SEGA™

Dreamcast SH4 C Compiler User's Manual

Hitachi Microcomputer Support Software

SH Series C Compiler
User's Manual

HITACHI

HS0700CLCU4SE
Rev. 4.0
10/6/97
Hitachi, Ltd.

Hitachi semiconductor

Preface

This manual explains the facilities and operating procedures for the SH series C compiler. Please read this manual and the related manuals listed below before using the C compiler to fully understand the system. The C compiler translates source programs written in C into relocatable object programs or assembly source programs for Hitachi superH RISC engine family microcomputers (SH1, SH2, SH3, and SH3E).

Features of this compiler system are as follows:

1. generates an object program that can be written to ROM to be installed in a user system.
2. supports an optimization option that increases execution speed of object programs or minimizes program size.
3. supports a debugging-information output function for a C source level debugging or C source analysis using a debugger .
4. selects an assembly source program or relocatable object program and outputs it.

This manual consists of four parts and appendixes. The information contained in each part is summarized below.

1. PART I OVERVIEW AND OPERATIONS

   The overview sections cover C compiler functions and developing procedures.

   The operation sections cover how to invoke the compiler, how to specify optional functions, and how to interprete listings created by the C compiler.
2. PART II C PROGRAMMING

   This part explains the limitations of the C compiler and the special factors in object program execution which should be considered when creating a program.
3. PART III SYSTEM INSTALLATION

   This part explains the object program being written in ROM and memory allocation when installing an object program generated by the C compiler on a system. In addition, specifications of the low-level interface routine must be made by the user when using C language standard I/O library and memory management library.
4. PART IV ERROR MESSAGES

   This part explains the error messages corresponding to compilation errors and the standard library error messages corresponding to run time errors.

This manual describes the SH C compiler that operates on UNIX*[1], or MS-DOS*[2] that runs (operates) on the IBM-PC*[3] and PC compatibles. In this manual, compilers functioning on a UNIX system are referred to as UNIX version and compilers functioning on an MS-DOS system are referred to as PC systems.

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. MS-DOS is an operating system administrated by Microsoft Corporation.
3. IBM PC is a registered trademark of International Business Machines Corporation.

Related Manuals: Refer to the following manuals together with the SH Series C Compiler when creating a program using the C compiler.

SH Series Cross Assembler User's Manual
SH Series Simulator/Debugger User's Manual
Integrated Manager User's Manual
H Series Linkage Editor User's Manual
H Series Librarian User's Manual
E7000 SH7032, SH7034 Emulator User's Manual
E7000 SH7604 Emulator User's Manual
E7000 SH7708 Emulator User's Manual

Refer to the following manuals for details on the SH instruction execution:

SH7000 Series Programming Manual
SH7000/SH7600 Series Programming Manual
SH7700 Series Programming Manual

Contents

Figures

Tables

PART I

OVERVIEW AND OPERATIONS

Section 1 Overview

The SH series C compiler converts source programs written in C to SH series relocatable object programs or assembly source programs.

The C compiler supports the SH1, SH2, SH3, and SH3E microcomputers (collectively referred to as SH).

Figure 1.1 shows C compiler functions.

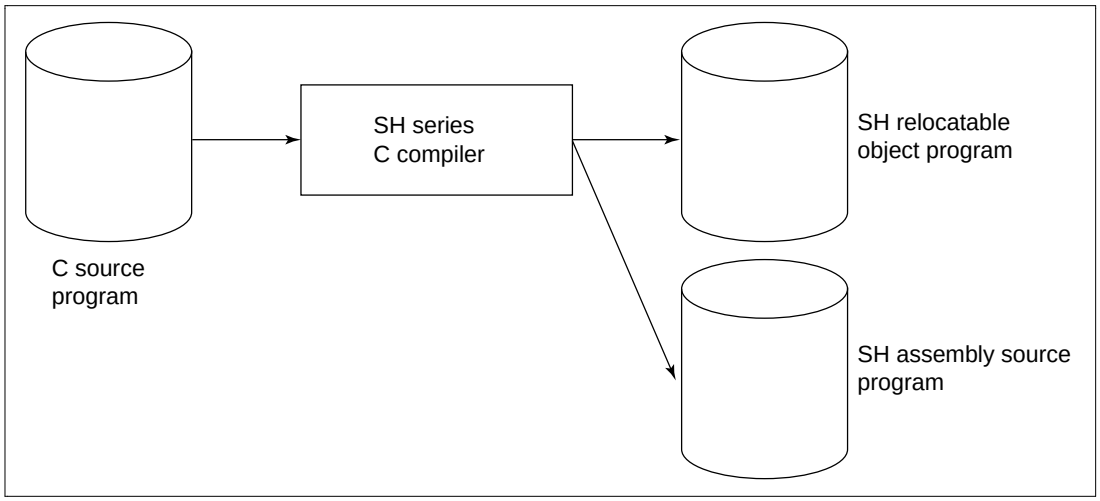
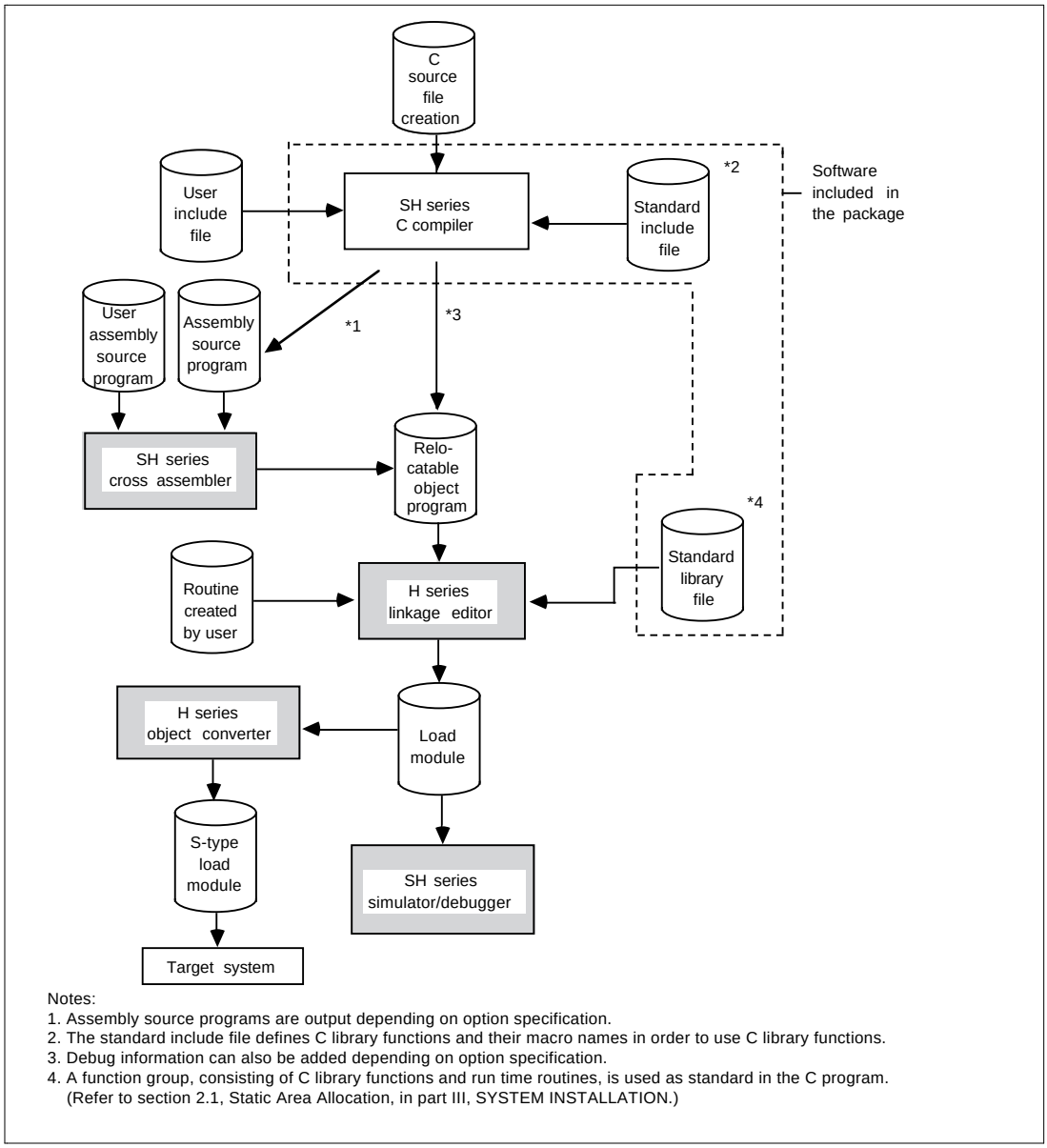


Figure 1.1 C Compiler Functions

A standard library file (a group of C language level functions that is used in C language program as standard) is also provided in addition to the C compiler.

Section 2 Developing Procedures

Figure 1.2 shows the relationship between the C compiler package and other software for program development. The C compiler package includes the software enclosed by the dotted line.



Notes:
1. Assembly source programs are output depending on option specification.
2. The standard include file defines C library functions and their macro names in order to use C library functions.
3. Debug information can also be added depending on option specification.
4. A function group, consisting of C library functions and run time routines, is used as standard in the C program. (Refer to section 2.1, Static Area Allocation, in part III, SYSTEM INSTALLATION.)

Figure 1.2 Relationship between the C Compiler and Other Software

Section 3 C Compiler Execution

This section explains how to invoke the C compiler, specify C compiler options, and interpret C compiler listings.

3.1 How to Invoke the C Compiler

The format for the command line used to invoke the C compiler is as follows.

```
shc[Δ<option>...][Δ<file name>[Δ<option>...]...]
```

The general operations of the C compiler are described below.

3.1.1 Compiling Programs

```
shcΔtest.c (RET)
```

The C source program test.c is compiled.

3.1.2 Displaying Command Line Format and Compiler Options

```
shc (RET)
```

The command line format and the list of the compiler options are displayed on the screen.

3.1.3 C Compiler Options

Insert minus (–) before options (**debug**, **listfile**, and **show**). Slash (/) can also be inserted in place of minus (–) for PC. When multiple options are specified, separate them with a space (Δ). The following shows the options for UNIX and PC. Also when multiple suboptions are specified, separate them with a comma (,).

```
shcΔ-debugΔ-listfileΔ-show=noobject,expansionΔtest.c (RET)
```

In PC, when multiple suboptions are specified, they can be enclosed in parentheses (( )).

```
shcΔ/debugΔ/listfileΔ/show=(noobject,expansion)Δtest.c(RET)
```

3.1.4 Compiling Multiple C Programs

Several C source programs can be compiled by a single command.

Example 1: Specifying multiple programs

```
shcΔtest1.cΔtest2.c (RET)
```

Example 2: Specifying options for all C source programs

```
shcΔ-listfileΔtest1.cΔtest2.c (RET)
```

The **listfile** option is valid for both test1.c and test2.c.

Example 3: Specifying options for particular C source programs

```
shcΔtest1.cΔtest2.cΔ-listfile (RET)
```

The **listfile** option is valid for only test2.c. Options specified for particular C source programs have priority over those specified for all C source programs.

3.2 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 C compiler and related software are shown in table 1.1. For details on naming files, refer to the user's manual of the host computer because naming rules vary according to each host computer.

Table 1.1 Standard File Extensions Used by the C Compiler

File Extension	Description
c	Source program file written in C
h	Include file
lis, lst	Listing file*
obj	Relocatable object program file
src	Assembly source program file
lib	Library file
abs	Absolute load module file
rel	Relocatable load module file
map	Linkage map listing file

Note: The listing file extension is lis on UNIX systems and lst on PC systems.

3.3 Compiler Options

Table 1.2 shows C compiler option formats, abbreviations, and defaults. Characters underlined indicate the minimum valid abbreviation. Bold characters indicate default assumptions.

Table 1.2 C Compiler Options

Item	Format	Suboption	Specification
CPU type	cpu =	**sh1** |	SH1 object is generated.
		sh2 |	SH2 object is generated.
		sh3 |	SH3 object is generated.
		sh3e	SH3E object is generated.
Optimization	optimize =	0 |	Object without optimization is output.
		1	Object with optimization is output.
Optimization select	speed		Optimization in both speed and size.
	nospeed		Optimization in balance between execution speed and execution size is selected.
	size		Optimization in program size is selected.
Debugging information	debug		Output
	nodebug		No output
Listings and formats	show =	source | **nosource** |	Source list yes/no
		object | noobject |	Object list yes/no
		statistics | nostatistics |	Statistics information yes/no
		include | **noinclude** |	List after include expansion yes/no
		expansion | **noexpansion** |	List after macro expansion yes/no
		width = <numeric value> |	Maximum characters per line: 0, 80 to 132
		length = <numeric value>	Maximum lines per page: 0, 40 to 255
	Default: **w = 132, l = 66**		

Table 1.2 C Compiler Options (cont)

Item	Format	Suboption	Specification
Listing file	listfile [= <list file name>]		Output
	nolistfile		No output
Object file	objectfile = <object file name>		Output
Object program format	code =	**machinecode** |	Program in machine language is output.
		asmcode	Assembly source program is output.
Macro name	define =	<macro name> = <name> |	<name> is defined as <macro name>.
		<macro name> = <constant> |	<constant> is defined as <macro name>.
		<macro name>	<macro name> is assumed to be defined.
Include file	include =	<path name>	Include file destination path name is specified (multi-specification is possible).
Section name	section =	program = <section name> |	Program area section name is specified.
		const = <section name> |	Constant area section name is specified.
		data = <section name> |	Initialized data area section name is specified.
		bss =<section name>	Non-initialized data area section name is specified.
	Default: **p=P, c=C, d=D, b=B**		
Help message	help		Output
Position independent code	pic =	**0** |	Position independent code is not generated.
		1	Position independent code is generated.
Area of string literal to be output	string =	**const** |	String literal is output to constant section (C).
		data	String literal is output to initialized data section (D).

Table 1.2 C Compiler Options (cont)

Item	Format	Suboption	Specification
Comment nesting	comment =	nest |	Permits comment (/* */) nesting.
		nonest	Does not permit comment (/* */) nesting.
Japanese code select in string literals	euc		Selects euc code.
	sjis		Selects sjis code.
Subcommand file select	subcommand =	<file name>	Includes command option from a file specified by <file name>.
Division operation	division =	**cpu** |	Uses cpu's division instruction.
		peripheral |	Uses a divider (with masking interruption).
		nomask	Uses a divider (without masking interruption).
Memory bit order	endian =	**big** |	Specifies maximum big endian.
		little	Specifies little endian.
Inline expansion specification	inline		Specifies inline expansion.
	inline =	<numeric value>	Specifies the maximum size of a function to expand where the function is called.
	noinline		
Default header file	preinclude =	<file name>	Includes contents of a specified file at the beginning of compilation units.
MACH and MACL registers	macsave =	0 |	Does not guarantee contents of MACH and MACL registers at function call.
		1	Guarantees contents of MACH and MACL registers at function call.

Table 1.2 C Compiler Options (cont)

Item	Format	Suboption	Specification
Information message output	message		Outputs information message.
	nomessage		Does not output information message.
Label 16-byte alignment	align16		Labels placed immediately after an unconditional branch instruction other than a subroutine call in a program section must be aligned in 16 bytes.
	noalign16		Does not place labels aligned in 16 bytes.
Double type to single precision	double =	float	Treats double type (double precision floating point number) as float type (single precision floating point number) as object.
Japanese character conversion	outcode =	euc |	Selects euc code.
		sjis	Selects sjis code.
ABS16 declaration	abs16 =	run |	Assumes all execution routines to have been declared with #pragma abs16.
		all	Generates all label addresses in 16 bits.
Loop unroll	loop		Optimizes loop unrolling.
	noloop		Does not optimize loop unrolling.
Inline expansion	nestinline =	<numeric value>	Specifies the number of times to expand nested inline functions.
EXTS and EXTU creation at data return	rtnext		Creates a sign-extension or zero-extension instruction for the upper bytes when returning a value to a program by the return statement.
	nortnext		Does not create a sign-extension or zero-extension instruction.

–cpu = **sh1** | sh2 | sh3 | sh3e

This option specifies a target CPU. A library to be linked differs according to a CPU. For details, refer to section 3.5, Correspondence to Standard Libraries in part I, OVERVIEW AND OPERATIONS.

–optimize = 0 | **1**

This option specifies compiler optimization.

optimize = 0 disables compiler optimization.

optimize = 1 enables compiler optimization.

–speed, **–nospeed**

This option specifies speed optimization. When a speed option is specified, program is executed faster but program size may increase. When nospeed is specified but size option is not specified, optimization is performed in program execution speed and program size.

–size

This option specifies optimization in object size.

–debug, **–nodebug**

This option specifies whether or not to output debugging information which is necessary for C source level debugging.

–show = source | **nosource** | **object** | noobject | **statistics** | nostatistics | include | **noinclude** | expansion | **noexpansion** | width = <numeric value> | length = <numeric value>

This option specifies the output format of a list file. This option is valid when a listfile option is specified.

show = width = 0	One line ends at a carriage code.
show = length = 0	The maximum number of lines is not specified; therefore, pagination is not performed.

–listfile [=<listfile name>], **–nolistfile**

This option specifies whether a list file is output. When a file name is not specified, a file that has the same name as the source file with a standard extension lis/lst is generated.

–objectfile = <objectfile name>

This option specifies an object file name to be output.

–code = **machinecode** | asmcode

This option specifies whether the compiler outputs an object file in a machine language or an assembler source file.

–define = <macro name> = <name> | <macro name> = <constant> | <macro name>

This option enables a macro definition at the beginning of a source program.
Table 1.3, describes macro names, names, and constants which can be specified using this option.

Table 1.3 Macro Names, Names, and Constants Specified by the Define Option

Item	Description	
Macro name	A string literal beginning with a letter or an underscore followed by zero or more letters, underscores, and numbers.	
Name	A string literal beginning with a letter or an underscore followed by zero or more letters, underscores, and numbers.	
Constant	Decimal constant:	A string literal of one or more numbers (0 to 9), or a string literal of one or more numbers followed by a period (.) followed by zero or more numbers.
	Octal constant:	A string literal that begins with a zero followed by one or more numbers (0 to 7).
	Hexadecimal constant:	A string literal that begins with a zero followed by an x, then followed by one or more numbers or alphabetical letters (A to F).

–include = <path name>

This option specifies a directory where an include file is searched for. For details on how to search, refer to Appendix A.1.13, Preproccessor.

–section = | program = <section name> | const = <section name> | data = <section name> | bss = < section name >

This option changes section names in object programs. Section names when this option is omitted are program area section P, constant area section C, initialized data area section D, and non-initialized data area section B.

–help

This option displays a list of compiler options. Once this option is specified, the other option(s) will be disabled.

–pic = **0** | 1

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. **Note that if cpu = SH1 is specified, pic = 1 is ignored.** A library to be linked varies according to the cpu, pic, endian, or double option. For details, refer to section 3.5, Correspondence to Standard Libraries in part I, OVERVIEW AND OPERATIONS.

Example

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

–string = **const** | data

When string = const is specified, string literals are output to constant area section (default is C). When string = data is specified, string literals are output to initialized data area section (default is D).

–comment = nest | **nonest**

This option specifies whether or not to permit comment /* */ nesting.

Example

```
/* comment
     int a; /*   nest1   /*   nset2   */  */
*/
```

When comment = nest is specified, an underlined section is treated as a nested comment and the outermost comment is enforced.
When comment = nonest is specified, a comment is treated to end by nest2*/. Therefore, a section after nest2*/ is treated as an error.

–euc

This option selects euc for the Japanese code for string literals in C program. When this option is omitted, euc or sjis is selected according to the host computer. For details, refer to section 3.5, Japanese Description in String Literals in part II, C Programming.

–sjis

This option selects sjis for the Japanese code for string literals in C program. When this option is omitted, euc or sjis is selected according to the host computer. For details, refer to section 3.5, Japanese Description in String Literals in part II, C Programming.

–subcommand = <file name>

This option assumes contents of a specified file name as an option. This option can be specified in a command line more than once. In a subcommand file, an parameters must be delimited by a space, a carriage return, or a tab. Contents in a subcommand file will be expanded to an area specified by a subcommand in a command line parameter. A subcommand option cannot be specified in a subcommand file.

Example: The following examples are the same as shc –debug –cpu=sh2 test.c.

Command line

shc –sub=test.sub test.c

Contents of test.sub

–debug
–cpu=sh2

–division = **cpu** | peripheral | nomask

This option selects an execution routine for an integer division in a C source program. This option can be combined with a suboption in the cpu option. However, only the SH2 can execute an object program that specifies peripheral or nonmask as suboption.

1. cpu: specifies an execution routine which uses the DIV1 instruction
2. peripheral: specifies an execution routine using a divider (15 is set to interrupt mask level)
3. nomask: specifies an execution routine using a divider (no change in interrupt mask level)

Note the following before specifying a peripheral or nomask option.

1. Zero division is not checked or errno is not set.
2. If nomask is specified and an interrupt occurs during operation of a divider and the divider is used in an interrupt routine, the correct operation is not guaranteed.
3. An overflow interrupt is not supported.
4. Results after operation such as zero division or overflow depend on the divider specifications. Some of them may be different from those when a cpu suboption is specified.

–endian = **big** | little

This option can be combined with a suboption in a cpu option. However, only the SH3 or SH3E can execute an object program for little endian. The library to be linked depends on endian, cpu, pic, and double options. For details, refer to section 3.5, Correspondence to Standard Libraries in part I, OVERVIEW AND OPERATIONS.

–inline, –inline = <numeric value>, –noinline

This option specifies whether to expand a function automatically at the statement where the function is called. The value specified in suboption <numeric value> indicates the maximum number of nodes of a function (the total number of characters of operators and variables excluding the declaration field) to expand where the function is called. The default of speed option specification is inline = 20. The default when nospeed, size, or optimize = 0 option is specified noinline.

–preinclude = <file name>

This option includes file contents at the beginning of compilation units.

