	With DSP

Note: Options for H8S, H8/300 series. They cannot be used in the SuperH.

Fpu

- Command Line Format

  `Fpu`
- Description

  Specifies that an FPU is available when the `CPU` is an `SH-2E`, `SH-3E`, or `SH-4`.
- Example

  ```
  helfcnv –fpu *.obj *.lib
  ```

  ; All `*.obj` and `*.lib` files in the directory are converted to the `elf` format.

Dsp

- Command Line Format

  `Dsp`
- Description

  Specifies that a DSP unit is available when the `CPU` is an `SH2-DSP` or `SH3-DSP`.
- Example

  ```
  helfcnv –dsp *.obj
  ```

  ; All `*.obj` files in the directory are converted to the `elf` format.

2. Conversion of Load Module Files

ELF-format load module files were converted to object file format output by earlier versions of the linkage editor (Ver. 6.0 or lower). When debugging information has been included in the load module file, the load module after conversion retains the debugging information.

Table 17.5 Options for Converting Load Module File

	Item	Option	Dialog Menu	Specification
1	Specification of conversion format	Sysrof	Output	Converted to the SYSROF format
		Dwarf1	[Type of output file:]	Converted to the ELF/DWARF1 format

Sysrof
Dwarf1

Output [Type of output file:]

- Command Line Format