–macsave = 0 | **1**

This option specifies whether contents of the MACH or MACL registers are guaranteed before and after a function call.
macsave = 0 does not guarantee the contents of the MACH or MACL registers before and after a function call. macsave = 1 guarantees the contents of MACH and MACL registers before and after a function call. A function that is compiled using macsave = 1 cannot call a function that is compiled using macsave = 0. However, the opposite is possible.

–message, **nomessage**

This option specifies information message output. nomessage option does not output information message.

–align16, **noalign16**

This option aligns all labels placed immediately after an unconditional branch instruction other than subroutine calls in a program section in 16 bytes. noalign16 option does not place labels aligned in 16 bytes.

–double = float

This option treats double type declaration/cast (double precision floating point number) as float type declaration/cast (single precision floating point number) before generating object.

–outcode = euc | sjis

This option selects euc for the Japanese character code when outcode = euc is specified, and sjis when outcode = sjis is specified.

–abs16 = run | all

This option assumes all execution routines to have been declared with #pragma abs16 when abs16 = run is specified, and generates all label addresses in 16 bits when abs16 = all is specified.

–loop, **–noloop**

This option specifies whether to optimize loop unrolling.
The loop option performs loop unrolling. The noloop option does not perform loop unrolling.

–nestinline = <numeric value>

This option specifies the number of times to expand the inline function. Up to 16 times can be specified. When this option is not specified (default), the inline function is expanded once (nestinline=1).

–rtnext, **–nortnext**

This option performs sign extension or zero extension after setting a value in R0, which is the place to set the return value, in a return statement of a function that returns a (unsigned) char type or (unsigned) short type (see section 2.2.3 in part II, C PROGRAMMING) to a program. This enables type conversion for a return value before the actual value is returned to a program. If a prototype is declared at the caller, this option is not required. The nortnext option does not perform sign extension or zero extension.

3.4 Option Combinations

If a pair of conflicting options or suboptions are specified for a file, only one of them is considered valid. Table 1.4 shows such option combinations.

Table 1.4 Option Combinations

Valid Option	Invalid Option
nolist	show
code = asmcode*	debug*
	show = object
help	All other options
cpu = sh1	pic = 1
optimize = 0	loop

Note: When debug option is specified during assembly source output, a .LINE directive is embedded in the output code. A .LINE directive gives C language source line information to a debugger. After that, C language source lines are displayed for debugging. However, C language level debugging is not performed for variable values.

3.5 Correspondence to Standard Libraries

There are 22 types of standard library combinations. Link a library listed in table 1.5 according to the combination of a cpu, pic, endian, or double option.

shclib.lib (for SH1)
shcnpic.lib (for SH2, not for position independent code)
shcpic.lib (for SH2, for position independent code)
shc3npb.lib (for SH3, not for position independent code, big endian)
shc3pb.lib (for SH3, for position independent code, big endian)
shc3npl.lib (for SH3, not for position independent code, little endian)
shc3pl.lib (for SH3, for position independent code, little endian)
shcenpb.lib (for SH3E, not for position independent code, big endian)
shcepb.lib (for SH3E, for position independent code, big endian)
shcenpl.lib (for SH3E, for position independent code, little endian)
shcepl.lib (for SH3E, for position independent code, little endian)
shclibf.lib (for SH1, double = float option specification)
shcnpicf.lib (for SH2, not for position independent code, double = float option specification)
shcpicf.lib (for SH2, for position independent code, double = float option specification)
shc3npbf.lib (for SH3, not for position independent code, big endian, double = float option specification)
shc3pbf.lib (for SH3, for position independent code, big endian, double = float option specification)
shc3nplf.lib (for SH3, not for position independent code, little endian, double = float option specification)
shc3plf.lib (for SH3, for position independent code, little endian, double = float option specification)
shcenpbf.lib (for SH3E, not for position independent code, big endian, double = float option specification)
shcepbf.lib (for SH3E, for position independent code, big endian, double = float option specification)
shcenplf.lib (for SH3E, not for position independent code, little endian, double = float option specification)
shceplf.lib (for SH3E, for position independent code, little endian, double = float option specification)

Table 1.5 Correspondence between Standard Libraries and Compile Options

double specification	**None**			
endian specification	**endian = big**		**endian = little**	
pic specification	**pic = 0**	**pic = 1**	**pic = 0**	**pic = 1**
cpu = sh1	shclib.lib	—	—	—
cpu = sh2	shcnpic.lib	shcpic.lib	—	—
cpu = sh3	shc3npb.lib	shc3pb.lib	shc3npl.lib	shc3pl.lib
cpu = sh3e	shcenpb.lib	shcepb.lib	shcenpl.lib	shcepl.lib

double specification	**double = float**			
endian specification	**endian = big**		**endian = little**	
pic specification	**pic = 0**	**pic = 1**	**pic = 0**	**pic = 1**
cpu = sh1	shclibf.lib	—	—	—
cpu = sh2	shcnpicf.lib	shcpicf.lib	—	—
cpu = sh3	shc3npbf.lib	shc3pbf.lib	shc3nplf.lib	shc3plf.lib
cpu = sh3e	shcenpbf.lib	shcepbf.lib	shcenplf.lib	shceplf.lib

3.6 C Compiler Listings

This section describes C compiler listings and their formats.

3.6.1 Structure of C Compiler Listings

Table 1.6 shows the structure and contents of C compiler listings.

Table 1.6 Structure and Contents of C Compiler Listings

List Structure	Contents	Option Specification Method*[1]	Default
Source listing	Listing consists of source programs	**show=[no]source**	No output
	Source program listing after include file and macro expansion	**(show=[no]include)*[2] (show=[no]expansion)**	No output
Object listing	Machine language generated by the C compiler and assembly code	**show=[no]object**	Output
Statistics	Total number of errors, number of source program lines, size of each section (byte), and number of symbols	**show=[no]statistics**	Output
Command line specification	File names and options specified in the command line	—	Output

Notes: 1. All options are valid when **listfile** is specified.
2. The option enclosed in parentheses is only valid when **show = source** is specified.

3.6.2 Source Listing

The source listing can 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 1.3 and 1.4 show examples of these output formats. Bold characters in figure 1.4 show the differences.

```
************ 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 1.3 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 1.4 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.

3.6.3 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 1.5 and 1.6 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 1.5 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 1.6 Object Listing Output for show = nosource, object

Description:

(1) Section attribute (P, C, D, and B) 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 (always output)

(8) Routine name that is being executed

3.6.4 Statistics Information

Figure 1.7 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):               0x000044 Byte(s)
CONSTANT    SECTION(C):               0x000000 Byte(s)
DATA        SECTION(D):               0x000000 Byte(s)
BSS         SECTION(B):               0x000000 Byte(s)

 TOTAL PROGRAM SIZE:      0x000044 Byte(s)


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

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

Figure 1.7 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 option **noobject** is specified. In addition, section size information (3) is output (indicated as “1”) or not output (indicated as “0”) according to its specification when option **code = asmcode** is specified.

3.6.5 Command Line Specification

The file names and options specified on the **command** line when the compiler is invoked are displayed. Figure 1.8 shows an example of **command** line specification information.

```
*** COMMAND PARAMETER ***

-listfile test.c
```

Figure 1.8 Command Line Specification

3.7 C Compiler Environment Variables

Environment variables to be used by the compiler are listed in table 1.7.

Table 1.7 Environment Variables

Environment Variable	Explanation in Use
SHC_LIB	Specifies a directory at which compiler load module and system include file exists.
SHC_INC	Specifies a directory at which a system include file exists. More than one directory can be specified by dividing directories using commas. A system include file is searched for at a directory specified using an include option specified directory, SHC_INC-specified directory, and system directory (SHC_LIB) in this order.
SHC_TMP	Specifies a directory where the compiler generates a temporary file. This environment variable is required for a PC. For UNIX, a directory indicated in TMPDIR is specified when this environment variable is specified. If SHC_TMP or TMPDIR is not specified, a temporary file is generated in /usr/tmp.
SHCPU	Specifies CPU type by compiler **–cpu** option using environment variables. The following is specified: SHCPU=SH1 (same as –cpu=sh1) SHCPU=SH2 (same as –cpu=sh2) SHCPU=SHDSP (same as –cpu=sh2) SHCPU=SH3 (same as –cpu=sh3) SHCPU=SH3E (same as –cpu=sh3e) An error will occur if anything other than the above is specified. Specifying lower case characters will also generate an error. When the specification of CPU by SHCPU environment variable and **–cpu** option differs, a warning message is displayed. **–cpu** option has priority to SHCPU specification.

3.8 Implicit Declaration by Option

Using **–cpu**, **–pic**, **–endian**, or **–doubl**e option results in an implicit **#define** declaration. See the following.

Table 1.8 Implicit Declaration

Option	Implicit Declaration
–cpu = sh1	#define _SH1 (including default)
–cpu = sh2	#define _SH2
–cpu = sh3	#define _SH3
–cpu = sh3e	#define _SH3E
–pic	#define _PIC
–endian = big	#define _BIG (including default)
–endian = little	#define _LIT
–double = float	#define _FLT

The following shows an specification example.

Example:

```
#ifdef _BIG
#ifdef _SH1
   ......                    Valid when –cpu = sh1 –endian = big option is specified
   ......                    (Also valid when no option is specified for –cpu or –endian)
#endif
#endif
#ifdef _SH2
   ......                    Valid when –cpu = sh2 option is specified
#endif
#ifdef _SH3
#ifdef _BIG
   ......                    Valid when –cpu = sh3 –endian = big option is specificed
#endif
#ifdef _LIT
   ......                    Valid when –cpu = sh3 –endian = little option is specified
#endif
#endif
```

Rules: 1. If no option is specified (default), **#define _SH1** or **#define _BIG** is set.
2. The implicit **#define** declaration is specified as **#undef** in the source file.

PART II

C PROGRAMMING

Section 1 Limits of the C Compiler

Table 2.1 shows the limits on source programs that can be handled by the C compiler. Source programs must fall within these limits. To edit and compile efficiently, it is recommended to split the source program into smaller programs (approximately two ksteps) and compile them separately.

Table 2.1 Limits of the C Compiler

Classification	Item	Limit
Invoking the C compiler	Number of source programs that can be compiled at one time	None*[1]
	Total number of macro names that can be specified using the **define** option	None
	Length of file name (characters)	128
Source programs	Length of one line (characters)	4096
	Number of source program lines in one file	65535
	Number of source program lines that can be compiled	None
Preprocessing	Nesting levels of files in an **#include** directive	30
	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	63
	Number of expansions of a macro name	32
	Nesting levels of **#if**, **#ifdef**, **#ifndef**, **#else**, or **#elif** directives	32
	Total number of operators and operands that can be specified in an **#if** or **#elif** directive	512
Declarations	Number of function definitions	512
	Number of internal labels*[2]	32767
	Number of symbol table entries*[3]	24576
	Total number of pointers, arrays, and functions that qualify the basic type	16
	Array dimensions	6

Table 2.1 Limits of the C Compiler (cont)

Classification	Item	Limit
Statements	Nesting levels of compound statements	32
	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	511
	Nesting levels of **for** statements	16
Expressions	Number of parameters that can be specified using a function definition or a function call operation	63
	Total number of operators and operands that can be specified in one expression	About 500
Standard library	Number of files that can be opened at once in open function	20

Notes: 1. For PC, the number of command line that can be compiled at one time is limited to 127 characters.

2. An internal label is internally generated by the C compiler to indicate a static variable address, **case** label address, **goto** label address, or a branch destination address generated by **if**, **switch**, **while**, **for**, and **do** statements.

3. The number of symbol table entries is determined by adding the following numbers:

   Number of external identifiers

   Number of internal identifiers for each function

   Number of string literals

   Number of initial values for structures and arrays in compound statements

   Number of compound statements

   Number of **case** labels

   Number of **goto** labels

Section 2 Executing a C Program

This section covers object programs which are generated by the C compiler. In particular, this section explains the items necessary for the linkage of the C program with an assembly program, or when incorporating a program into an SH system.

2.1 Structure of Object Programs: This section discusses the characteristics of memory areas used for C programs and standard library functions.

2.2 Internal Data Representation: This section explains the internal representation of data used by a C program. This information is required when data is shared among C programs, hardware, and assembly programs.

2.3 Linkage with Assembly Programs: This section explains the rules for variables and function names that can be mutually referenced by multiple object programs. This section also discusses how to use registers, and how to transfer parameters and return values when a C program calls a function. This information is required for C program functions calling assembly program routines or vice versa.

Refer to respective hardware manuals for details on SH hardware.

2.1 Structure of Object Programs

This section discusses the characteristics of memory areas used by a C program or standard library function in terms of the following items.

1. Section
   Composed of memory areas which are allocated statically by the C compiler. Each section has a name and type. A section name can be changed by the compiler option **section**.
2. Write Operation
   Indicates whether write operations are enabled or disabled at program execution.
3. Initial Value
   Shows whether there is an initial value when program execution starts.
4. Alignment
   Restricts addresses to which data is allocated.

Table 2.2 shows the types and characteristics of those memory areas.

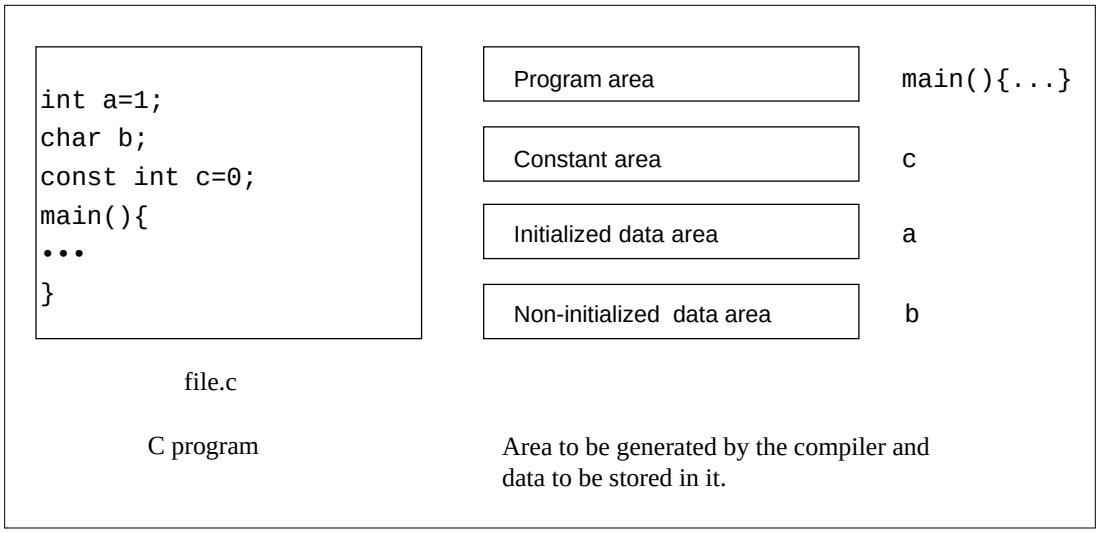
Table 2.2 Memory Area Types and Characteristics

Memory Area Name	Section Name[*1]	Section Type	Write Operation	Initial Value	Alignment	Contents
Program area	P	code	Disabled	Yes	4 bytes[*2]	Stores machine codes.
Constant area	C	data	Disabled	Yes	4 bytes	Stores **const** data.
Initialized data area	D	data	Enabled	Yes	4 bytes	Stores initial value.
Non-initialized data area	B	data	Enabled	No	4 bytes	Stores data whose initial values are not specified.
Stack area	—	—	Enabled	No	4 bytes	Required for program execution. Refer to section 2.2 Dynamic Area Allocation, in part III, SYSTEM INSTALLATION.
Heap area	—	—	Enabled	No	—	Used by a library function (**malloc**, **realloc**, or **calloc**). Refer to section 2.2 Dynamic Area Allocation, in part III SYSTEM INSTALLATION.

Notes: 1. Section name shown is the default generated by the C compiler when a specific name is not specified by the compiler **–section** option.
2. Becomes 16 bytes when **–align16** option is specified.

Example:

This program example shows the relationship between a C program and the sections generated by the C compiler.



2.2 Internal Data Representation

This section explains the internal representation of C language data types. The internal data representation is determined according to the following four items:

1. Size

   Shows the memory size necessary to store the data.
2. 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 an even byte address, and 4-byte alignment in which data is allocated to an address indivisible by four.
3. Data range

   Shows the range of scalar-type data.
4. Data allocation example

   Shows how the elements of combined-type data are allocated.

2.2.1 Scalar-Type Data

Table 2.3 shows the internal representation of scalar-type data used in C.

Table 2.3 Internal Representation of Scalar-Type Data

Data Type	Size (bytes)	Alignment (bytes)	Sign	Data Range	
				Minimum Value	Maximum Value
char (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	4	Used	– ∞	+ ∞
double **long double**	8*1	4	Used	– ∞	+ ∞
Pointer	4	4	Unused	0	$2^{32} - 1$ (4294967295)

Note: The size of double type is 4 bytes if **–double=float** option is specified.

2.2.2 Combined-Type Data

This part explains the internal representation of array, structure, and union data types. Table 2.4 shows the internal data representation of combined-type data.

Table 2.4 Internal Representation of Combined-Type Data

Data Type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array	Maximum array element alignment	Number of array elements x element size	`int a[10];` Alignment: 4 bytes Size: 40 bytes

Structure*1	Maximum structure member alignment	Total size of members*1	`struct {`
			`  int a, b;`
			`}`
			Alignment: 4 bytes
			Size: 8 bytes

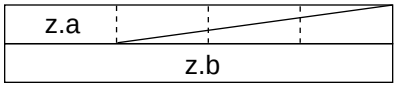
Union	Maximum union member alignment	Maximum size of member*2	`union {`
			`  int a,b;`
			`}`
			Alignment: 4 bytes
			Size: 4 bytes

In the following notes, a rectangle indicates four bytes.

Note 1:

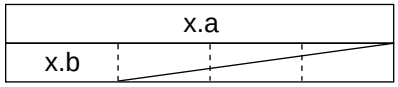
When allocating a member of a structure type, an empty area may be created between a member and the previous member to adjust the alignment of a data type of the member.

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



When a structure has a four-byte alignment, and the last member ends at the first, second or third byte, the remaining bytes are included in a structure type area.

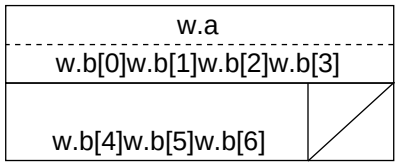
```
struct   {
  int a;
  char b;}x;
```



Note 2:

When a union has a four-byte alignment, and the maximum value of the member size is not a multiple of four, the remaining bytes up to a multiple of four are included in the union type area.

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



2.2.3 Bit Fields

A bit field is a member of a structure. This part explains how bit fields are allocated.

Bit field members: Table 2.5 shows the specifications of bit field members.

Table 2.5 Bit Field Member Specifications

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

Notes: 1. To use a member of a bit field, 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**.
2. Zero extension: Zeros are written to the high-order bits to extend data.
3. 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:

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

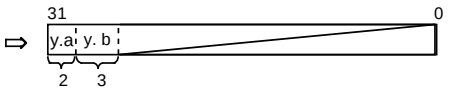
   Example:

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

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

   Example:

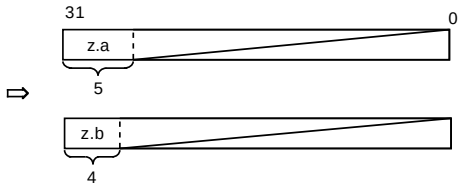
```
struct b1{
  long         a:2;
  unsigned int b:3;
}y;
```



3. Bit field members having type specifier with different sizes are allocated to the following areas.

   Example:

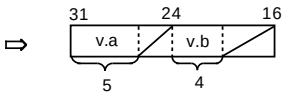
```
struct b1{
  int  a:5;
  char b:4;
}z;
```



4. 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.

   Example:

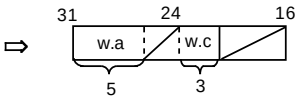
```
struct b2{
  char a:5;
  char b:4;
}v;
```



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

   Example:

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



2.2.4 Memory Allocation of Little Endian

Memory is allocated to a data array using a little endian as follows.

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

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

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

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

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

Example: When a four-byte data 0x12345678 is allocated in 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 order of eight-byte data will be reversed for a big endian and a little endian.

Example: When a four-byte data 0x123456789abcdef is allocated in 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

Combined-Type Data: Members of combined-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 in 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 in 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

2.3 Linkage with Assembly Programs

The C compiler supports intrinsic functions such as access to the SH microcomputer registers as. Refer to section 3.2, Intrinsic Functions, in part II, C PROGRAMMING, for details on intrinsic functions. However, processes that cannot be written in C, such as the multiply and accumulate operation using the MAC instruction, should be written in assembly language and afterwards linked to the C program.

This section explains two key items which must be considered when linking a C program to an assembly program:

- External identifier reference
- Function call interface

2.3.1 External Identifier Reference

Functions and variable names declared as external identifiers in a C program can be referenced or modified by both assembly programs and C programs. The following are regarded as external identifiers by the C compiler:

- A global variable which has a storage class other than **static**
- A variable name declared in a function with storage class **extern**
- A function name whose storage class is other than **static**

When variable names which are defined as external identifiers in C programs, are used in assembly programs, an underscore character (_) must be added at the beginning of the variable name (up to 250 characters without the leading underscore).

Example 1: An external identifier defined in an assembly program is referenced by a C program

- In an assembly program, symbol names beginning with an underscore character (_) are declared as external identifiers by an .EXPORT directive.
- In a C program, symbol names (with no underscore character (_) at the head) are declared as external identifiers.

Assembly program (definition)

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

C program (reference)

```
extern int a,b;


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

Example 2: An external identifier defined in a C program is referenced by an assembly program

- In a C program, symbol names (with no underscore character (_) at the head) are defined as external identifiers.
- In an assembly program, external references to symbol names beginning with an underscore character (_) are declared by an .IMPORT directive.

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
```

2.3.2 Function Call Interface

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

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

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.

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 one must be done by the calling function.

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

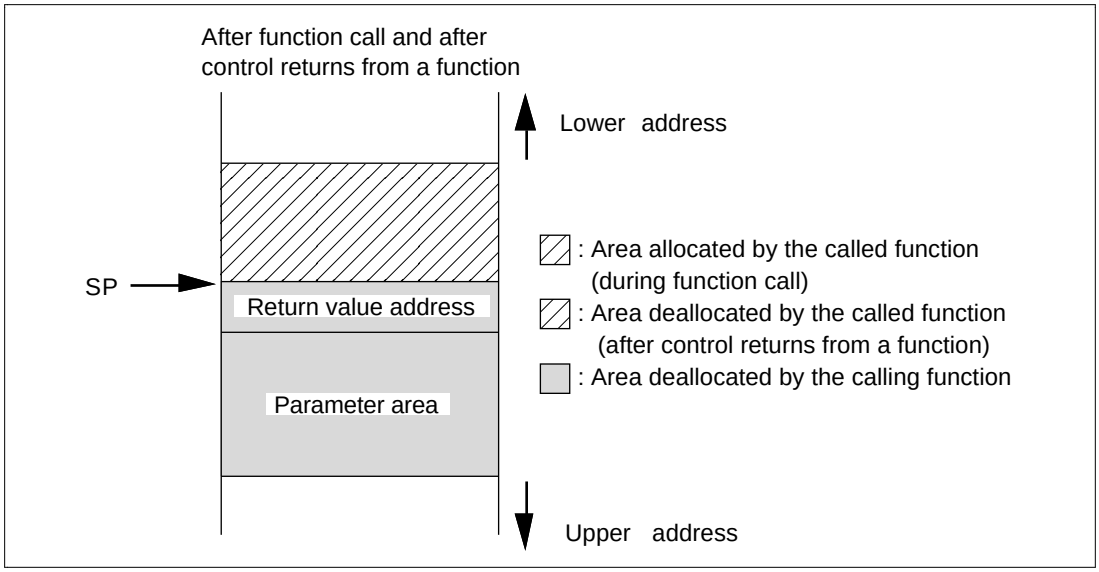


Figure 2.1 Allocation and Deallocation of a Stack Frame

Registers: Some registers change after a function call, while some do not. Table 2.6 shows how registers change according to the rules.

Table 2.6 Rules on Changes in Registers After a Function Call

Item	Registers Used in a Function	Notes on Programming
Registers whose contents may change	R0 to R7, FR0 to FR11*, FPUL*, and FPSCR*	If registers used in a function contain valid data when a program calls the function, the program must push the data onto the stack or register before calling the function. The data in registers used in called function can be used freely without being saved.
Registers whose contents may not change	R8 to R15, MACH, MACL, PR, and FR12 to FR15*	The data in registers used in functions is pushed onto the stack or register before calling the function, and popped from the stack or register only after control returns from the function. Note that data in the MACH and MACL registers are not guaranteed if the option **macsave=0** is specified.

Note: Indicates a register for SH3E floating point.

The following examples show the rules on register changes.

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

  Assembly program (called program)

```
        .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
```

Data in those registers needed by the called function is pushed onto the stack.

Function processing
(Since data in registers R0 to R7 is pushed onto a stack by the calling C program, the assembly program can use them freely without having to save them first.)

Register data is popped from the stack.

C program (calling program)

```
extern void sub();

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

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

  C program (called program)

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

  Assembly program (calling program)

```
        .IMPORT  _sub
        .SECTION P,CODE,ALIGN=2
              .
              .


        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.

Store 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.

The PR register is restored.

Address data of function sub.

Setting and Referencing Parameters and Return Values: This section explains how to set and reference parameters and return values. The ways of setting and referencing parameters and return values for each function depend on whether or not the type of the parameter or the return value is declared explicitly. A prototype function declaration is used to declare parameters and returns values explicitly.

This section first explains the general rules concerning parameters and return values, and then how the parameter area is allocated, and how areas are established for return values.

- General rules concerning parameters and return values
  - — Passing parameters

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

    char, **unsigned char**, **short**, and **unsigned short** type parameters are converted to **int** type parameters.

    float type parameters are converted to **double** type parameters.

    Types other than the above cannot be converted to another type.

    Return value type conversion:

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

Example:

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

(2)  void p ( int,... );
     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.

- Parameter area allocation

  Parameters are allocated to registers, or when this is impossible, to a stack parameter area. Figure 2.2 shows the parameter area allocation. Table 2.7 lists rules on general parameter area allocation.

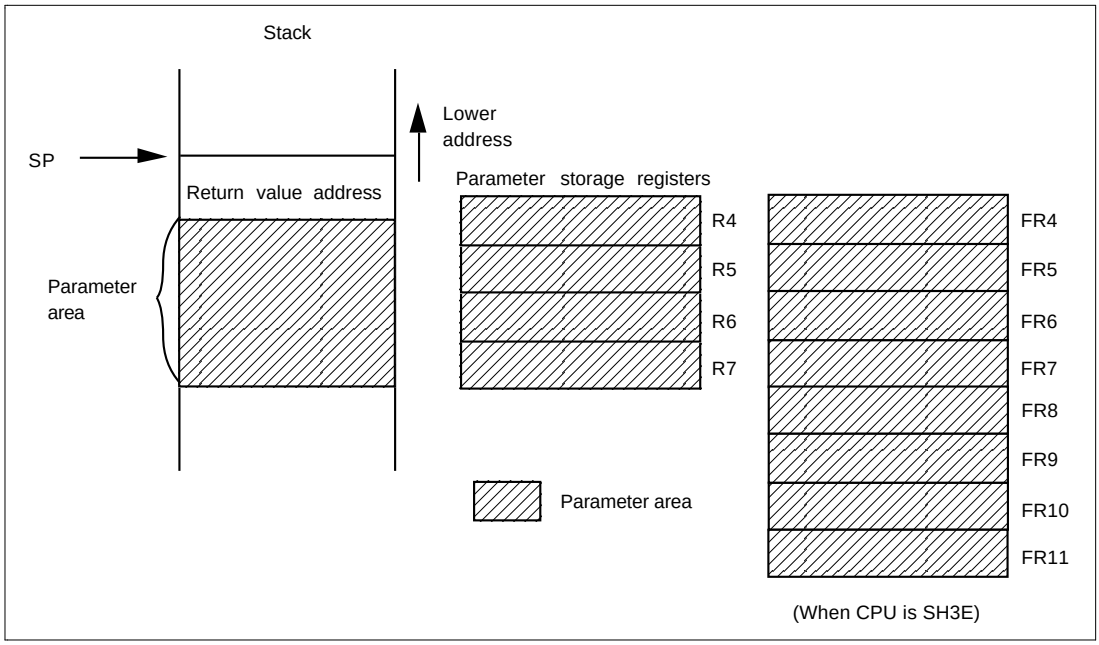


Figure 2.2 Parameter Area Allocation

Table 2.7 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, short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float** (when CPU is not SH3E), and pointer	(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[*2] (3) Other parameters are already allocated to R4 to R7.
FR4 to FR11[*1]	**float** (when CPU is SH3E)	

Notes: 1. Indicates a register for SH3E floating point.
2. 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.
```

- 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 2.3 shows an example of parameter allocation to registers.

```
f(char a,int b)
{
      .
      .
}
```

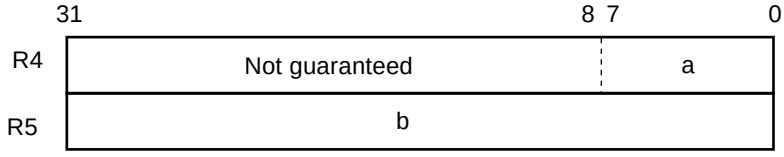


Figure 2.3 Example of Allocation to Parameter Registers

— 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 or union 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 SH stack pointer is incremented or decremented in 4-byte units.

Refer to appendix B, Parameter Allocation Example, 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 2.8 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 former in the return value address area before calling the function (see figure 2.4). The return value is not written if its type is **void**.

Table 2.8 Return Value Type and Setting Area

Return Value Type	Return Value Area
char, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, and pointer	R0: 32 bits (The contents of the upper three bytes of **char**, or **unsigned char** and the contents of the upper two bytes of **short** or **unsigned short** are not guaranteed.) However, when the **–rtnext** option is specified, sign extension is performed for **char** or **short** type, and zero extension is performed for **unsigned char** or **unsigned short** type. FR0: 32 bits (When cpu is SH3E, and the return value is **float** type.)
double, **long double**, structure, union	Return value setting area (memory)

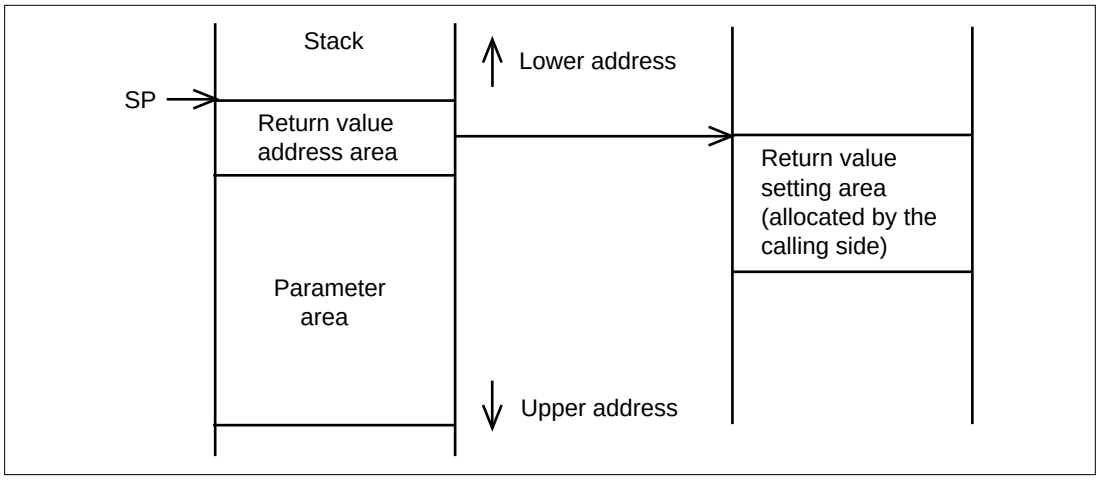


Figure 2.4 Return Value Setting Area Used When Return Value Is Written to Memory

Section 3 Extended Specifications

This section describes C compiler extended specifications:

- interrupt functions
- intrinsic functions
- section change function
- single-precision floating-point library
- Japanese description in string literals
- inline function
- inline expansion in assembly language
- specifying two-byte address variable
- specifying GBR base variable
- register save and recovery control
- global variable register allocation

3.1 Interrupt Functions

A preprocessor directive (**#pragma**) specifies an external (hardware) interrupt function. The following section describes how to create an interrupt function. Since the interrupt operation of SH3 and SH3E differ from that of the SH1 and SH2, interrupt handlers are necessary.

3.1.1 Description

`#pragma interrupt` (function name [(interrupt specifications)]
[, function name [(interrupt specifications)]])

Table 2.9 lists interrupt specifications.

Table 2.9 Interrupt Specifications

Item	Form	Options	Specifications
Stack switching specification	sp=	<variable> | &<variable> | <constant>	The address of a new stack is specified with a variable or a constant. <variable>: Variable value &<variable>: Variable (pointer type) address <constant>: Constant value
Trap-instruction return specification	tn=	<constant>	Termination is specified by the TRAPA instruction <constant>: Constant value (trap vector number)

3.1.2 Explanation

#pragma interrupt declares an interrupt function. An interrupt function will preserve 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:

```
extern int  STK[100];

int  *ptr = STK + 100;
#pragma interrupt ( f(sp=ptr, tn=10) )
                          (a)      (b)
```

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 2.5. After the previous PC and SR (status register) are popped from the stack by the RTE instruction in the trap routine, control is returned from the interrupt function.

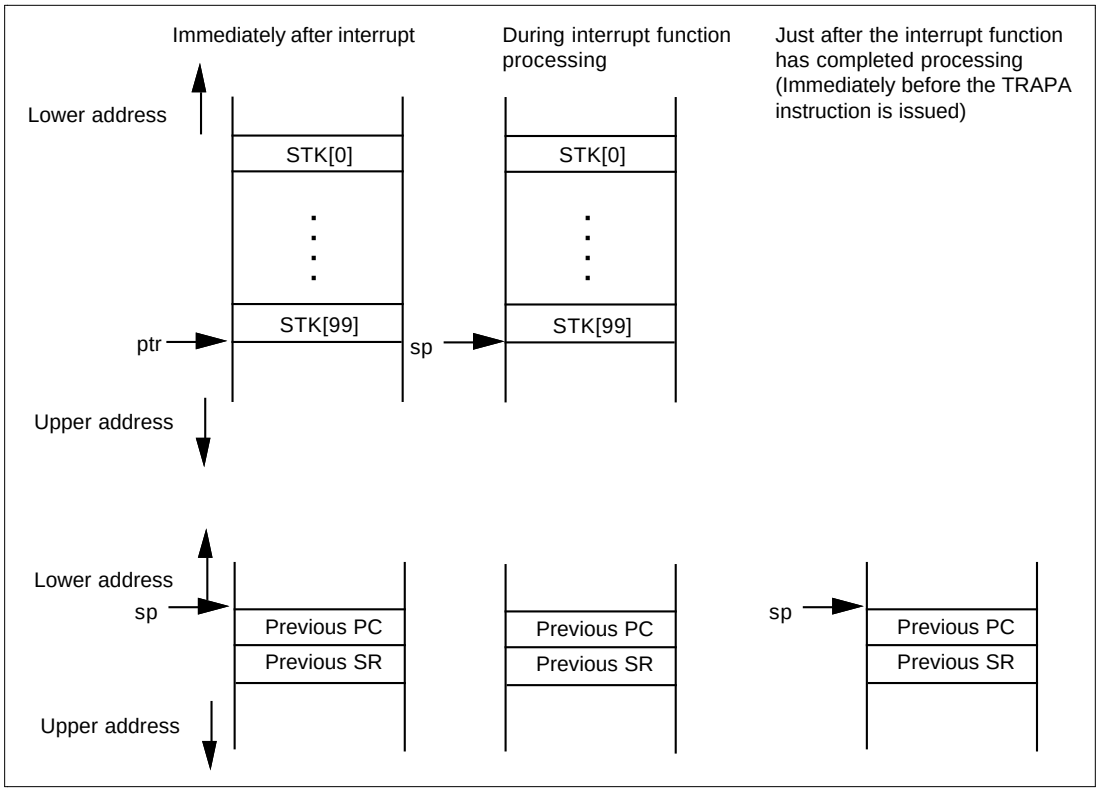


Figure 2.5 Stack Processing by an Interrupt Function

3.1.3 Notes

1. Only global functions can be specified for an interrupt function definition and the storage class specifier must be **extern**. Even if storage class **static** is specified, the storage class is handled as **extern**.

   The function must return **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 declared correctly.
   (b) returns data that is not void, thus (b) is declared incorrectly. An error is output.

2. A function declared as an interrupt function cannot be called within the program. If attempted, an error is output. However, if the function is called within a program which does not declare it to be an interrupt function, an error is not output but correct program execution will not be guaranteed.

   Example (An interrupt function is declared)**:**

```
#pragma interrupt(f1)
void f1(void){...}
int f2(){ f1();} ............ (a)
```

   Description: Function f1 cannot be called in the program because it is declared as an interrupt function. An error is output at (a).

   Example (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 for extern int f1(); is generated. If function f1 is declared as an interrupt function in another file, correct program execution cannot be guaranteed.

3.2 Intrinsic Functions

The C compiler provides the intrinsic functions for the SH microcomputer, which (functions) are described below.

3.2.1 Intrinsic Functions

The following functions can be specified by 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

3.2.2 Description

<machine.h>, <umachine.h>, or <smachine.h> must be specified when using intrinsic functions.

3.2.3 Intrinsic Function Specifications

Table 2.10 lists intrinsic functions.

Table 2.10 Intrinsic Functions

No	Item	Function	Specification	Description
1	Status register (SR)	Setting the status register	`void set_cr(int cr)`	Sets cr (32 bits) in the status register
2		Referencing to the status register	`int get_cr(void)`	Refers to the status register
3		Setting the interrupt mask	`void set_imask(int mask)`	Sets mask (4 bits) in the interrupt mask (4 bits)
4		Referencing to the interrupt mask	`int get_imask(void)`	Refers to the interrupt mask (4 bits)
5	Vector base register (VBR)	Setting the vector base register	`void set_vbr(void **base)`	Sets **base (32 bits) in VBR
6		Referencing to the vector base register	`void **get_vbr(void)`	Refers to VBR
7	Global base register (GBR)	Setting GBR	`void set_gbr(void *base)`	Sets *base (32 bits) in GBR
8		Referencing to GBR	`void *get_gbr(void)`	Refers to GBR
9		Referencing to GBR- based byte	`unsigned char gbr_read_byte(int offset)`	Refers to byte data (8 bits) at the address indicated by adding GBR and the offset specified
10		Referencing to GBR- based word	`unsigned short gbr_read_word(int offset)`	Refers to word data (16 bits) at the address indicated by adding GBR and the offset specified
11		Referencing to GBR- based long word	`unsigned long gbr_read_long(int offset)`	Refers to long word data (32 bits) at the address indicated by adding GBR and the offset specified

Table 2.10 Intrinsic Functions (cont)

No	Item	Function	Specification	Description
12	Global base register (GBR) (cont)	Setting GBR-based byte	`void gbr_write_byte (int offset, unsigned char data)`	Sets data (8 bits) at the address indicated by adding GBR and the offset specified
13		Setting GBR-based word	`void gbr_write_word (int offset, unsigned short data)`	Sets data (16 bits) at the address indicated by adding GBR and the offset specified
14		Setting GBR-based long word	`void gbr_write_long (int offset, unsigned long data)`	Sets data (32 bits) at the address indicated by adding GBR and the offset specified
15		AND of GBR base	`void gbr_and_byte (int offset, unsigned char mask)`	ANDs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
16		OR of GBR base	`void gbr_or_byte (int offset, unsigned char mask)`	ORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
17		XOR of GBR base	`void gbr_xor_byte (int offset, unsigned char mask)`	XORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address

Table 2.10 Intrinsic Functions (cont)

No	Item	Function	Specification	Description
18	Global base register (GBR) (cont)	TEST of GBR base	`int gbr_tst_byte (int offset, unsigned char mask)`	ANDs mask with the byte data at the address indicated by adding GBR and the offset specified, and checks if the byte data at the offset from GBR is 0 or not, and sets the result in the T bit
19	Special instruc-tions	SLEEP instruction	`void sleep(void)`	Expands the SLEEP instruction
20		TAS instruction	`int tas(char *addr)`	Expands TAS.B @addr
21		TRAPA instruction	`int trapa(int trap_no)`	Expands TRAPA #trap_no
22	Special instruc-tions (cont)	OS system call	`int trapa_svc( int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)` trap-no: Trap number code: Function code para 1 to 4: Parameter (0 to 4 variables) type 1 to 4: Parameter type: general integer or pointer type	Enables executing HI-SH7 (Hitachi Industrial Realtime Operating System SH7000 Series) and other OS system calls. When trapa_svc is executed, code is specified in R0, and para 1 to para4 in R4 to R7, respectively. Then, TRAPA #trap_no is executed.
23		PREF instruction	`void prefetch (void *p)` Note: The instruction is prefetched only when the compiler option **cpu = sh3** is specified.	If the instruction is prefetched, an area indicated by the pointer (16-byte data from (int)p&0xfffffff0) is written to the cache memory. This does not affect any programming logical operation.

Table 2.10 Intrinsic Functions (cont)

No	Item	Function	Specification	Description

24	Mutiply and accumulate operation	MAC.W instruction	`int macw( short *ptr1, short *ptr2, unsigned int count)`
			
			`int macwl( short *ptr1, short *ptr2, unsigned int count, unsigned int mask)`
			ptr1: Start address of data to be multipled or accumulated
			ptr2: Same as above
			count: Number of times the operation is performed
			mask: Address mask that correspond to the ring buffer
				A multiply and accumulate operation intrinsic function multiplies and accumulates contents of two data tables.
				Example:
				`short tbl1[]= {a1,a2,a3,a4};`
				`short tbl2[]= {b1,b2,b3,b4};`
				In this case, macw(tbl1, tbl2, 3) calculates a1*b1 +a2*b2+a3*b3. Using a ring buffer

25		MAC.L instruction	`int macl( int *ptr1, int *ptr2, unsigned int count)`
			
			`int macll( int *ptr1, int*ptr2, unsigned int count, unsigned int mask)`
			The parameter specification is the same as those of No. 24.
			Note: macl and macll can be used only when the compiler option **cpu = sh2**, **sh3**, or **sh3e** is specified.
				function, tbl2 can be calculated recursively. The number of calculation times is 2^n.
				Example:
				When the data size is two bytes and the ring buffer mask is four bytes (0xfffffffb or up to 0x4), macwl(tbl1, tbl2, 4, 0xfffffffb) is calculated as a1*b1+a2*b2+a3*b1+ a4*b2.

3.2.4 Notes

1. The offsets (excluding No. 15 to 18) and masks (excluding No.3) shown in table 2.10, Intrinsic Functions, must be constants.
2. The specification range for offsets is +255 bytes when the access size is shown as a byte, +510 bytes when the access size is shown as a word, and +1020 bytes when the access size is shown as a long word.
3. Masks which can be specified for performing logical operations (AND, OR, XOR, or TEST) on a location relative to GBR (global base register) must be within the range of 0 to +255.
4. As GBR is a control register whose contents are not preserved by all functions in this C compiler, take care when changing GBR settings.
5. The multiply and accumulate operation´s instrinsic function does not check for parameters. Therefore, keep the following in mind:
   a. Tables indicated by ptr1 and ptr2 must be aligned to sizes in 2 bytes and 4 bytes, respectively.
   b. Tables indicated by ptr2 in macwl and macwll must be aligned to the size of the ring buffer mask x 2.