  Sysrof
  Dwarf1

- Description

  This option 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 optimizing linkage editor specifies the sdebug option, 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.
```

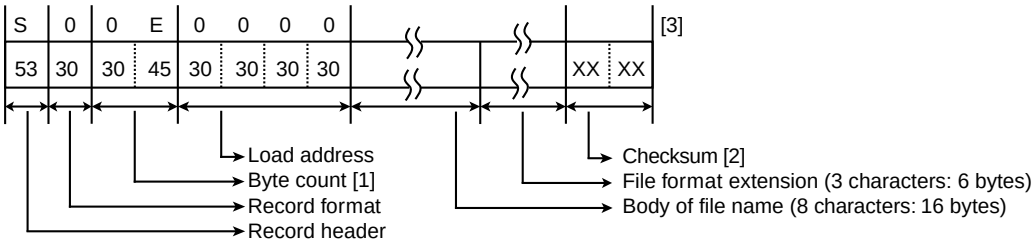
Section 18 Appendix

18.1 S-Type and HEX File Format

This section describes the S-type files and HEX files that are output by the optimizing linkage editor.

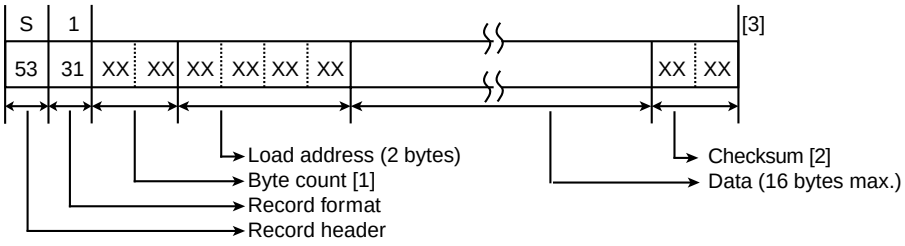
18.1.1 S-Type File Format

(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

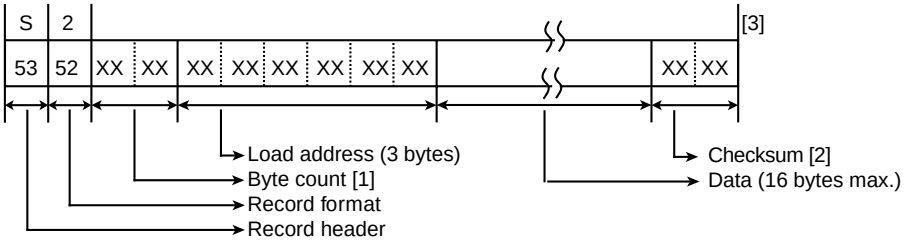
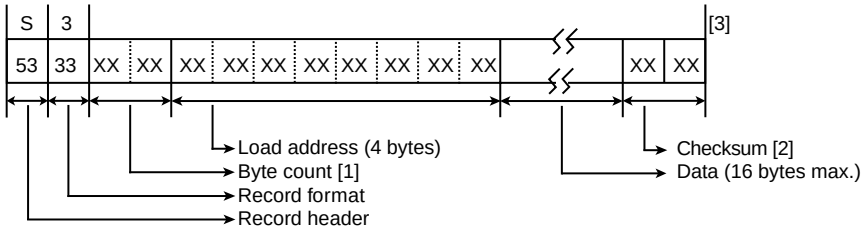


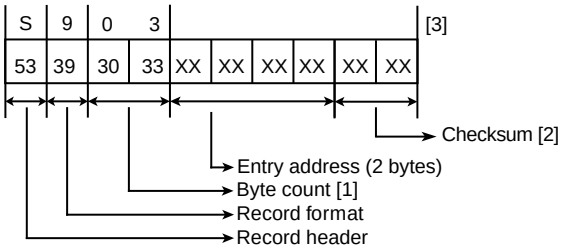
Figure 18.1 S-Type File Format

(iii) When the load address is 1000000 to FFFFFFFF

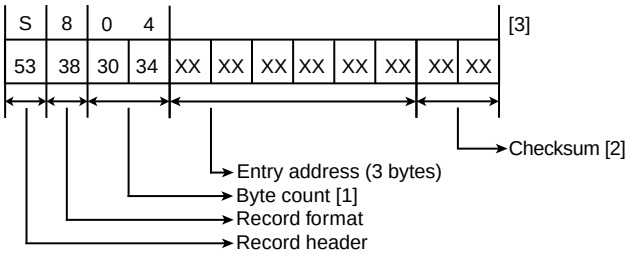


(c) Entry Record (S9, S8, and S7 Records)

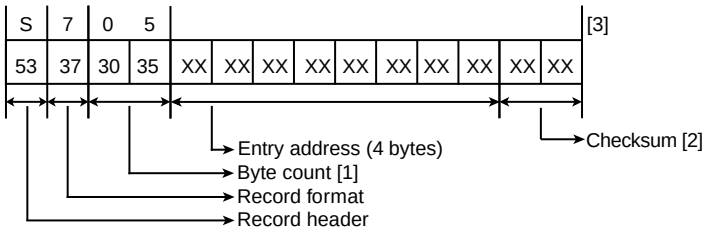
(i) When the entry address is 0 to FFFF



(ii) When the entry address is 10000 to FFFFFF



(iii) When the entry address is 1000000 to FFFFFFFF



Notes: [1] The number of bytes from the load address (or the entry address) to the checksum.
[2] 1's complement of the sum of the byte count and the data between the checksum and the byte count, in byte units.