3.2.5 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);               /* Set the start address of table to  */
         :                         /* GBR.                               */
   gbr_write_byte( CDATA2, 10);    /* Set 10 to table.cdata2.            */
   gbr_write_long( IDATA2, 100);   /* Set 100 to table.idata2.           */
         :
   if(gbr_read_byte( CDATA2) != 10) /* Refer to table.cdata2.            */
      gbr_and_byte( CDATA2, 10);   /* AND 10 and table.cdata2, and set   */
                                   /* it in table.cdata2.                */
   gbr_or_byte( CDATA2, 0x0F);     /* OR 0x0F and table.cdata2, and set  */
         :                         /* it in table.cdata2.                */
   sleep();                        /* Expand to the sleep instruction    */
}
```

Effective Use of Intrinsic Functions:

1. Allocate frequently accessed object to memory and set the start address of the object to GBR.
2. In step 1., byte data frequently used in logical operations should be declared within 128 bytes of the start address of the structure. As a result, the following instructions can be reduced: start address load instruction necessary for structure accessing and load/store instructions necessary for performing logical operation.

3.2.6 Dividing <machine.h>

<machine.h> is divided as follows to correspond to the SH3 execution mode:

1. <machine.h>: Overall intrinsic functions
2. <smachine.h>: Intrinsic functions that can be used in the privilege mode
3. <umachine.h>: Intrinsic functions except <smachine.h>:

3.3 Section Change Function

A section name to be output in a C program by the compiler can be changed using #pragma section. By using this section change function, you do not need to divide files in units of functions or variables to allocate addresses, which was required previously. The following explains more details on this function.

3.3.1 Description

#pragma section name | value
<source program>
#pragma section

3.3.2 Explanation

Specify a section name using #pragma section name or #pragma section value. A section after a declaration in a source program will be P section name + name (numeric value), D section name + name (numeric value), C section name + name (numeric value) and, B section name + name (numeric value). A default section name becomes valid after #pragma section is declared.

3.3.3 Notes

1. #pragma section must be specified outside the function declaration.
2. A maximum of 64 section names can be declared in one file.

3.3.4 Example

```
#pragma section abc
int a;                    /* a is allocated to section Babc. */
extern const int c=1;     /* c is allocated to section Cabc. */
f( ) {                    /* f is allocated to section Pabc. */
   a=c;
}
#pragma section           /* b is allocated to section B.    */
int b;                    /* g is allocated to section P.    */
g( ) {
   b=c;
}
```

In the above example, when the compile option **section = P = PROG** is specified, **f** and **g** are allocated to section **PROGabc** and **PROG**, respectively.

3.4 Single-Precision Floating-Point Library

A single-precision floating-point library (mathf.h) can be used in addition to an ANSI standard floating-point library (math.h). The single-precision floating-point library consists of functions listed in table 2.11.

3.4.1 Description

A suffix f is added to a double-precision ANSI standard library function name to be a single-precision floating point library function name. If a parameter or return type is **double** or pointer to a **double**-type, it will be **float** or pointer to **float**, respectively. Other specifications are the same as those of the ANSI standard C library.

3.4.2 Notes

Before using this library, be sure to declare #include<mathf.h> and #include<math.h>.

Table 2.11 Function List of Single-Precision Floating-Point Library

Function Name	Description
float acosf (float x)	Anti cosine: acos x
float asinf (float x)	Anti sine: asin x
float atanf (float x)	Anti tangent: atan x
float atan2f (float y, float x)	Anti tangent of a result given by division: atan (x / y)
float cosf (float x)	Cosine: cos x
float sinf (float x)	Sine: sin x
float tanf (float x)	Tangent: tan x
float coshf (float x)	Hyperbolic cosine: cosh x
float sinhf (float x)	Hyperbolic sine: sinh x
float tanhf (float x)	Hyperbolic tangent: tanh x
float expf (float x)	Exponential function: e^x
float frexpf (float x, int *p)	Divided into 0.5 and 1.0, and the square of two and multiplication: suppose result=frexp (x, p), x=2*p x result ($0.5 \leq$ result < 1.0)
float ldexpf (float x, int i)	Square of two and multiplication: x X 2^i
float logf (float x)	Natural logarithm: log x
float log10f (float x)	Common logarithm that has 10 as a base: $\log_{10} x$
float modff (float x, float *p)	Suppose result = modff (x, y), x is divided into integer *p and floating point result
float powf (float x, float y)	Square: x^y
float sqrtf (float x)	Positive square root: $\sqrt{x}$
float ceilf (float x)	Result given by rounding up numbers after a decimal point of x
float fabsf (float x)	Absolute value: | x |
float floorf (float x)	Result given by rounding down numbers after a decimal point of x
float fmodf (float x, float y)	Reminder after division Suppose result = fmodf (x, y) and q quotient, x = q $\times$ y + result

3.5 Japanese Description in String Literals

Japanese can be included in string literals. Select a character code of **euc** or **sjis** option. When this option is omitted, the default setting is specified as table 2.12.

Table 2.12 Default Settings of Japanese Code

Host Computer	Default Settings
SPARC	EUC
HP9000 / 7000	Shift JIS
IBM-PC	Shift JIS

Note: The character code in the object program will be the same as that in the source program. Character constants cannot be specified in Japanese.

3.6 Inline Function

A function name to expand at compilation is specified.

3.6.1 Description

#pragma inline (function name, ...)

3.6.2 Explanation

A function specified by #pragma inline or a function with specifier inline will be expanded where the function is called. However, a function will not be expanded where the function is called in the following cases:

- a function definition exists before the #pragma inline specification
- a function has a flexible parameter
- a parameter address is referenced in a function
- an address of a function to be expanded is used to call a function

3.6.3 Notes

1. Specify #pragma inline before defining 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. If **static** is specified, an external definition will not be created.

3.6.4 Example

Source Program

```
#pragma inline(func)
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;
}
```

3.7 Inline Expansion in Assembly Language

A function that is written in an assembly language is expanded where the function is called in a C source file.

3.7.1 Description

#pragma inline_asm (function name[(size=numeric value)], ...)

3.7.2 Explanation

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.
A return value of a function that is written in an assembly should be set in R0.
The specification (size=numeric value) specifies the size of the assembler inline function.

3.7.3 Notes

1. Specify #pragma inline_asm before defining 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 using registers R8 to R15 in a function written in an assembly language, the contents of these registers must be saved and recovered at the start and end of the function.
5. Do not use RTS at the end of a function written in an assembly language.
6. When using this function, be sure to compile programs using the object type specification option **code=asmcode**.
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 below 0 is specified, an error will occur.

3.7.4 Example

Source Program

```
#pragma inline_asm(rot1)
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
```

3.8 Specifying Two-byte Address Variables

A variable can be allocated to a two-byte address area (H’0000000 to H’0007FFF and H’FFF8000 to H’FFFFFFF).

3.8.1 Description

#pragma abs16 (identifier, ...)

3.8.2 Explanation

A variable specified using an identifier or an address of a function is treated as two-byte data. Then, program size can be reduced.

3.8.3 Notes

1. Directive #pragma abs16 cannot be used to specify an automatic object.
2. Variables declared in directive #pragma abs16 must be allocated in addresses H’0000000 to H’0007FFF or H’FFF8000 to H’FFFFFFF.

3.9 Specifying GBR Base Variables

A variable is accessed using a GBR register with an offset value.

3.9.1 Description

#pragma gbr_base (variable name, ...)
#pragma gbr_base1 (variable name, ...)

3.9.2 Explanation

Variables specified by #pragma gbr_base and #pragma gbr_base1 are allocated to sections $G0 and $G1, respectively. The directive #pragma gbr_base is used when 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 is used when the variable is located in an offset of 128 or more bytes from the address specified in the GBR register, that is, when a variable is in a range that cannot be accessed 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** type data in the $G0 section.

3.9.3 Notes

1. If the total program size after linking with section $G0 exceeds 128 bytes, the correct operation will not be guaranteed. In addition, if there is data that has an offset value that exceeds 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 when linking.
3. When using this function, be sure to set the start address of section $G0 in the GBR register at the beginning of program execution.

3.10 Register Save and Recovery Control

Register contents of a function can be saved or recovered.

3.10.1 Description

#pragma noregsave (function name, ...)
#pragma noregalloc (function name, ...)
#pragma regsave (function name, ...)

3.10.2 Explanation

1. Functions specified by #pragma noregsave do not save or allow the recovery of the contents of registers to guarantee their values (see table 2.6) at the beginning or end of a function.
2. Functions specified by #pragma noregalloc do not save or allow the recovery of the contents of registers to guarantee their values at the beginning or end of a function, but do generate an object before or after the function call. Registers R8 to R14 are not allocated to the object.
3. Functions specified by #pragma regsave do not save or allow the recovery of the contents of registers to guarantee their values at the beginning or end of a function, but do generate an object before or after the function call. Registers R8 to R14 are not allocated to the object.
4. #pragma regsave and #pragma noregalloc can specify the same function at the same time. In this case, the contents of registers R8 to R14 that guarantee their values are saved and recovered at the beginning or end of a function, and generate an object before or after the function call. Registers R8 to R14 are not allocated to the object.

5 Functions specified by #pragma noregsave can be used in the following conditions:

   a. A function is first 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.

3.10.3 Notes

If a function that is specified by #pragma noregsave is called in a way other than explained above, the obtained data is not guaranteed.

3.10.4 Example

```
#pragma noregsave (f)
#pragma noregalloc (g)
#pragma regsave (h)
h ( )
{
   g ( );
   f ( );    /*  function call immediately after function call (f) #pragma noregsave */
}            /*  from function (h) #pragma regsave                                    */

g ( )
{
   f ( );    /*  function call (f) #pragma noregsave from function (h) #pragma      */
             /*  regsave through function (g) #pragma noregalloc                    */
}

f ( )
{
}
```

3.11 Global Variable Register Allocation

Registers are allocated to global variables.

3.11.1 Description

#pragma global_register (<variable name>=<register name>, ...)

3.11.2 Explanation

This function allocates the register specified in <register name> to the global variable specified in <variable name>.

3.11.3 Notes

1. This function is used for a simple or pointer type variable in the global variable. Do not specify a **double** type variable unless **–double=float** option is specified.
2. Only use registers R8 to R14 and FR12 to FR15 (FR12 to FR15: when using SH3E).
3. The initial value cannot be set. In addition, the address cannot be referenced.
4. The specified variable cannot be referenced from the linked side.

3.11.4 Example

```
#pragma global_register(x=R13,y=R14)

int      x;
char     *y;

func1()
{
  X++;
}

func2()
{
  *y=0;
}

func(int  a)
{
  x = a;
  func1();
  func2();
}
```

Section 4 Notes on Programming

This section contains notes on coding programs for the C compiler and troubleshooting when compiling or debugging programs.

4.1 Coding Notes

4.1.1 float Type Parameter Function

Functions must declare prototypes or treat **float** type as **double** type when receiving and passing **float** type parameters. Data cannot be preserved (guaranteed) when a **float** type parameter function without a prototype declaration receives and passes data.

Example:

```
void f (float); ------------------(1)
g ()
{
   float a;
   f (a);
)
void
f (float x)
(
    •
    •
    •
}
```

Function **f** has a **float** type parameter. Therefore, a prototype must be declared as shown in (1) above.

4.1.2 Program Whose Evaluation Order is Not Regulated

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.

4.1.3 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 for one or more constants, error messages will be output at compilation.

Example:

```
        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 for one or more         */
    /* 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.   */

        }
```

4.1.4 Assignment to const Variables

Even if a variable is declared with **const** type, if assignment is done to a variable other than **const** converted from **const** type or if a program compiled separately uses a parameter of a different type, the C compiler cannot detect the error

Example:

1.
```
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.                               */
```

2. 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.                   */
```

4.1.5 Precision of Mathematical Function Libraries

For function acos (x) and asin (x), an error is $x \cong 1$. 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} ($\varepsilon = 2^{-16}$)

4.2 Notes on Program Development

Table 2.13 shows troubleshootings for developing programs from compilation through debugging.

Table 2.13 Troubleshooting

Trouble	Check Points	Solution	References
When linking, error 314, cannot found section, is output	The section name which is output by the C compiler must be specified in capitals in **start** option of linkage editor.	Specify the correct section name.	Section 2.1, Structure of Object Programs in part II, C PROGRAMMING
When linking, error 105, undefined external symbol, is output	If identifiers are mutually referenced by a C program and an assembly program, an underscore must be attached to the symbol in the assembly program.	Refer to parameters with the correct parameters.	Section 2.3.1, External Identifier Reference, in part II, C PROGRAMMING
	Check if the C program uses a library function.	Specify a standard library as the input library at linkage.	Standard library specification: Section 3.5, Correspondence to Standard Libraries, in part I, OVERVIEW AND OPERATIONS
	An undefined reference symbol identifier must not start with a _ _(A run time routine in a standard library must be used.)		Routine: Section 2.1 part III, SYSTEM INSTALLATION
	Check if a standard I/O library function is used in the C program.	Create low level interface routines for linking.	Section 4.6, Creating Low-Level Interface Routines, in part III, SYSTEM INSTALLATION
Debugging at the C source level cannot be performed	**debug** option must be specified at both compilation and linkage.	Specify **debug** option at both compilation and linkage.	Section 3.3, Compiler Options, in part I, OVERVIEW AND OPERATIONS
	A linkage editor of Ver.5.0 or higher must be used.	Use a linkage editor of. Ver.5.0 or higher.	

Table 2.13 Troubleshooting (cont)

Trouble	Check Points	Solution	References
When linking, error No. 108 relocation size overflow is output	Check if an offset value of a variable specified using a GBR base is within the range.	Delete #pragma gbr_base/ gbr_base1 declaration for data beyond the range.	Section 3.9, Specifying GBR Base Variables, in part II, C PROGRAMMING
When linking, error No. 104 duplicate symbol is output	Check if a variable or function whose name is the same as that of other variables or functions exists in more than one file.	Change the name of the variable or function, or specify static.	
	Check if a variable or function is externally defined in a header file to be included in more than one file (the above is the same in the case of a function specified (#pragma inline/ inline_asm).	Specify static.	Section 3.6.3, Notes and 3.7.3, Notes, in part II, C PROGRAMMING

PART III

SYSTEM INSTALLATION

Section 1 Overview of System Installation

Part III describes how to install object programs generated by the C compiler on an SH system. Before installation, memory allocation and execution environment for the object program must be specified.

Memory Allocation: Allocate a stack area, a heap area, and each section of a C-compiler-generated object program in ROM or RAM on a SH system.

Execution Environment Setting for a C-Compiler-Generated Object Program: Set the execution environment by register initialization, memory area initialization, and C program initiation. Write these processing functions in assembly language.
If C library functions such as the I/O function are used, library must be initialized when setting the execution environment specification.

Section 2 describes how to allocate C programs in memory area and how to specify linkage editor's commands that actually allocate a program in memory area, using examples.

Section 3 describes items to be specified in execution environment setting and execution environment specification programs.

Section 4 describes how to create C library function initialization and low-level routines.

Note: If I/O function (stdio.h) and memory allocation function (stdlib.h) are used, the user must create low-level I/O routines and memory allocation routines appropriate to the user system.

Section 2 Allocating Memory Areas

To install an object program generated by the C 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 the size of each area is determined and how to allocate an area in memory.

2.1 Static Area Allocation

2.1.1 Data to be Allocated in Static Area

Allocate sections of object programs such as program area, constant area, initialized data area, and non-initialized data area to the static area.

2.1.2 Static Area Size Calculation

Calculate the static area size by adding the size of C-compiler-generated object program and that of library functions used by the C program. After object program linkage, determine the static area size from each section size including library size output on a linkage map listing. Before object program linkage, the approximate size of the static area can be determined from the section size information on a compile listing. Figure 3.1 shows an example of section size information.

```
* * * * * * * SECTION SIZE INFORMATION * * * * * * *
PROGRAM   SECTION(P):       0x00004A Byte(s)
CONSTANT  SECTION(C):       0x000018 Byte(s)
DATA      SECTION(D):       0x000004 Byte(s)
BSS       SECTION(B):       0x000004 Byte(s)

TOTAL PROGRAM SIZE: 0x00006A Byte(s)
```

Figure 3.1 Section Size Information

If the standard library is not used, calculate the static area size by adding the memory area size used by sections shown in section size information. However, 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 required for C program execution. Accordingly, link the standard library to the C source program even if library functions are not used in the C source program.

The C compiler provides the standard library including C library functions (based on the C language specifications), and arithmetic routines (runtime routines required for C program execution). The size required for run time routines must also be added to the memory area size in the same way as C library functions.

The run time routine used by the C programs are output as external reference symbols in the assembly programs generated by the C compiler (option **code = asmcode**). The user can see the run time routine names used in the C programs through the external reference symbols.

The following shows the example of C program and assembly program listings.

C program

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

Assembly program output by the C compiler

```
.IMPORT    _ _divls     ; An external reference definition for the run time 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
```

In the above example, **_ _divls** is a run time routine used in the C program.

2.1.3 ROM and RAM Allocation

When allocating a program to memory, allocate static areas to either ROM and RAM as shown below.

Program area (section P): ROM
Constant area (section C): ROM
Non-initialized data area (section B): RAM
Initialized data area (section D): ROM and RAM (for details, refer to the following section)

2.1.4 Initialized Data Area Allocation

The initialized data area contains data with initial value. Since the C language specifications allow the user to modify initialized data in programs, the initialized data area must be allocated to ROM when linking and is copied to RAM before program execution. Therefore, the initialized data area must be allocated in both ROM and RAM.

However, if the initialized data area contains only static variables that are not modified during program execution, the initialized data needs to be allocated only to the ROM area. In this case, the data does not need to be allocated to the RAM area.

2.1.5 Memory Area Allocation Example and Address Specification at Program Linkage

Each program section must be addressed by the option or subcommand of the linkage editor when the absolute load module is created, as described below.

Figure 3.2 shows an example of allocating static areas.

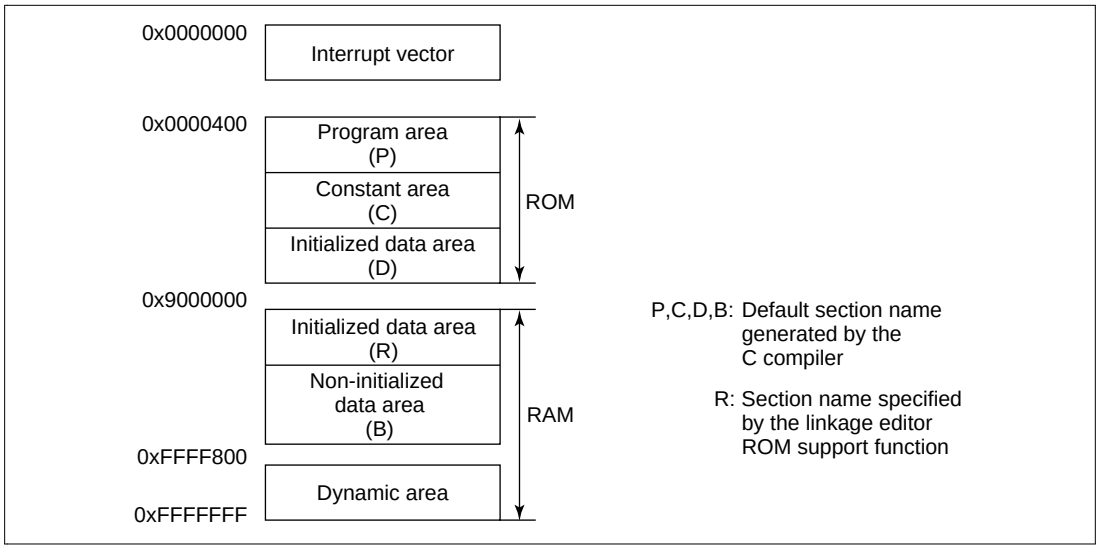


Figure 3.2 Static Area Allocation

Specify the following subcommands when allocating the static area as shown in figure 3.2.

```
        :
ROMΔ(D,R)                ----------------(1)
STARTΔP,C,D(400),R,B(9000000)--------(2)
        :
```

Description:

(1) Define section R having the same size as section D, in the output load module. To reference the symbol allocated to section D, reallocate to the address of section R and reference to the symbol in section R. Sections D and R are allocated to initialized data section in ROM and RAM, respectively.

(2) Allocate sections P, C, and D to internal ROM starting from address 0x400 and allocate sections R and B to RAM starting from address 0x9000000.

2.2 Dynamic Area Allocation

2.2.1 Dynamic Areas

Two types of dynamic areas for C program are used:

1. Stack area
2. Heap area (used by the memory allocation library functions)

2.2.2 Dynamic Area Size Calculation

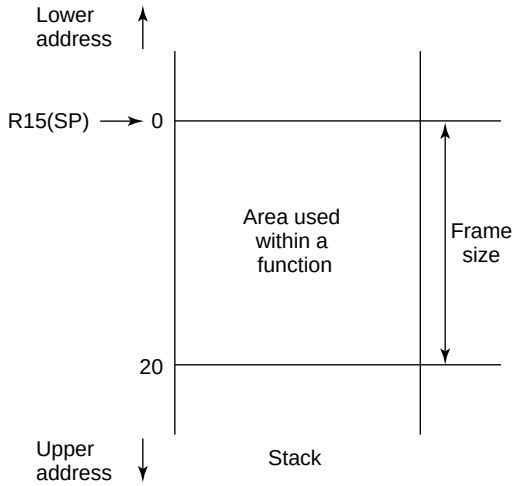
Stack Area: The stack area used in C programs is allocated each time a function is called and is deallocated each time a function is returned. The total stack area size is calculated based on the stack size used by each function and the nesting of function calls.

Stack Area Used by Each Function: The object list (frame size) output by the C 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.

```
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
                                 :
```

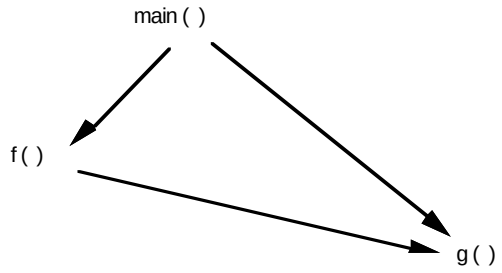


The size of the stack area used by a function is equal to frame size. Therefore, in the above example, the stack size used by the function **h** is 20 bytes which is shown as **frame size = 20** in COMMENT in OBJECT LISTING.

For details on the size of parameters to be pushed onto the stack, refer to the description of parameter and return value setting and referencing in section 2.3.2, Setting and Referencing Parameters and Return Values, Function Call Interface, in Part II, C Programming.

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

Example: Figure 3.3 illustrates the function call nestings and stack size.



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

Figure 3.3 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 3.1.

Table 3.1 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 3.1, the maximum size of stack area required for the longest function calling route should be determined (80 bytes in this example) and this size of memory should be allocated in RAM.

When using standard library functions, the stack area sizes for library functions must also be accounted for. Refer to the Standard Library Memory Stack Size Listing, included with the C compiler package.

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

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**, or **realloc**) in the C program. An additional 4 bytes must be summed for one call because a 4-byte management area is used every time a memory management library function allocates an area.

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

Note: Areas released by the free 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.

2.2.3 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 SH3 and SH3E differ from that of the SH1 and SH2, interrupt handlers are necessary. The heap area is determined by the initial specification in the low-level interface routine (**sbrk**). For details on stack and heap areas, refer to section 3.1, Vector Table Setting (**VEC_TBL**), and section 4.6, Creating Low-Level Interface Routine in part III, System Installation, respectively.

Section 3 Setting the Execution Environment

This section describes the environment required for C program execution. A C program environment specification program must be created according to the user system specifications because the C program execution environment differs depending on the user system. In this section, basic C program execution specification, where no C library function is used, is described as an example. Refer to section 4, Setting the C Library Function Execution Environment in part III, System Installation, for details on using C library functions when using C library functions, low-level I/O interface routine, or memory allocation routine.

Figure 3.4 shows an example of program configuration.

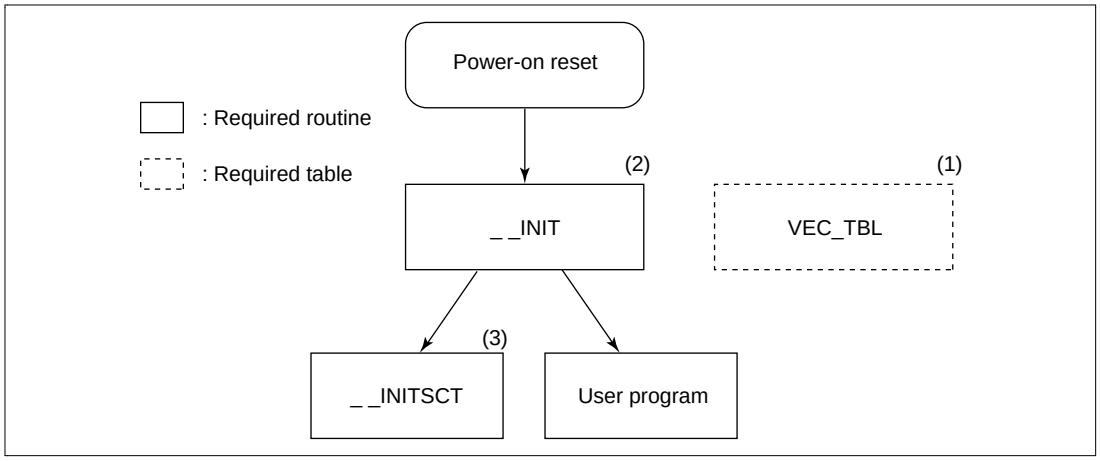


Figure 3.4 Program Configuration (No C Library Function is Used)

Each routine is described below.

Vector table setting (VEC_TBL) (shown as (1) in figure 3.4): Sets the vector table so as to initiate register initialization program **_ _INIT** and set the stack pointer (SP) by power-on reset. Since the interrupt operation of the SH3 and SH3E differ from those of the SH1 and SH2, interrupt handlers are necessary.

Initialization (_ _INIT) (shown as (2) in figure 3.4): Initializes registers and sequentially calls initialization routines.

Section initialization (_ _INITSCT) (shown as (3) in figure 3.4): Clears the non-initialized data area with zeros and copies the initialized data area in ROM to RAM.

The following describes how each process is implemented (in the order as described above).

3.1 Vector Table Setting (VEC_TBL)

To call register initialization routine **_ _INIT** at power-on reset, specify the start address of function **_ _INIT** 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 SH3 and SH3E differ from those of the SH1 and SH2, interrupt handlers are necessary. When the user system executes interrupt handling, interrupt vector settings are also performed in the **VEC_TBL** routine. The coding example of **VEC_TBL** is shown below.

Example:

```
.SECTION VECT,DATA,LOCATE=H'0000
                        ; Assigns section VECT to address H'0 by the SECTION directive.
 .IMPORT    _ _INIT
 .IMPORT    _IRQ0
 .DATA.L    _ _INIT      ; Assigns the start address of _ _INIT to addresses H'0x0 to H'0x3.
 .DATA.L    (a)          ; Assigns the SP to addresses H'0x4 to H'0x7.
                         ; (a):  The highest address of the stack
 .ORG       H'00000100
 .DATA.L    _IRQ0        ; Assigns the start address of IRQ0 to addresses H'0x100 to H'0x103
 .END
```

3.2 Initialization (_ _INIT)

_ _INIT initializes registers, calls initialization routine sequentially, and then calls the main function. The coding example of this routine is shown below.

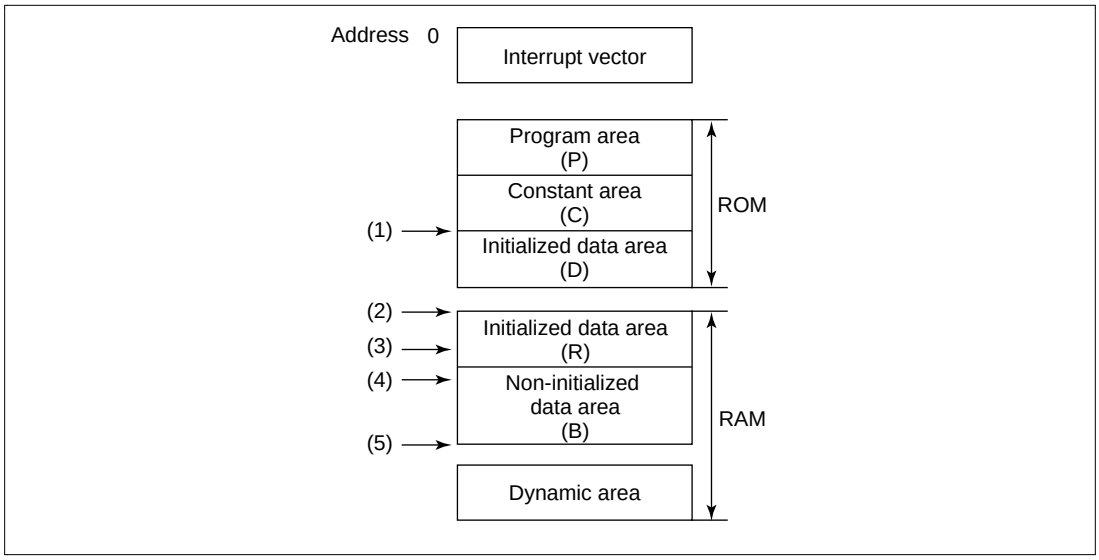
Example:

```
extern void _INITSCT (void);
extern void main (void);

void _INIT()
{

   _INITSCT();     /* Calls section initialization routine          */
                   /* _INITSCT.                                     */

   main();         /* Calls main routine _main.                     */

   for( ; ; )      /* Branches to endless loop after executing main */
    ;              /* function and waits for reset.                 */

}
```

3.3 Section Initialization (_ _INITSCT)

To set the 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 in ROM



To obtain the above addresses, create the following assembly programs and link them 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 non-initialized data area and initialized data area section names specified with the compiler option **section**. 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 as shown below.

Example:

Section initialization routine

```
extern int  *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;

extern 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.

Section 4 Setting the C Library Function Execution Environment

To use C library functions, they must be initialized to set the C program execution environment. To use I/O (**stdio.h**) and memory allocation (**stdlib.h**) functions, or to use the C library function to terminate program processing, low-level I/O and memory allocation routines must be created for each system.

This section describes how to set the C program execution environment when C library functions are used. Figure3.5 shows the program configuration when C library functions are used.

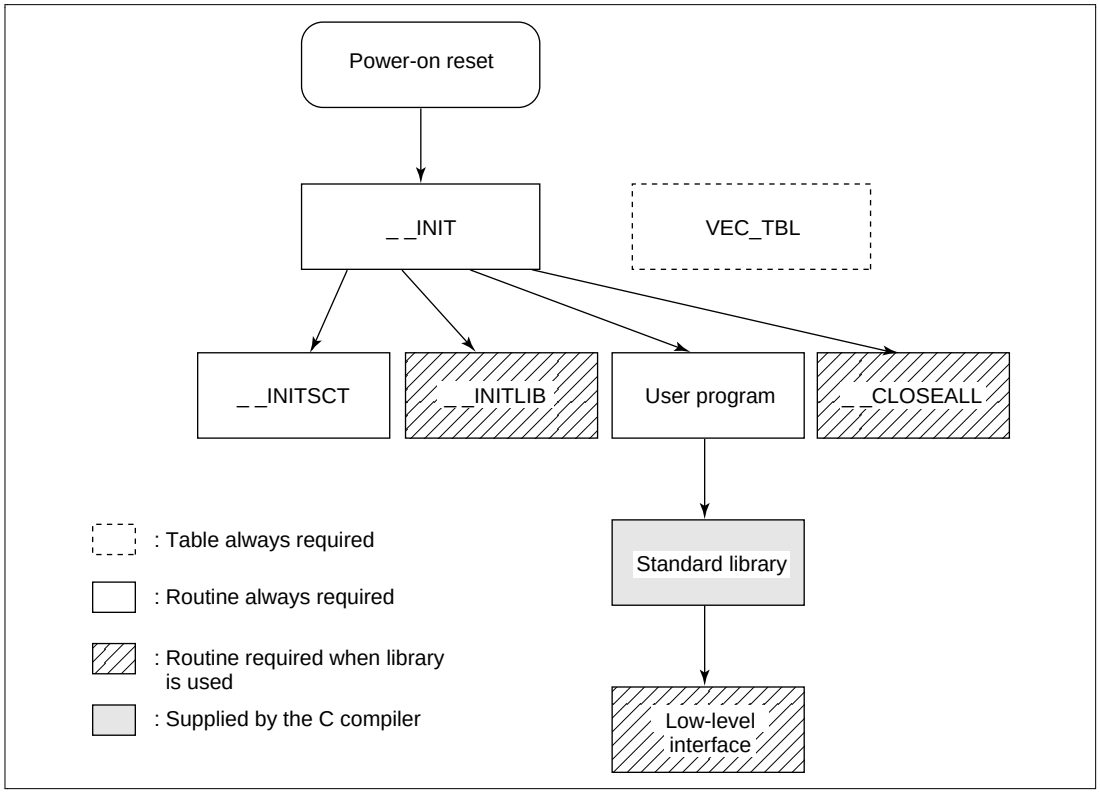


Figure 3.5 Program Configuration When C Library Functions are Used

To use a C library function **exit**, **onexit**, or **abort**, which performs program termination processing, the C library function that corresponds to the user system must be created beforehand. For details on a program example, refer to Appendix D, Creating Termination Functions. If you use a C library function assert macro, you must create an abort function first.

Each routine is required to execute library functions as follows.

Vector Table Setting (VEC_TBL): Sets the vector table to initiate register initialization program (**_ _INIT**) and set the stack pointer (SP) at power-on reset. Since the interrupt of the SH3 and SH3E differ from the SH1 and SH2, interrupt handlers are necessary.

Initializing Registers (_ _INIT): 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 C Library Functions (_ _INITLIB): Initializes C library functions required to be initialized and prepares standard I/O functions.

Closing Files (_ _CLOSEALL): Closes all files with open status.

Low-Level Interface Routine: Interfaces library functions and user system when standard I/O and memory management library functions are used.

Creation of the above routines is described below.

4.1 Vector Table Setting (VEC_TBL)

Same as when no C library function is used. For details, refer to section 3, Setting the Execution Environment, in part III, System Installation.

4.2 Initializing Registers (_ _INIT)

Initializes registers and sequentially calls the initialization routine **_ _INITLIB** and file closing routine **_ _CLOSEALL**. The coding example of **_ _INIT** is shown below. Since the interrupt operation of the SH3 and SH3E differ from those of the SH1 and SH2, interrupt handlers are necessary.

Example:

```
extern void _INITSCT(void);
extern void _INITLIB(void);
extern void main(void);
extern void _CLOSEALL(void);

void _INIT(void)
{
  _INITSCT();         /* Calls section initialization routine _ _INITSCT.   */
  _INITLIB();         /* Calls library initialization routine _ _INITLIB.   */
  main();             /* Calls C program main function _main.               */
  _CLOSEALL();        /* Calls file close routine _ _CLOSEALL.              */

  for( ; ; )          /* Branches to endless loop after executing main      */
    ;                 /* function and waits for reset.                      */
}
```

4.3 Initializing Sections (_ _INITSCT)

Same as when the C library functions are not used. For details, refer to section 3, Setting the Execution Environment in part III, System Installation.

4.4 Initializing C Library Functions (_ _INITLIB)

Some C library functions must be initialized before being used. The following description assumes the case when the initialization is performed in **_ _INITLIB** in the program initiation routine.

To perform initialization, the following must be considered.

1. **errno** indicating the library error status must be initialized for all library functions.
2. When using each function of **<stdio.h>** and **assert** macro, standard I/O library function must be initialized. The low-level interface routine must be initialized according to the user low-level initialization routine specification if required.
3. When using the **rand** and **strtok** functions, library functions other than the standard I/O must be initialized.

Library function initialization program example is shown below.

Example:

```
#include <errno.h>

extern void _INIT_LOWLEVEL(void) ;
extern void _INIT_IOLIB(void) ;
extern void _INIT_OTHERLIB(void) ;


void _INITLIB(void)          /* Deletes an underline from symbol name  */
                             /* used in the assembly routine           */
{
   errno=0;                  /* Initializes library functions commonly */

   _INIT_LOWLEVEL( ) ;       /* Calls low-level interface              */
                             /* initialization routine                 */
   _INIT_IOLIB( ) ;          /* Calls standard I/O initialization      */
                             /* routine                                */
   _INIT_OTHERLIB( ) ;       /* Calls initialization routine other     */
                             /* than that for standard I/O             */
}
```

The following shows examples of initialization routine (**_INIT_IOLIB**) for standard I/O library function and initialization routine (**_INIT_OTHERLIB**) for other standard library function. Initialization routine (**_INIT_LOWLEVEL**) for low-level interface routine must be created according to the user low-level interface routine's specifications.

4.4.1 Creating Initialization Routine (_INIT_IOLIB) for Standard I/O Library Function

The initialization routine for standard I/O library function initializes **FILE**-type data used to reference files and open the standard I/O files. The initialization must be performed before opening the standard I/O files (figure 3.6).

The following shows an example of **_INIT_IOLIB**.

Example:

```
#include <stdio.h>
void _INIT_IOLIB(void)
{
  FILE *fp ;

/*Initializes FILE-type data*/

  for (fp=_iob; fp<_iob+_NFILE; fp++){
     fp -> _bufptr=NULL ;                    /*Clears buffer pointer   */
     fp -> _bufcnt=0 ;                       /*Clears buffer counter   */
     fp -> _buflen=0 ;                       /*Clears buffer length    */
     fp -> _bufbase=NULL ;                   /*Clears base pointer     */
     fp -> _ioflag1=0 ;                      /*Clears I/O flag         */
     fp -> _ioflag2=0 ;
     fp -> _iofd=0 ;
  }

               /*Opens standard I/O file  */

                  *1
  if (freopen( "stdin" , "r", stdin)==NULL)  /*Opens standard input file */
    stdin->_ioflag1=0xff ;                   /*Disables file access*2    */
    stdin->_ioflag1 |= _IOUNBUF ;            /*No data buffering*3       */
                  *1
  if (freopen( "stdout" , "w", stdout)==NULL)/*Opens standard output file*/
    stdout-> _ioflag1=0xff ;
    stdout->_ioflag1 |= _IOUNBUF ;
                  *1
  if (freopen( "stderr", "w", stderr)==NULL) /*Opens standard error file */
    stderr-> _ioflag1=0xff ;
    stderr->_ioflag1 |= _IOUNBUF ;
 }
```

Notes: 1. Standard I/O file names are specified. These names are used by the low-level interface routine **open**.

2. If file could not be opened, the file access disable flag is set.

3. For equipment that can be used in interactive mode such as a console, the buffering disable flag is set.

```
/*Declares FILE-type data in the C language*/

        #define _NFILE 20
        struct _iobuf{
                    unsigned char *_bufptr;   /*Buffer pointer      */
                    long           _bufcnt;   /*Buffer counter      */
                    unsigned char *_bufbase;  /*Buffer base pointer */
                    long           _buflen;   /*Buffer length       */
                    char           _ioflag1;  /*I/O flag            */
                    char           _ioflag2;  /*I/O flag            */
                    char           _iofd;     /*I/O flag            */
        }_iob[_NFILE];
```

Figure 3.6 FILE-Type Data

4.4.2 Creating Initialization Routine (_INIT_OTHERLIB) for Other Library Function

The following shows an example of initial setting program of C library function (rand function and strtok function) that is necessary for initial setting beside the standard I/O.

```
#include <stddef.h>

extern char *_s1ptr;
extern void srand(unsigned int) ;

void _INIT_OTHERLIB(void)
{

   srand(1) ;        /*Sets initial value when rand function is used */
   _s1ptr=NULL ;     /*Initializes the pointer used in the strtok    */
                     /* function                                     */
}
```

4.5 Closing Files (_ _CLOSEALL)

When a program ends normally, all open files must be closed. Usually, the data destined for a file is stored in a memory buffer. When the buffer becomes full, data is output to an external storage device. Therefore, if the files are not closed, data remaining in buffers is not output to external storage devices and will be lost.

When an program is installed in a device and executed, the program will not end unless it finishes its operation. However, if the **main** function is terminated by a program error, all open files must be closed.

The following shows an example of **_ _CLOSEALL**.

Example:

```
#include <stdio.h>

void _CLOSEALL(void)       /* Deletes an underscore character       */
                           /* from symbol name in assembly routine  */
{

      int i;

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

                           /*Checks that file is open*/

            if(_iob[i]._ioflag1 & ( _IOREAD|_IOWRITE|_IORW))