[3] A line feed character is added immediately after the checksum.

Figure 18.1 S-Type File Format (cont)

18.1.2 HEX File Format

The execution address of each data record is obtained as described below.

- Segment address
  (Segment base address << 4) + (Address offset of the data record)
- Linear address
  (Linear base address << 16) + (Address offset of the data record)

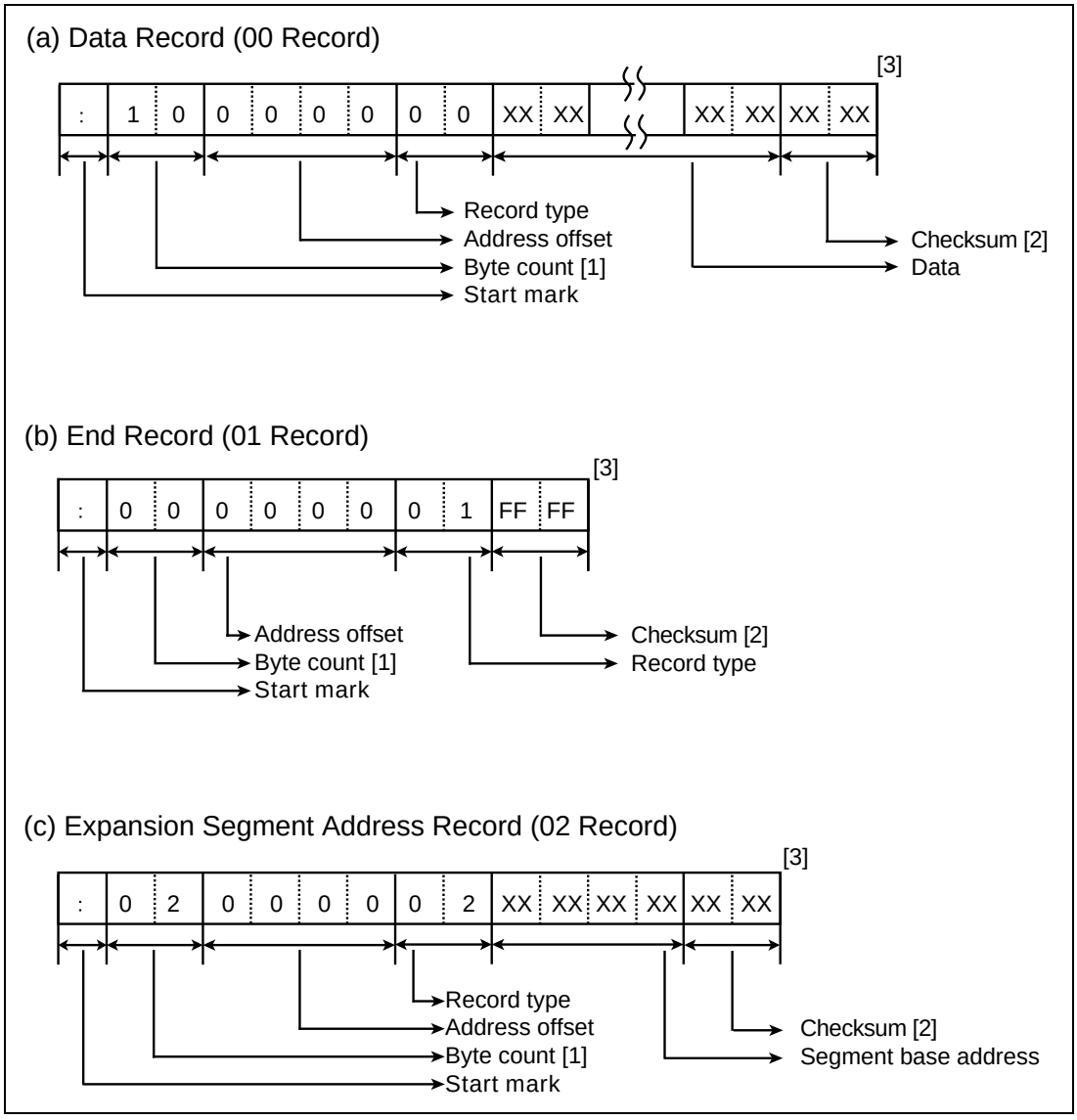
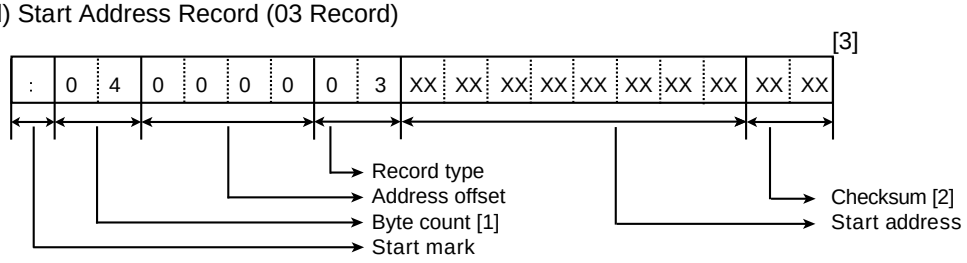
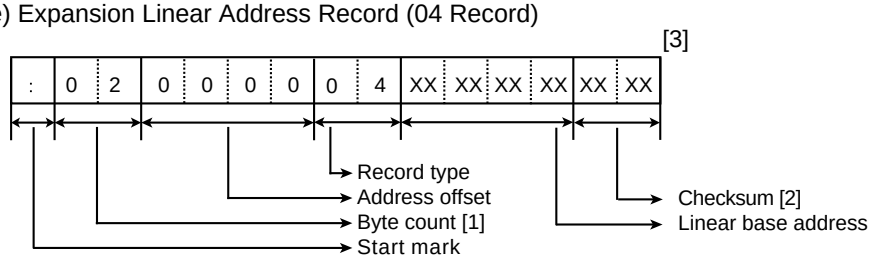


Figure 18.2 HEX File Format

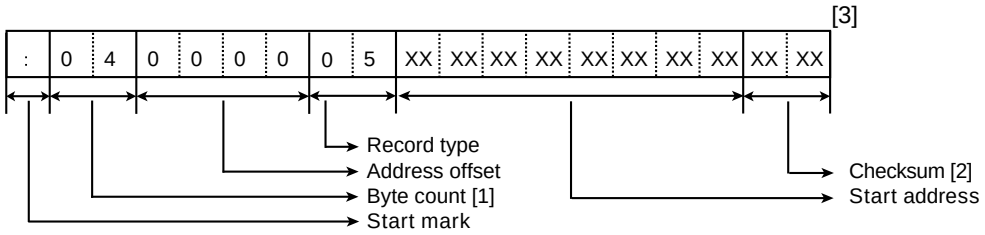
(d) Start Address Record (03 Record)



(e) Expansion Linear Address Record (04 Record)



(f) 32-Bit Start Linear Address Record (05 Record)



Notes: [1] The number of bytes from the record type to the previous bit of the checksum.
[2] 2's complement of the sum of the byte count and the data between the byte count and checksum, in hexadecimal (lower 8 bits are valid).
[3] Line feed is added immediately after the checksum.

Figure 18.2 HEX File Format (cont)

18.2 ASCII Code List

Table 18.1 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

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

HITACHI

T

U

V

W

X

Y

Z