                           /*Closes open files*/

                     fclose(&_iob[i]) ;
}
```

4.6 Creating Low-Level Interface Routines

Low-level interface routines must be supplied for C programs that use the standard input/output or memory management library functions. Table 3.2 shows the low-level interface routines used by standard library functions.

Table 3.2 Low-Level Interface Routines

Name	Explanation
open	Open a file
close	Close a file
read	Reads data from a file
write	Writes data to a file
lseek	Sets the file read/write position for data
sbrk	Allocates a memory area

Refer to the attached Standard Library Memory Stack Size Listing for details on low-level interface routines required for each C library function.

Initialization of low-level interface routines must be performed when the program is started. For more information, see the explanation concerning the **_INIT_LOWLEVEL** function in section 4.4, Initializing C Library Functions (**_ _INITLIB**).

The rest of this section explains the basic concept of low-level input and output, and gives the specifications for each interface routine. Refer to appendix E, Examples of Low-Level Interface Routines, for details on the low-level interface routines that run on the SH-series simulator debugger.

4.6.1 Concept of I/O Operations

Standard input/output library functions manage files using the **FILE**-type data. Low-level interface routines manage files using file numbers (positive integers) which correspond directly to actual files.

The open routine returns a file number for a given file name. The open routine must determine the following, so that other functions can access information about a file using the file number:

- File device type (console, printer, disk, etc.)
  (For a special device such as a console or printer file, the user chooses a specific file name that can be recognized uniquely by the **open** routine.)
- Information such as the size and start address of the buffer used for the file
- For a disk file, the offset (in bytes) from the beginning of the file to the next read/write position.

The input and output is determined by the **read** and **write** routine, respectively, or the start position for read/write operations is determined by the **lseek** routine according to the information determined by the **open** routine.

If buffers are used, the **close** routine outputs the contents to their corresponding files. This allows the areas of memory allocated by the **open** routine to be reused.

4.6.2 Low-Level Interface Routine Specifications

This section explains the specifications for creating low-level interface routines, gives examples of actual interfaces and explains their operations, and notes on implementation.

The interface for each routine is shown using the format below.

Create each interface routine by assuming that the prototype declaration is made.

Example:

(Routine name)

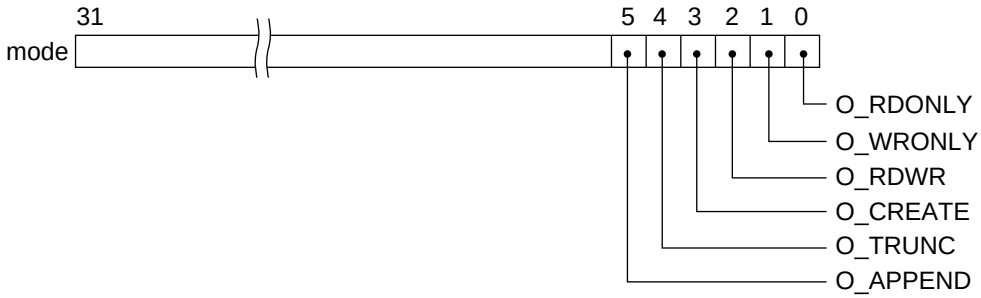
Purpose	**(Purpose of the routine)**			
Interface	(Shows the interface as a C function declaration)			
Parameters	No.	Name	Type	Meaning
	1	(Parameter name)	(Parameter type)	(Meaning of the parameter)
	•	•	•	•
	•	•	•	•
	•	•	•	•
Return value	Type	(Type of return value)		
	Normal	(Return value for normal termination)		
	Abnormal	(Return value for abnormal termination)		

open routine

Purpose	**Opens a file**			
Interface	`int open (char *name,` `  int  mode);`			
Parameters	No.	Name	Type	Meaning
	1	`name`	Pointer to **char**	String literal indicating a file name
	2	mode	**int**	Processing specification
Return value	Type	**int**		
	Normal	File number of the file opened		
	Abnormal	–1		

Explanation: The **open** routine opens the file specified by the first parameter (file name) and returns a file number. The **open** routine must determine the file device type (console, printer, disk, etc.) and assign this information to the file number. The file type is referenced using the file number each time a read/write operation is performed.

The second parameter (mode) gives processing specifications for the file. The effect of each bit of this parameter is explained as follows:



Description:

(1) O_RDONLY (bit 0)

If this bit is 1, the file becomes read only.

(2) O_WRONLY (bit 1)

If this bit is 1, the file becomes write only.

(3) O_RDWR (bit 2)

If this bit is 1, the file becomes read/write.

(4) O_CREATE (bit 3)

If this bit is 1 and the file indicated by the file name does not exist, a new file is created.

(5) O_TRUNC (bit 4)

If this bit is 1 and the file indicated by the file name exists, the file contents are discarded and the file size is set to zero.

(6) O_APPEND (bit 5)

If this bit is 1, the read/write position is set to the end of the file. If this bit is 0, the read/write position is set to the beginning of the file.

An error is assumed if the file processing specifications contradict with the actual characteristics of the file.

The **open** routine returns a file number (positive integer) which can be used by the **read**, **write**, **lseek**, and **close** routines, provided the file opens normally. The relationship between file numbers and actual files must be managed by the low-level interface routines. The **open** routine returns a value of –1 if the file fails to open properly.

close routine

Purpose	**Closes a file**			
Interface	`int close(int fileno);`			
Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the file to be closed
Return value	Type	**int**		
	Normal	0		
	Abnormal	–1		

Explanation: The file number, determined by the **open** routine, is given as the parameter. Release the area of memory allocated by the **open** routine for file management information, so that it can be reused. If buffers are used, the contents are output to their corresponding files. Zero is returned if the file closes normally. Otherwise, –1 is returned.

read routine

Purpose	**Reads data from a file**			

Interface	`int read (int fileno,`
	`  char *buf,`
	`  unsigned int count);`			

Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the file to be read
	2	`buf`	Pointer to **char**	Area to be used to store the read data
	3	`count`	**unsigned int**	Byte length of data to be read
Return value	Type	**int**		
	Normal	Byte length of the data actually read		
	Abnormal	–1		

Explanation: The read routine loads data from the file indicated by the first parameter (**fileno**) into the area indicated by the second parameter (**buf**). The amount of data to be read is indicated by the third parameter (**count**). If an end of file is encountered during a read, less than the specified number of bytes are read. The file read/write position is updated using the byte length of the data actually read. If data is read normally, the routine returns the number of bytes of the data read. Otherwise, the **read** routine returns a value of –1.

write routine

Purpose	**Writes data to a file**			

Interface	`int write (int  fileno,`
	`char *buf,`
	`unsigned int count);`			

Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number
	2	`buf`	Pointer to **char**	Area storing data to be written in the file
	3	`count`	**unsigned int**	Byte length of the data to be written
Return value	Type	**int**		
	Normal	Byte length of the data actually written		
	Abnormal	–1		

Explanation: The **write** routine outputs data, whose byte length is indicated by the third parameter (**count**), from the area indicated by the second parameter (**buf**) into the file indicated by the first parameter (**fileno**). If the device (such as a disk) where a file is stored becomes full, data less than the specified byte length is written to the file. If zero is returned as the byte length of data actually written several times, the routine assumes that the device is full and sends a return value of –1. The file read/write position is updated using the byte length of data actually written. If the routine ends normally, it returns the byte length of data actually written. Otherwise, the routine returns a value of –1.

lseek routine

Purpose	**Determines the next read/write position in a file**			

Interface	`long lseek (int fileno,`
	`long offset,`
	`int base);`			

Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the target file
	2	`offset`	**long**	Offset in bytes from specified point in the file
	3	`base`	**int**	Base used for offset (bytes)
Return value	Type	**long**		
	Normal	The offset (bytes) from the beginning of the file for the next read/write position		
	Abnormal	–1		

Explanation: The **lseek** routine determines the next read/write position as an offset in bytes. The next read/write position is determined according to the third parameter (**base**) as follows:

1. Base = 0
   The second parameter gives the new offset relative to the beginning of the file.
2. Base = 1
   The second parameter is added to the current position to give the new offset.
3. Base = 2
   The second parameter is added to the file size to give the new offset.

An error occurs if the file is on an interactive device (such as a console or printer), the new offset value is negative, or the new offset value exceeds the file size in the case of 1 or 2, above.

If **lseek** correctly determines a new file position, the new offset value is returned. This value indicates the new read/write position relative to the beginning of the file. Otherwise, the **lseek** routine returns a value of –1.

sbrk routine

Purpose	**Allocates a memory area**			
Interface	`char *sbrk (` `  unsigned long size);`			
Parameters	No.	Name	Type	Meaning
	1	`size`	**unsigned long**	Size of the area to be allocated (in bytes)
Return value	Type	Pointer to **char**		
	Normal	Start address of the allocated area		
	Abnormal	(char *) – 1		

Explanation: The size of the area to be allocated is given as a parameter. Create the **sbrk** routine so that consecutive calls allocate consecutive areas beginning with the lowest available address. An error will occur if there is insufficient memory. If the routine ends normally, it returns the start address of the allocated area. Otherwise, the routine returns (char *) – 1.

PART IV

ERROR MESSAGES

Section 1 Error Messages

This section gives lists of error messages in order of error number. A list of error messages are provided for each level of errors (I = Information error, W=Warning error, E = Error, F = Fatal error, or (–) = Internal error) in the format below.

Error number (Error Level: I, W, E, F, or (–)) Error Message Explanation

0001 (I) Character combination /* in comment
String literal /* exists in comment.

0002 (I) No declarator
A declaration without a declarator exists.

0003 (I) Unreachable statement
A statement that will not be executed exists.

0004 (I) Constant as condition
A constant expression is specified as condition for **if** or **switch** statement.

0005 (I) Precision lost
Precision may be lost when assigning with type conversion a right hand side value to the left hand side value.

0006 (I) Conversion in argument
A function parameter expression is converted into a parameter type specified in the prototype declaration.

0008 (I) Conversion in return
A return statement expression is converted into a value type that should be returned from a function.

0010 (I) Elimination of needless expression
A needless expression exists.

0011 (I) Used before set symbol: “variable name”
A variable is used before setting its value.

0015 (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.

0100 (I) Function "function name" not optimized
A function which is too large cannot be optimized.

0200 (W) No prototype function
There is no prototype declaration.

1000 (W) Illegal pointer assignment
A pointer is assigned to a pointer with a different data type.

1001 (W) Illegal comparison in "operator"
The operands of the binary operator == or != are a pointer and an integer other than 0.

1002 (W) Illegal pointer for "operator"
The operands of the binary operator ==, !=, >, <, >=, or <= are pointers assigned to different types.

1005 (W) Undefined escape sequence
An undefined escape sequence (a character following a backslash) is used in a character constant or string literal.

1007 (W) Long character constant
A character constant consists of two or more characters.

1008 (W) Identifier too long
An identifier's length exceeds 250 characters.

1010 (W) Character constant too long
A character constant consists of four or more characters.

1012 (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.

1013 (W) Integer constant overflow
The value of unsigned long integer constant exceeds the limit. Assumes a value ignoring the overflown upper bits.

1014 (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.

1015 (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.

1016 (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.

1017 (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.

1019 (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.

1020 (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.

1021 (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. #pragma is ignored.

1022 (W) First operand of operator is not lvalue
The first operand operator cannot be the lvalue.

1023 (W) Can not convert Japanese code “code” to output type
A Japanese code “code” cannot be converted to the specified output code.

1200 (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.

1201 (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.

1300 (W) Command parameter specified twice
The same SH C compiler option is specified more than once. Uses the last specified compiler option.

1400 (W) Function "function name" in #pragma inline is not expanded
A function specified using #pragma inline could not be expanded where the function is callled. Compiling processing continues.

2000 (E) Illegal preprocessor keyword
An illegal keyword is used in a preprocessor directive.

2001 (E) Illegal preprocessor syntax
There is an error in preprocessor directive or in a macro call specification.

2002 (E) Missing ","
A comma (,) is not used to delimit two arguments in a #define directive.

2003 (E) Missing ")"
A right parenthesis ( )) does not follow a name in a defined expression. The defined expression determines whether the name is defined by a #define directive.

2004 (E) Missing ">"
A right angle bracket (>) does not follow a file name in an #include directive.

2005 (E) Cannot open include file "file name"
The file name specified by an #include directive cannot be opened.

2006 (E) Multiple #define's
The same macro name is redefined by #define directives.

2008 (E) Processor directive #elif mismatches
There is no #if, #ifdef, #ifndef, or #elif directive corresponding to an #elif directive.

2009 (E) Processor directive #else mismatches
There is no #if, #ifdef, or #ifndef directive corresponding to an #else directive.

2010 (E) Macro parameters mismatch
The number of macro call arguments and the number of macro definition arguments are not equal.

2011 (E) Line too long
After macro expansion, a source program line exceeds the compiler limit.

2012 (E) Keyword as a macro name
A preprocessor keyword is used as a macro name in a #define or #undef directive.

2013 (E) Processor directive #endif mismatches
There is no #if, #ifdef, or #ifndef directive corresponding to an #endif directive.

2014 (E) Missing #endif
There is no #endif directive corresponding to an #if, #ifdef, or #ifndef directive, and the end of file is detected.

2016 (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.

2017 (E) Missing ”
A closing double quotation mark (") does not follow a file name in an #include directive.

2018 (E) Illegal #line
The line count specified by a #line directive exceeds the limit.

2019 (E) File name too long
The length of a file name exceeds 128 characters.

2020 (E) System identifier "name" redefined
The name of the defined symbol is the same as that of the run time routine.

2100 (E) Multiple storage classes
Two or more storage class specifiers are used in a declaration.

2101 (E) Address of register
An unary-operator & is used for a variable that has a register storage class.

2102 (E) Illegal type combination
A combination of type specifiers is illegal.

2103 (E) Bad self reference structure
A struct or union member has the same data type as its parent.

2104 (E) Illegal bit field width
A constant expression indicating the width of a bit field is not an integer or it is negative.

2105 (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.

2106 (E) Extern variable initialized
A compound statement specifies an initial value for an extern storage class variable.

2107 (E) Array of function
An array with a function type is specified.

2108 (E) Function returning array
A function with an array return value type is specified.

2109 (E) Illegal function declaration
A storage class other than extern is specified in the declaration of a function variable used in a compound statement.

2110 (E) Illegal storage class
The storage class in an external definition is specified as auto or register.

2111 (E) Function as a member
A member of a struct or union is declared as a function.

2112 (E) Illegal bit field
A type other than an integer type is specified for a bit field.

2113 (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.

2114 (E) Multiple variable declarations
A variable name is declared more than once in the same scope.

2115 (E) Multiple tag declarations
A struct, union, or enum tag name is declared more than once in the same scope.

2117 (E) Empty source program
There are no external definitions in the source program.

2118 (E) Prototype mismatch “function name”
A function type differs from the one specified in the declaration.

2119 (E) Not a parameter name “parameter name”
An identifier not in the function parameter list is declared as a parameter.

2120 (E) Illegal parameter storage class
A storage class other than register is specified in a function parameter declaration.

2121 (E) Illegal tag name
The combination of a struct, union, or enum with tag name differs from the declared combination.

2122 (E) Bit field width 0
The width of a bit field specifying a member name is 0.

2123 (E) Undefined tag name
An undefined tag name is specified in an enum declaration.

2124 (E) Illegal enum value
A non-integral constant expression is specified as a value for an enum member.

2125 (E) Function returning function
A function with a function type return value is specified.

2126 (E) Illegal array size
The value specifying the number of array elements is out of range of 1 to 2147483647.

2127 (E) Missing array size
The number of elements in an array is not specified where it is required.

2128 (E) Illegal pointer declaration for "*"
A type specifier other than const or volatile is specified following an asterisk (*), which indicates a pointer declaration.

2129 (E) Illegal initializer type
The initial value specified for a variable is not a type that can be assigned to another variable.

2130 (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.

2131 (E) No type nor storage class
Storage class or type specifiers is not given in an external data definition.

2132 (E) No parameter name
A parameter is declared even though the function parameter list is empty.

2133 (E) Multiple parameter declarations
Either a parameter name is declared in a macro function definition parameter list more than once or a parameter is declared inside and outside the function declarator.

2134 (E) Initializer for parameter
An initial value is specified in the declaration of a parameter.

2135 (E) Multiple initialization
A variable is initialized more than once.

2136 (E) Type mismatch
An extern or static storage class variable or function is declared more than once with different data types.

2137 (E) Null declaration for parameter
An identifier is not specified in the function parameter declaration.

2138 (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 members of a union are scalar.

2139 (E) No parameter type
A type is not specified in a function parameter declaration.

2140 (E) Illegal bit field
A bit field is used in a union.

2141 (E) Struct has no member name
The member name of a struct is not specified.

2142 (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 value type
3. To explicitly specify that a function whose prototype is declared does not have a parameter

2143 (E) Illegal static function
There is a function declaration with a static storage class function that has no definition in the source program.

2144 (E) Type mismatch
Variables or functions with the same name which have an extern storage class are assigned to different data types.

2145 (E) Const/volatile specified for imcomplete type
An incomplete type is specified as a const or volatile type.

2200 (E) Index not integer
An array index expression type is not an integer.

2201 (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.

2202 (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.

2203 (E) Illegal member reference for "."
The expression to the left-hand side of the (.) operator is not a struct or union.

2204 (E) Illegal member reference for "->"
The expression to the left of the -> operator is not a pointer to a struct or union.

2205 (E) Undefined member name
An undeclared member name is used to reference a struct or union.

2206 (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).

2207 (E) Scalar required for "!"
The unary operator ! is used on an expression that is not scalar.

2208 (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.

2209 (E) Arithmetic type required for "operator"
The unary operator + or - is used on a non-arithmetic expression.

2210 (E) Integer required for "~"
The unary operator ~ is used on a non-integral expression.

2211 (E) Illegal sizeof
A sizeof operator is used for a bit field member, function, void, or array with an undefined size.

2212 (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.

2213 (E) Arithmetic type required for "operator"
The binary operator *, /, *=, or /= is used in an expression that is not an arithmetic expression.

2214 (E) Integer required for "operator"
The binary operator <<, >>, &, |, ^, %, <<=, >>=, &=, |=, ^=, or %= is used in an expression that is not an integer expression.

2215 (E) Illegal type for "+"
The combination of operand types used with the binary operator + is not allowed.

2216 (E) Illegal type for parameter
Type void is specified for a function call parameter type.

2217 (E) Illegal type for "-"
The combination of operand types used with the binary operator - is not allowed.

2218 (E) Scalar required
The first operand of the conditional operator ?: is not a scalar.

2219 (E) Type not compatible with "?:"
The types of the second and third operands of the conditional operator ?: do not match with each other.

2220 (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 | =.

2221 (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.

2222 (E) Type not compatible for "="
The operand types for the assignment operator = do not match.

2223 (E) Incomplete tag used in expression
An incomplete tag name is used for a struct or union in an expression.

2224 (E) Illegal type for assign
The operand types of the assignment operator += or -= are illegal.

2225 (E) Undeclared name “name”
An undeclared name is used in an expression.

2226 (E) Scalar required for "operator"
The binary operator && or || is used in a non-scalar expression.

2227 (E) Illegal type for equality
The combination of operand types for the equality operator == or != is not allowed.

2228 (E) Illegal type for comparison
The combination of operand types for the relational operator >, <, >=, or <= is not allowed.

2230 (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.

2231 (E) Address of bit field
The unary operator & is used on a bit field.

2232 (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.

2233 (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.

2234 (E) Illegal typedef name reference
A typedef name is used as a variable in an expression.

2235 (E) Illegal cast
An attempt is made to cast a pointer with a floating-point type.

2236 (E) Illegal cast in constant
In a constant expression, an attempt is made to cast a pointer with a char or short type.

2237 (E) Illegal constant expression
In a constant expression, a pointer constant is cast with an integer and the result is manipulated.

2238 (E) Lvalue or function type required for "&"
The unary operator & is not used on the lvalue or is used in an expression other than function type.

2300 (E) Case not in switch
A case label is specified outside a switch statement.

2301 (E) Default not in switch
A default label is specified outside a switch statement.

2302 (E) Multiple labels
A label name is defined more than once in a function.

2303 (E) Illegal continue
A continue statement is specified outside a while, for, or do statement.

2304 (E) Illegal break
A break statement is specified outside a while, for, do, or switch statement.

2305 (E) Void function returns value
A return statement specifies a return value for a function with a void return type.

2306 (E) Case label not constant
A case label expression is not an integer constant expression.

2307 (E) Multiple case labels
Two or more case labels with the same value are used for one switch statement.

2308 (E) Multiple default labels
Two or more default labels are specified for one switch statement.

2309 (E) No label for goto
There is no label corresponding to the destination specified by a goto statement.

2310 (E) Scalar required
The control expression (that determines statement execution) for a while, for, or do statement is not a scalar.

2311 (E) Integer required
The control expression (that determines statement execution) for a switch statement is not an integer.

2312 (E) Missing (
The control expression (that determines statement execution) does not follow a left parenthesis (( ) for an if, while, for, do, or switch statement.

2313 (E) Missing ;
A do statement is ended without a semicolon (;).

2314 (E) Scalar required
A control expression (that determines statement execution) for an if statement is not a scalar.

2316 (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.

2400 (E) Illegal character "character"
An illegal character is detected.

2401 (E) Incomplete character constant
An end of line indicator is detected in the middle of a character constant.

2402 (E) Incomplete string
An end of line indicator is detected in the middle of a string literal.

2403 (E) EOF in comment
An end of file indicator is detected in the middle of a comment.

2404 (E) Illegal character code "character code"
An illegal character code is detected.

2405 (E) Null character constant
There are no characters in a character constant (i.e., no characters are specified between two quotation marks).

2406 (E) Out of float
The number of significant digits in a floating-point constant exceeds 17.

2407 (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.

2408 (E) Comment nest too deep
The nesting level of the comment exceeds the limit of 255 level.

2500 (E) Illegal token “phrase”
An illegal token sequence is used.

2501 (E) Division by zero
An integer is divided by zero in a constant expression.

2600 (E) String literal(s)
An error message specified by string literal #error is output to the list file if nolist option is not specified.

2650 (E) Invalid pointer reference
The specified address does not match the boundary alignment value.

2700 (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.

2701 (E) Multiple interrupt for one function
An interrupt function declaration #pragma interrupt has been declared more than once for the same function.

2702 (E) Multiple #pragma interrupt options
The same type of interrupt is declared more than once.

2703 (E) Illegal #pragma interrupt declaration
An interrupt function declaration #pragma interrupt is specified incorrectly.

2704 (E) Illegal reference to interrupt function
The interrupt function is referenced incorrectly.

2705 (E) Illegal parameter in interrupt function
Argument types to be used for an interrupt function do not match.

2706 (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.

2707 (E) Parameter out of range in interrupt function
The parameter value tn of an interrupt function exceeds the limit of 256.

2709 (E) Illegal section name declaration
The #pragma section specification is illegal.

2710 (E) Section name too long
The specified section name exceeds the limit of 31 characters.

2711 (E) Section name table overflow
The number of section specified in one file exceeds the limit of 64.

2712 (E) GBR based displacement overflow
The variable declared in #pragma gbr_base overflows.

2713 (E) Illegal #pragma interrupt function type
The function type specified #pragma interrupt in illegal.

2800 (E) Illegal parameter number in in-line function
Parameters to be used for an intrinsic function do not match.

2801 (E) Illegal parameter type in in-line function
There are different parameter types in an intrinsic function.

2802 (E) Parameter out of range in in-line function
A parameter exceeds the range that can be specified by an intrinsic function.

2803 (E) Invalid offset value in in-line function
An argument for an intrinsic function is specified incorrectly.

2804 (E) Illegal in-line function
An intrinsic function that cannot be used by the specified **cpu** option exists.

2805 (E) Function "function name" in #pragma inline/inline_asm already declared
The function indicated by a function name exists before the #pragma specification.

2806 (E) Multiple #pragma for one function
Two or more #pragma directives are specified for one function incorrectly.

2807 (E) Illegal #pragma inline/inline_asm declaration
The #pragma inline or #pragma inline_asm is specified illegally.

2808 (E) Illegal option for #pragma inline_asm
The -code=machinecode option is specified in addition to the #pragma inline_asm specification declaration.

2809 (E) Illegal option for #pragma inline/inline_asm function type
An identifier type that specifies #pragma inline or #pragma inline_asm is illegal.

2810 (E) Global variable “variable name” in #pragma gbr_base/gbr_basel already declared
A variable definition indicated by variable name exists before #pragma specification.

2811 (E) Multiple #pragma for one global variable
Two or more #pragma directives are specified for one variable incorrectly.

2812 (E) Illegal #pragma gbr_base/gbr_basel declaration
The #pragma gbr_base or #pragma gbr_base1 specification is illegal declaration.

2813 (E) Illegal #pragma gbr_base/gbr_basel global variable type
An identifier type that specifies #pragma gbr_bsee or #pragma gbr_base1 is illegal.

2814 (E) Function “function name” in #pragma noregsave/norealloc/regsave already declared
The function indicated by a function name exists before the #pragma specification declaration.

2815 (E) Illegal #pragma noregsave/noregalloc/regsave declaration
The #pragma noregsave, or #pragma noregalloc, or #pragma regsave specification is illegal.

2816 (E) Illegal #pragma noregsave/noregalloc/regsave function type
An identifier type that specifies #pragma noregsave, #pragma noregalloc, or #pragma regsave is illegal.

2817 (E) Symbol "identifier" in #pragma abs16 already declared
A name indicated by an identifier exists before the #pragma specification declaration.

2818 (E) Multiple #pragma for one symbol
More than one #pragma is incorrectly specified for one identifier.

2819 (E) Illegal #pragma abs16 declaration
The #pragma abs16 specification is illegal declaration.

2820 (E) Illegal #pragma abs16 symbol type
An identifier type that specifies #pragma abs16 is illegal.

2821 (E) Global variable “variable name” in #pragma global_register already declared
The variable that specifies #pragma global_register has already been specified.

2822 (E) Illegal register “register” in #pragma global_register
The register that specified #pragma global_register is illegal.

2823 (E) Illegal #pragma global_register declaration
The specification method of #pragma global_register is illegal.

2824 (E) Illegal #pragma global_register type
A variable that cannot specify #pragma global_register exists.

3000 (F) Statement nest too deep
The nesting level of an if, while, for, do, and switch statements exceeds the limit. The maximum number is 32 levels.

3001 (F) Block nest too deep
The nesting level of compound statements exceeds the limit. The maximum number is 32 levels.

3002 (F) #if nest too deep
The conditional compilation (#if, #ifdef, #ifndef, #elif, and #else) nesting level exceeds the limit. The maximum number is 32 levels.

3006 (F) Too many parameters
The number of parameters in either a function declaration or a function call exceeds the limit. The maximum number is 63.

3007 (F) Too many macro parameters
The number of parameters in a macro definition or a macro call exceeds the limit. The maximum number is 63.

3008 (F) Line too long
After a macro expansion, the length of a line exceeds the limit. The maximum number is 4096 characters.

3009 (F) String literal too long
The length of string literal exceeds 512 characters. The length of string literal equals to the number of bytes when linking string literals specified continuously. The length of the string literal is not the length in the source program but the number of bytes included in the string literal data. Escape sequence is counted as one character.

3010 (F) Processor directive #include nest too deep
The nesting level of the #include directive exceeds the limit. The maximum level is 30.

3011 (F) Macro expansion nest too deep
The nesting level of macro expansion performed by a #define directive exceeds the limit. The maximum level is 32.

3012 (F) Too many function definitions
The number of function definitions exceeds the limit. The maximum number is 512.

3013 (F) Too many switches
The number of switch statements exceeds the limit. The maximum number is 256.

3014 (F) For nest too deep
The nesting level of a for statement exceeds the limit. The maximum level is 16.

3015 (F) Symbol table overflow
The number of symbols to be generated by the SH C compiler exceeds the limit. The maximum number is 24576.

3016 (F) Internal label overflow
The number of internal labels to be generated by the SH C compiler exceeds the limit. The maximum number is 32767.

3017 (F) Too many case labels
The number of case labels in one switch statement exceeds the limit. The maximum number is 511.

3018 (F) Too many goto labels
The number of goto labels defined in one function exceeds the limit. The maximum number is 511.

3019 (F) Cannot open source file "file name"
A source file cannot be opened.

3020 (F) Source file input error "file name"
A source or include file cannot be read.

3021 (F) Memory overflow
The SH C compiler cannot allocate sufficient memory to compile the program.

3022 (F) Switch nest too deep
The nesting level of a switch statement exceeds the limit. The maximum level is 16.

3023 (F) Type nest too deep
The number of types (pointer, array, and function) that qualify the basic type exceeds 16.

3024 (F) Array dimension too deep
An array has more than six dimensions.

3025 (F) Source file not found
A source file name is not specified in the command line.

3026 (F) Expression too complex
An expression is too complex.

3027 (F) Source file too complex
The nesting level of statements in the program is too deep or an expression is too complex.

3028 (F) Source line number overflow
The last source line number exceeds the limit. The maximum number is 65535.

3030 (F) Too many compound statements
The number of compound-statements exceeds the limit.

3031 (F) Data size overflow
The size of an array or a structure exceeds the limit of 2147483647 bytes.

3033 (F) Symbol table overflow
The number of symbols used for debugging information exceeds 32767.

3100 (F) Misaligned pointer access
There has been an attempt to refer or specify using a pointer that has an invalid alignment.

3201 (F) Object size overflow
The object file size exceeds the limit of 4 Gbytes.

3202 (F) Too many source lines for debug
There are too many source line to output debugging information.

3203 (F) Assembly source line too long
The assembly source line is too long to output.

3204 (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.

3300 (F) Cannot open internal file
An error occurred due to one of the following causes:
(1) An intermediate file internally generated by the SH C 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 SH C compiler uses internally cannot be opened.

3301 (F) Cannot close internal file
An intermediate file internally generated by the SH C compiler cannot be closed. Make sure the SH C compiler is installed correctly.

3302 (F) Cannot input internal file
An intermediate file internally generated by the SH C compiler cannot be read. Make sure the SH C compiler is installed correctly.

3303 (F) Cannot output internal file
An intermediate file internally generated by the SH C compiler cannot be written.

3304 (F) Cannot delete internal file
An intermediate file internally generated by the SH C compiler cannot be deleted.

3305 (F) Invalid command parameter "option name"
An invalid compiler option is specified.

3306 (F) Interrupt in compilation
An interrupt generated by a (CNTL) C command (from a standard input terminal) is detected during compilation.

3307 (F) Compiler version mismatch
File versions specified in the SH C compiler do not match the other file versions.

3320 (F) Command parameter buffer overflow
The command line specification exceeds 256 characters.

3321 (F) Illegal environment variable
An error occurred due to one of the following causes:

1. SHC_LIB was not specified.
2. 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.
3. Other than SH1, SH2, SHDSP, SH3, or SH3E is set for the environment variable SHCPU.

4000 – 4999 (—) Internal error
An internal error occurs during compilation. Report the error occurrence to your local Hitachi dealer.

Section 2 C Standard Library Error Messages

For some library functions, if an error is generated during the library function execution, an error number is set in the macro **errno** defined in the header file <errno.h> contained in the standard library. Error messages are defined in the error numbers so that error messages can be output. The following shows an example of an error message output program.

Example:

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

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 number corresponding to **errno** is set.
2. The **strerror** function returns a pointer of the string literal of the corresponding error message when the error number is passed as an actual parameter. An error message is output by specifying the output of the string literal of the printf function.

Table 4.1 List of Standard Library Error Messages

Error No.	Error Message/Explanation	Functions to Set Error Numbers
1100 (ERANGE)	`Data out of range` An overflow occured.	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 4.1 List of Standard Library Error Messages (cont)

Error No.	Error Message/Explanation	Functions to Set Error Numbers
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, fscanf
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

APPENDIX

Appendix A Language and Standard Library Function Specifications of the C Compiler

A.1 Language Specifications of the C Compiler

A.1.1 Compilation Specifications

Table A.1 Compilation Specifications

Item	C Compiler Specification
Error information when an error is detected	Refer to part IV, Error Messages

A.1.2 Environmental Specifications

Table A.2 Environmental Specifications

Item	C Compiler Specification
Actual argument for the main function	Not specified
Interactive I/O device configuration	Not specified

A.1.3 Identifiers

Table A.3 Identifier Specifications

Item	C Compiler Specification
Number of valid characters of internal identifiers not used for external linkage	The first 250 characters are valid for an internal or external identifier.
Number of valid characters of external identifiers used for external linkage	
Lowercase and uppercase character distinction in external identifiers used for external linkage	Lowercase characters are distinguished from uppercase characters.

Note: Two different identifiers with the same first 250 characters are considered to be identical even if the 251st or later characters are different.

Example:

1. longabcde ... ab; (the 250th character is a and the 251st character is b)
2. longabcde ... ac; (the 250th character is a and the 251st character is c)

Identifiers1. and 2. are indistinguishable because the first 250 characters are the same.

A.1.4 Characters

Table A.4 Character Specifications

Item	C Compiler Specification
Elements of source character set and execution environment character set	ASCII character set
	Kanji used in host environment can be used for source program comment.
Shift state used for encoding multiple-byte characters	Shift state is not supported
The number of bits used to indicate a character set during program execution	Eight bits are used for each character.
Correspondence between source character set used in character constant or character string and execution environment character set	ASCII is used for both.
Value of character constant including characters and escape sequence that are not specified in the C language	Characters and escape sequence other than that specified by the C language are not supported.
Character constant of two or more characters or wide character constant including multiple-byte characters of two or more characters	The upper one character of the character constant is valid. Wide character constant is not valid. If a character constant of more than one character is specified, a warning error message is output.
locale specifications used to convert multiple-byte character to wide character	**locale** is not supported
Simple **char** having normal the value range same as **signed char** or **unsigned char**.	The same range as the **signed char.**

A.1.5 Integer

Table A.5 Integer Specifications

Item	C Compiler Specification
Integer-type data representation and value	Table A.6 shows data representation and value. (A negative value is shown in two's complement.)
Effect when an integer is too large to be converted into a **signed** integer-type value or into a value which cannot be expressed using **signed char** type (when the resulting value cannot be represented with the resulting converted type)	The lower one or two bytes of the integer is used as the conversion result.
The result of bitwise operations on **signed** integers	**signed** value
Sign of the remainder for integer division	Same as the sign of the dividend.
Effect of a right shift operation on the sign bit of **signed** integer-type data	The sign bit is unchanged by the shift operation.

Table A.6 Integer Types and Their Corresponding Data Range

Type	Range of Values	Data Size
char (signed char)	–128 to 127	1 byte
unsigned char	0 to 255	1 byte
short	–32768 to 32767	2 bytes
unsigned short	0 to 65535	2 bytes
int	–2147483648 to 2147483647	4 bytes
unsigned int	0 to 4294967295	4 bytes
long	–2147483648 to 2147483647	4 bytes
unsigned long	0 to 4294967295	4 bytes

Note: Type specification in parenthesis () can be omitted. The order of type specification is arbitrary.

A.1.6 Floating-Point Numbers

Table A.7 Floating-Point Number Specifications

Item	C Compiler Specification
Data that can be represented as floating-point type and value	The **float**, **double**, and **long double** are provided as floating-point types. See section A.3, Floating-Point Number Specifications, for details on floating-point numbers (internal representation, conversion specifications, and operation specifications). Table A.8 shows the limits on representing floating-point numbers.
Rounding down direction when converting an integer number to a floating-point number that cannot represent the integer's original value correctly	
Rounding down or rounding method when converting a floating-point number to a lower-precision number	

Table A.8 Limits on Floating-Point Numbers

	Limit	
Item	**Decimal**[*1]	**Hexadecimal**
Maximum value of **float** type	3.4028235677973364e+38f (3.4028234663852886e+38f)	`7f7fffff`
Positive minimum value of **float** type	7.0064923216240862e–46f (1.4012984643248171e–45f)	`00000001`
Maximum value of **double**[*2] or **long double** type	1.7976931348623158e+308 (1.7976931348623157e+308)	`7fefffffffffffff`
Positive minimum value of **double**[*2] or **long double** type	4.9406564584124655e–324 (4.9406564584124654e–324)	`0000000000000001`

Notes: 1. Limits on decimal is non-zero minimum value or maximum value not infinitive value. Values within **( ) indicate theoretical values.**

2. **double** type will have the same value as **float** type when **–double=float** option is specified.

A.1.7 Arrays and Pointers

Table A.9 Array and Pointer Specifications

Item	C Compiler Specification
Integer type required for holding array's maximum size (size_t)	**unsigned long**
Conversion from pointer-type data to integer-type data (Pointer-type data size ≥ Integer-type data size)	The lower byte of pointer-type data is used.
Conversion from pointer-type data to integer-type data (Pointer-type data size < Integer-type data size)	Extended with signs
Conversion from integer-type data to pointer-type data (Integer-type data size ≥ Pointer-type data size)	The lower byte of integer-type data is used.
Conversion from integer-type data to pointer-type data (Integer-type data size < Pointer-type data size)	Extended with signs
Integer type required for holding pointer difference between members in the same array **(ptrdiff_t)**	**int**

A.1.8 Register

Table A.10 Register Specifications

Item	C Compiler Specification
The maximum number of register variables that can be allocated to registers	7
Type of register variables that can be allocated to registers	**char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float,** and **pointers**

A.1.9 Structure, Union, Enumeration, and Bit Field Types

Table A.11 Specifications for Structure, Union, Enumeration, and Bit Field Types

Item	C Compiler Specification
Effect of referencing a union-type member using another member whose data type is different	Reference is possible but the referred value is not guaranteed.
Structure member alignment	The maximum data size among structure members is a boundary alignment number. Refer to table A.6 Integer Types and Their Corresponding Data Range. *1
Sign of an **int** bit field	Assumed to be **signed int**
Allocation order of bit fields in **int** area	Beginning from the high order bit to low order bit. *2
Result when a bit field has been allocated in an **int** area and the next bit field to be allocated is larger than the remaining **int**	The next bit field is allocated to the next **int** area.*2
Type specifier allowed for bit field	**char, unsigned char, short, unsigned short, int, unsigned int, long,** and **unsigned long**
Integer describing enumeration	**int**

Notes: 1. See section 2.2.2 Combined-Type Data, in part II C Programming, for details on structure member allocation.
2. See section 2.2.3, Bit Fields, in part II C Programming, for details on bit field allocation.

A.1.10 Qualifier

Table A.12 Qualifier Specifications

Item	C Compiler Specification
volatile data access type	Not specified

A.1.11 Declarations

Table A.13 Declaration Specifications

Item	C Compiler Specification
Types that can qualify the basic types (pointer, array, and function)	Up to 16 types can be specified.

1. Example of counting the number of types that qualify the basic types

Examples:

- int a;

  a is **int** (basic type) and the number of declarators that qualify the basic type is zero.

- char *f( );

  f is a function type that returns pointer to **char** (basic type). The number of declarators that qualify the basic type is two.

A.1.12 Statement

Table A.14 Statement Specifications

Item	C Compiler Specification
The number of case label that can be declared in a switch statement	Up to 511 labels can be specified.

A.1.13 Preprocessor

Table A.15 Preprocessor Specifications

Item	C Compiler Specification
Correspondence between single character constant in a constant expression and execution environment character set in the conditional compilation	Character strings in the preprocessor statement match the execution environment character set
Reading an include file	The file within < > is read from a directory specified by the **include** option. When more than one directory is specified, a file is searched for in the specified order. If a specified file is not found at a specified directory, the search continues at a directory specified by environment variable SHC_INC and a system directory (SHC_LIB) in this order.
Supporting an include file whose name is enclosed in a pair of double quotation marks	The C compiler supports include files whose names are delimited by double quotation marks. The C compiler reads these include files from the current directory. If the include files are not in the current directory, the C compiler reads them from the directory specified in advance.
Blank character in the character string of an actual parameter for a **#define** statement after expansion	Strings of blanks are expanded as one blank character.
#pragma directive operation	#pragma interrupt, #pragma section, #pragma inline, #pragma inline_asm, #pragma abs16, #pragma gbr_base, #pragma gbr_base1, #pragma noregsave, #pragma noregalloc, #pragma regsave, and #pragma global_register are supported.*[1]
Value of _ _DATE_ _, _ _TIME_ _	Data depending on the host machine timer when the compilation starts.

Note: See section 3, Extended Specification, in part II C Programming, for details on #pragma specifications.

A.2 C Library Function Specifications

This section explains the specifications for C library functions that are not declared in C language specification.

A.2.1 stddef.h

Table A.16 stddef.h Specifications

Item	C Compiler Specification
Value of macro **NULL**	The value is 0 for a pointer type to a void type
Contents of **ptrdiff_t**	**int** type

A.2.2 assert.h

Table A.17 assert.h Specifications

Item	C Compiler Specification
Information output and terminal operation of **assert** function	See 1. for the format of output information. The program outputs information and then calls the **abort** function to stop the operation.

1. The following message is output when the value of the expression is 0 for assert (expression):
   `Assertion Failed:` Δ<expression>Δ`File`Δ<file name>, `Line`Δ<line number>

A.2.3 ctype.h

Table A.18 ctype.h Specifications

Item	C Compiler Specification
The character set for which the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions check	Character set that can be expressed in **unsigned char** type. Table A.19 shows the character set that results in a true return value.

Table A.19 Set of Characters that Returns True

Function Name	Characters That Become True
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'

A.2.4 math.h

Table A.20 math.h Specifications

Item	C Compiler Specification
Value returned by a mathematical function if an input parameter is out of the range	For details on format for not a number, refer to A.3, Floating-Point Number Specifications Returns a not a number.
Is errno set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function?	No, it is not.
Does a range error occur if the 2nd actual parameter in the **fmod** function is 0	Returns a not a number and a range error occurs.

Note: **math.h** defines macro names **ENUM** and **ERANGE** that indicates a standard library error number.

A.2.5 setjmp.h

Table A.21 setjmp.h Specifications

Item	C Compiler Specification
What programs can a **setjmp** function be called in ?	The following statements can call a **setjmp** function if specified with **setjmp()** or **ver=setjmp()** format: 1. A single statement or an **if**, **while**, **do**, or **for** statement that specifies condition 2. **switch** or **return** statement

A.2.6 stdio.h

Table A.22 stdio.h Specifications

Item	C Compiler Specification
Is a carriage return character indicating the last line of input data required?	Not specified. Depends on the low-level interface routine specifications.
Is a blank character immediately before the carriage return character read?	
Number of NULL characters added to data written to binary file	
Initial value of file position specifier in addition mode	
Is a file data lost following text file output?	
File buffering specifications	
Does a file with file length 0 exist?	
File name configuration rule	
Can the same files be opened simultaneously?	
Output data representation of the %p format conversion in the **fprintf** function	Hexadecimal representation
Output data representation of the %p format conversion in the **fscanf** function, the meaning of (–) in the **fscanf** function	Hexadecimal representation If (–) is not placed at the beginning or end of a **fscanf** character string or does not follow (^) in a **fscanf** character string, indicates the range between the previous and following characters.
Value of **errno** specified by **fgetpos** and **ftell** functions	The **fgetpos** function is not supported. The **ftell** function does not specify the errno value. The errno value is determined depending on the low-level interface routine.
Output format of messages generated by the **perror** function	See 1. below for the output message format.
calloc, **malloc**, or **realloc** function operation when the size is 0	0 byte area is allocated.

1. Messages generated by a **perror** function follow this format:
   <character string> : <error message corresponding to the error number indicated by errno>
2. Table A.23 shows the format used to indicate infinity and not a number for floating-point numbers when using the **printf** or **fprintf** function.

Table A.23 Infinity and Not a number

Value	C Compiler Specification
Positive infinity	++++++
Negative infinity	−−−−−−
Not a number	******

A.2.7 string.h

Table A.24 string.h Specifications

Item	C Compiler Specification
Return value from an **memcmp**, **strcmp**, or **strncmp** function.	Value is treated as a signed value.
Error message returned by the **strerror** function	Refer to section 2, C Standard Library, in part IV Error Messages.

A.2.8 errno.h

Table A.25 errno.h Specifications

Item	C Compiler Specification
errno	An error number is specified when an error occurs in an **int** type variable or a library function.
ERANGE	Refer to section 2, C Standard Library Error Messages, in part IV Error Messages.
EDOM	
EDIV	
ESTRN	
PTRERR	
ECBASE	
ETLN	
EEXP	
EEXPN	
EFLOATO	
EFLOATU	
EDBLO	
EDBLU	
ELDBLO	
ELDBLU	
NOTOPN	
EBADF	
ECSPEC	

A.2.9 Libraries that are Not Supported by the SH C Compiler

Table A.26 shows a list of libraries that are not supported by the SH C compiler but are defined in the H series C language manual specification. Header files are not supported for **signal.h** and **time.h**.

Table A.26 Libraries that are Not Supported by the SH C Compiler

Header File	Library
signal.h	signal, raise
stdio.h	remove, rename, tmpfile, tmpnam
stdlib.h	getenv, system
time.h	clock, difftime, time, asctime, ctime, gmtime, localtime

A.3 Floating-Point Number Specifications

A.3.1 Internal Representation of Floating-Point Numbers

The internal representation of floating-point numbers follows the IEEE standard format. This section explains the outline of the internal representation of IEEE-type floating-point numbers.

Internal Representation Format: float is represented in IEEE single precision (32 bits), **double** and **long double** are represented in IEEE double precision (64 bits).

Internal Representation Structure: Figure A.1 shows the structure of **float**, **double**, and **long double** in internal representation.

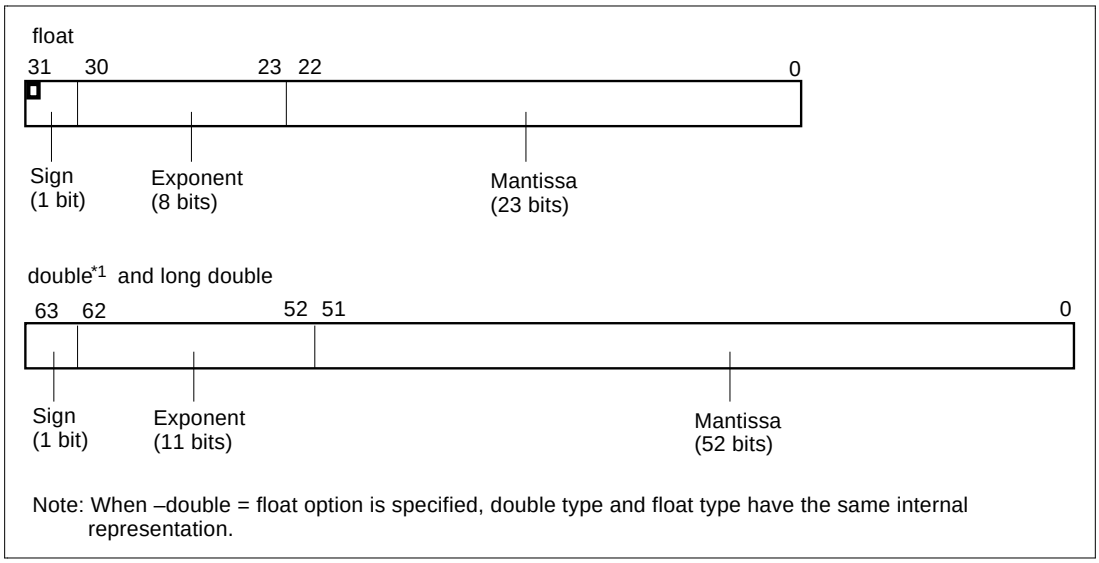


Figure A.1 Structure for the Internal Representation of Floating-Point Numbers

The elements of the structure have the following meanings.

1. Sign

   This indicates the sign of a floating-point number. Positive and negative are represented by 0 and 1, respectively.
2. Exponent

   This indicates the exponent of a floating-point number as a power of two.
3. Mantissa

   This determines the significant digits of a floating-point number.

Types of Values: Floating-point numbers can represent infinity in addition to real numbers. The rest of this section explains the types of values that can be represented by floating-point numbers.

1. Normalized Number

   The exponent is not 0 or the maximum. A normalized number represents a real number.
2. Denormalized Number

   The exponent is 0 and the mantissa is not 0. A denormalized number is a real number whose absolute value is very small.
3. Zero

   The exponent and mantissa are both 0. Zero represents the value 0.0.
4. Infinity

   The exponent is the maximum and mantissa is 0.
5. Not a Number

   The exponent is the maximum and the mantissa is not 0. This is used to represent an operation result that is undefined (such as 0.0/0.0, ∞/∞, ∞ – ∞).

Note: A denormalized number represents a floating-point number whose absolute value is so small that it cannot be represented as a normalized number. Denormalized numbers have less significant digits than normalized numbers. The significant digits of a result are not guaranteed if either the operation result or an intermediate result is a denormalized number.

Table A.27 Types of Values Represented by Floating-Point Numbers

	Exponent		
Mantissa	**0**	**Other than 0 or Maximum**	**Maximum**
0	0	Normalized number	Infinity
Other than 0	Denormalized number		Not a number

A.3.2 float

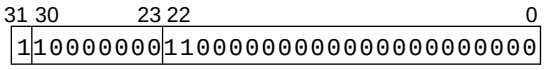
float is internally represented as 1 sign bit, 8 exponent bits, and 23 mantissa bits.

Normalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is a number from 1 to 254 ($2^8 - 2$). From the value 1 to 254, 127 is subtracted and the result is used as the actual exponent. The range of actual exponents is –126 to 127. The mantissa is a value from 0 to $2^{23} - 1$. The actual mantissa is assumed that the highest order bit (2^{23}) is 1 and a decimal point follows it.

Value represented by a normalized number is shown in the following expression:

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

Example:



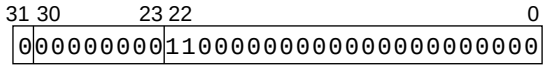
Sign: –
Exponent: $10000000_{(2)} - 127 = 1$ ($_{(2)}$ indicates binary data throughout this manual.)
Mantissa: $1.11_{(2)} = 1.75$
Value: $-1.75 \times 2^1 = -3.5$

Denormalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is 0 which makes the actual exponent equal to –126. The mantissa is a value from 1 to $2^{23} - 1$. The actual mantissa is assumed that a highest order bit (2^{23}) is 0 and a decimal point follows it.

Value represented by a denormalized number is shown in the following expression:

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

Example:



Sign: +
Exponent: $0_{(2)} - 126 = -126$
Mantissa: $0.11_{(2)} = 0.75$
Value: 0.75×2^{-126}

Zero: The sign bit is either 0 (positive) or 1 (negative) and indicates +0.0 and –0.0, respectively. The exponent and mantissa are 0. Both +0.0 and –0.0 represent 0.0. See appendix A.3.4, Floating-Point Operation Specifications, for differences in each operation depending on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) and indicates +∞ and –∞, respectively. The exponent is 255 ($2^8 – 1$). The mantissa is 0.

Not a Number: The exponent is 255 ($2^8 – 1$) and the mantissa is not equal to 0.

Note: When the CPU is SH3E, the not a number for the most significant bit of the mantissa which is 0 is called qNaN, and the not a number for the most significant bit of the mantissa which is 1 is called sNaN. Other mantissa field values and sign parts are not specified.

A.3.3 double and long double

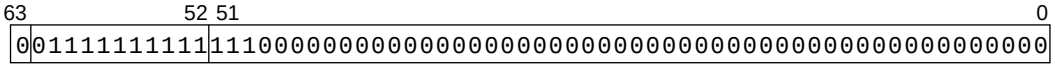
A **double** or **long double** is represented as 1 sign bit, 11 exponent bits, and 52 mantissa bits.

Normalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is a number from 1 to 2046 ($2^{11} – 2$). From the value 1 to 2046, 1023 is subtracted and the result is used as the actual exponent. The range of actual exponents is –1022 to 1023. The mantissa is a value from 0 to $2^{52} – 1$. The actual mantissa is assumed that the highest order bit (2^{52}) is 1 and a decimal point follows it.

Value represented by a normalized number is shown in the following expression:

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

Example:



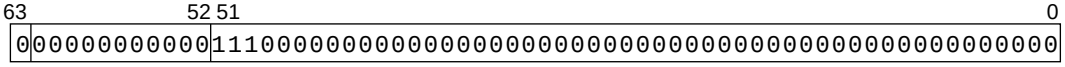
Sign: +
Exponent: $1111111111_{(2)} – 1023 = 1$
Mantissa: $1.111_{(2)} = 1.875$
Value: $1.875 \times 2^0 = 1.875$

Denormalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is 0 which makes the actual exponent equal to –1022. The mantissa value is from 1 to $2^{52} - 1$. The actual mantissa is assumed that the highest order bit (2^{52}) is 0 and a decimal point follows it.

Value represented by a denormalized number is shown in the following expression:

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

Example:



Sign: –
Exponent: $0_{(2)} - 1022 = -1022$
Mantissa: $0.111_{(2)} = 0.875$
Value: $0.875 \times 2^{-1022} = 1.875$

Zero: The sign bit is either 0 (positive) or 1 (negative) and indicates +0.0 and –0.0, respectively. The exponent and mantissa are 0. Both +0.0 and –0.0 represent 0.0. See appendix A.3.4, Floating-Point Operation Specifications, for differences in each operation depending on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) and indicate +∞ and –∞ respectively. The exponent is 2047 ($2^{11} - 1$). The mantissa is 0.

Not a Number: The exponent is 2047 ($2^{11} - 1$) and the mantissa is not equal to 0.

Note: When the CPU is SH3E, the not a number for the most significant bit of the mantissa which is 0 is called qNaN, and the not a number for the most significant bit of the mantissa which is 1 is called sNaN. Other mantissa field values and sign parts are not specified.

A.3.4 Floating-point Operation Specifications

This section explains the floating-point arithmetic used in C language functions. It also gives the specifications for converting between the decimal representation and the internal representation of floating-point numbers generated during C compiler or standard library function processing.

Arithmetic Operation Specifications:

1. Result Rounding

   If the precise result of a floating-point operation exceeds the significant digits of the internally represented mantissa, the result is rounded as follows:

   a. The result is rounded to the nearest internally representable floating-point number.
   b. If the result is directly between the two nearest internally representable floating-point numbers, the result is rounded so that the lowest bit of the mantissa becomes 0.
   c. When the CPU is SH3E, the number of digits that exceed the significant digit are rounded down.

2. Overflow/Underflow and Invalid Operation Handling

   Invalid operations, overflows and underflows resulting from numeric operations are handled as follows:

   a. For an overflow, positive or negative infinity is used depending on the sign of the result.
   b. For an underflow, positive or negative zero is used depending on the sign of the result.
   c. An invalid operation is assumed when: i. infinity is added to infinity and each infinity has a different sign, ii. infinity is subtracted from infinity and each infinity has the same sign, iii. zero is multiplied by infinity, iv. zero is divided by zero, or v. infinity is divided by infinity. In each case, the result is not a number.
   d. Data accuracy cannot be guaranteed if the data overflows when converting floating-point data to integer data.

Note: Operations are performed with constant expressions at compile time. If an overflow, underflow, or invalid operation is detected during these operations, a warning-level error occurs.

3. Special Value Operations

   More about special value (zero, infinity, and not a number) operations:

   a. If positive zero and negative zero are added, the result is positive zero.
   b. If zero is subtracted from zero and both zeros have the same sign, the result is positive zero.
   c. The operation result is always a not a number if one or both operands are not a numbers.
   d. Positive zero is equal to a negative zero for comparison operations.
   e. If one or both operands are not a numbers in a comparison or equivalence operation, the result of != is always true and all other results are false.

Conversion between Decimal Representation and Internal Representation: This section explains the conversion between floating-point constants in a source program and floating-point constants in internal representation. The conversion between decimal representation and internal representation of ASCII character string floating-point numbers by library functions is also explained.

1. To convert a floating-point number from decimal representation to internal representation, the floating-point number in decimal representation is first converted to a floating-point number in normalized decimal representation. A floating-point number in normalized decimal representation is in the format $\pm M \times 10^{\pm N}$. The following ranges of M and N are used:
   a. For normalized **float**

      $0 \leq M \leq 10^9 - 1$

      $0 \leq N \leq 99$

   b. For normalized **double** and **long double**

      $0 \leq M \leq 10^{17} - 1$

      $0 \leq N \leq 999$

   An overflow or underflow occurs if a floating-point number in decimal representation cannot be normalized. If a floating-point number in normalized decimal representation contains too many significant digits, as a result of the conversion, the lower digits are discarded. In the above cases, a warning-level error occurs at compilation and the variable **errno** is set equal to the corresponding error number at run time.

   To convert a floating-point number from decimal representation to normalized decimal representation, the length of the original ASCII character string must be less than or equal to 511 characters. Otherwise, an error occurs at compile time and the variable **errno** is set equal to the corresponding error number at run time.

   To convert a floating-point number from internal representation to decimal representation, the floating-point number is first converted from internal representation to normalized decimal representation. The result is then converted to an ASCII character string according to a specified format.

2. Conversion between Normalized Decimal Representation and Internal Representation

   If the exponent of a floating-point number to be converted between decimal representation and internal representation is too large or too small, a precise result cannot be obtained. This section explains the range of exponents for precise conversion and the error that results from exceeding the range.

   a. Range of Exponents for Precise Conversion

      Rounding as explained in the description, Result Rounding, in appendix A.3 4, Floating-point Operation Specifications, is performed precisely for floating-point numbers whose exponents are in the following ranges:

      For **float** : $0 \leq M \leq 10^9 - 1,\ 0 \leq N \leq 13$

      For **double** and **long double**: $0 \leq M \leq 10^{17} - 1,\ 0 \leq N \leq 27$

      An overflow or underflow will not occur if the exponent is within the proper ranges.

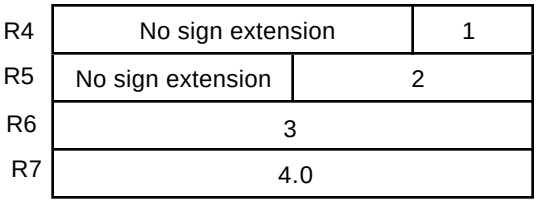
   b. Conversion and Rounding Error

      The difference between, a. the error occurring when the exponent outside the proper range is converted, and b. the error occurring when the value is precisely rounded, does not exceed the result of multiplying the least significant digit by 0.47. If an exponent outside the proper range is converted, an overflow or underflow may occur. In such a case, a warning-level error occurs at compilation and the variable errno is set to the corresponding error number at run time.

Appendix B Parameter Allocation Example

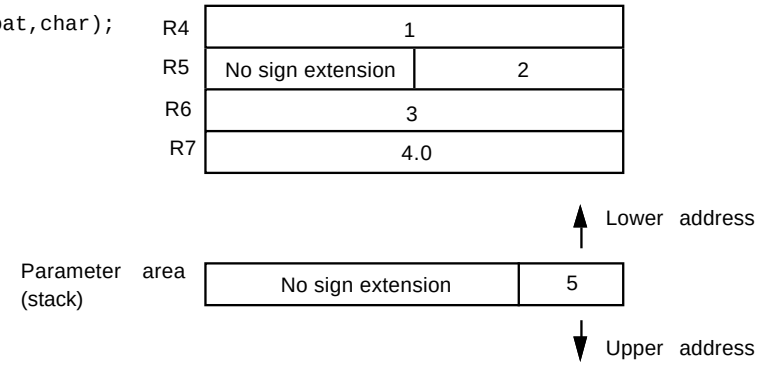
Example 1: Register parameters are allocated to registers R4 to R7 depending on the order of declaration.

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



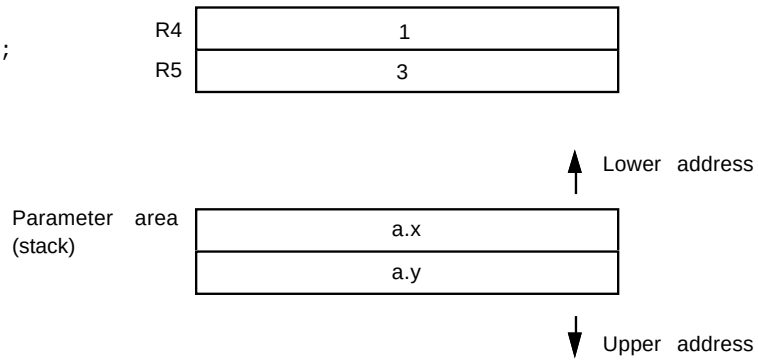
Example 2: Parameters which could not be allocated to registers R4 to R7 are allocated to the stack area as shown below. If a **char** (**unsigned**) or **short** (**unsigned**) type parameter is allocated to a parameter area on a stack, it is extended to a 4-byte area.

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



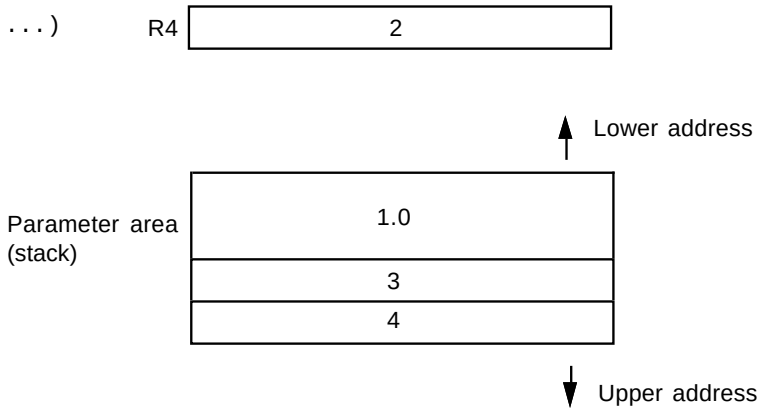
Example 3: Parameters having a type that cannot be allocated to registers from R4 to R7 are allocated to the stack area.

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



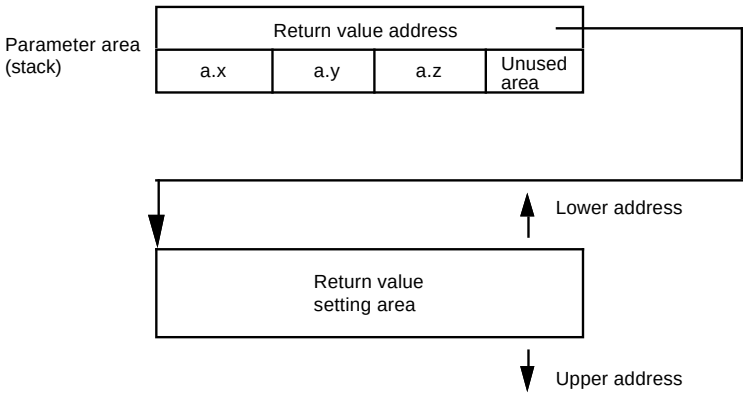
Example 4: If a function whose number of parameters changes is specified by prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding parameters are allocated to a stack.

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



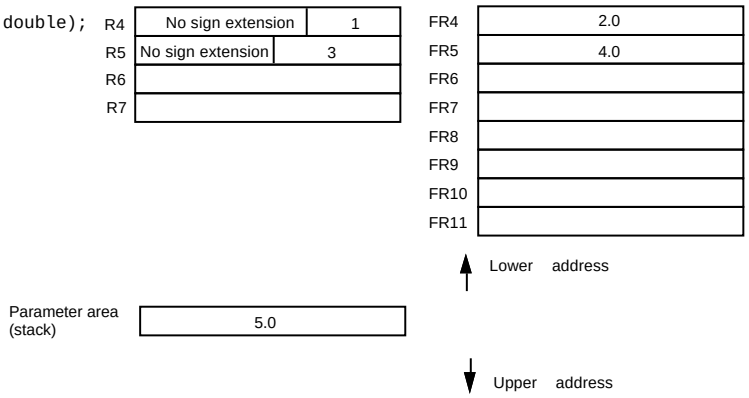
Example 5: If a value returned by a function exceeds four bytes, or is a structure type, a return value is specified just before parameter area. If structure size is not a multiple of four, an unused area is generated.

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



Example 6: When the CPU is SH3E, float type parameters are allocated to FPU registers.

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



Appendix C Usage of Registers and Stack Area

This section describes how to use registers and stack area by the C compiler. The user does not have to take care how to use this area, because registers and stack area used by a function are operated by the C compiler. Figure C.1 shows the usage of registers and stack area.

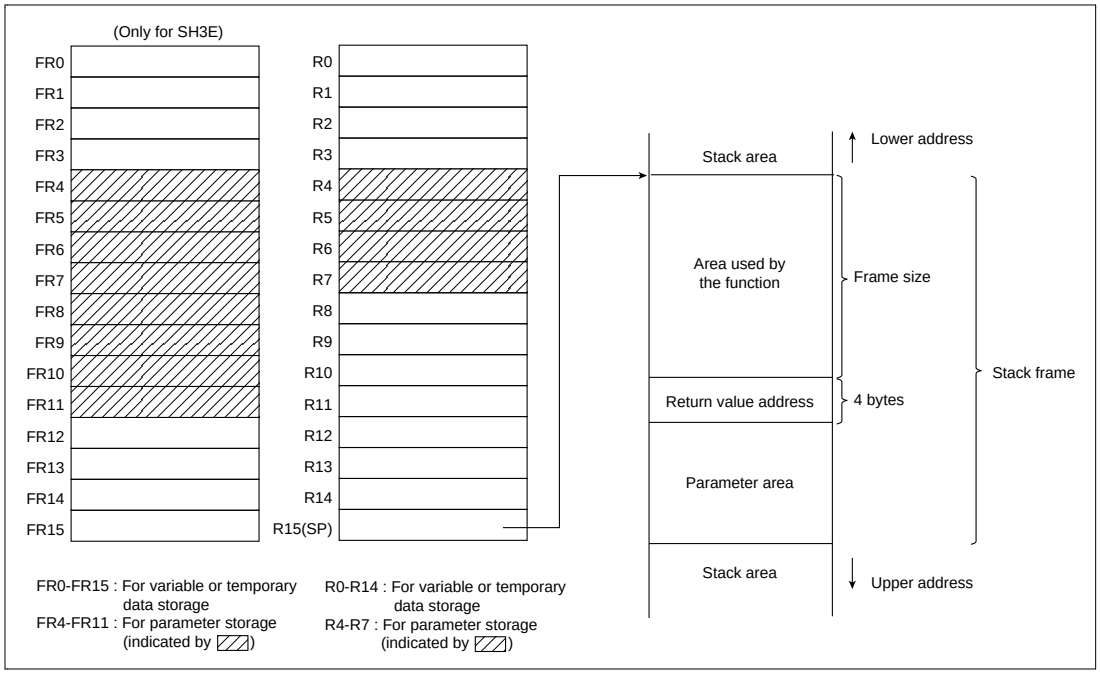


Figure C.1 Usage of Registers and Stack Area

Appendix D Creating Termination Functions

D.1 Creating Library onexit Function

This section describes how to create library onexit function that defines termination routines. The onexit function defines a function address, which is passed as a parameter, in the termination routine table. If the number of defined functions exceeds the limit value (assumed to be 32 in the following example), or if the same function is defined twice or more, NULL is returned. Otherwise, value other than NULL is returned. An example of onexit routine is shown below.

Example:

```
#include <stdlib.h>
typedef void *onexit_t;

int _onexit_count=0;
onexit_t (*_onexit_buf[32])(void);

extern onexit_t onexit(onexit_t (*)(void));

onexit_t onexit(f)
onexit_t (*f)(void);
{
      int i;

      for(i=0; i<_onexit_count ; i++)
         if( _onexit_buf[i]==f)            /* Checks if the same function */
                  return NULL;             /* has been defined            */
         if( _onexit_count==32)            /* Checks if the No. of        */
                                           /* defined functions exceed    */
                                           /* limit                       */
           return NULL;
         else{
           _onexit_buf[ _onexit_count]=f;    /*Defines the function address*/
           _onexit_count++;
           return &_onexit_buf[_onexit_count -1];
         }
}
```

D.2 Creating exit Function

This section describes how to create exit function that terminates program execution. Note that the exit function must be created according to the user system specifications referring to the following example, because how to terminate a program differs depending on the user system.

The exit function terminates C program execution based on the termination code returned as a parameter and then returns to the environment at program initiation. Returning to the environment at program initiation is achieved by the following two steps:

1. Sets a termination code in an external variable
2. Returns to the environment that is saved by the setjmp function immediately before calling the main function

An example of the exit function is shown below.

```
#include <setjmp.h>
#include <stddef.h>

typedef void *onexit_t;
extern int _onexit_count;
extern onexit_t (*_onexit_buf[32])(void);

extern jmp_buf _init_env ;
extern int _exit_code ;

extern void _CLOSEALL();
extern void exit(int);

void exit(code)
int code ;
{

      _exit_code=code ;                  /*Sets return code to _exit_code */

      for(int i=_onexit_count-1; i>0; i--)
       (*_onexit_buf[i])();              /*Sequencially executes functions
                                           defined by onexit*/
      _CLOSEALL();                       /*Closes all files opened*/

      longjmp(_init_inv, 1) ;           /*Returns to the environment saved
                                            by the setjmp*/
}
```

Note: To return to the environment before program execution, create the **callmain** function and call the **callmain** function instead of calling the **main** function from the **init** routine as shown below.

```
#include <setjmp.h>
jmp_buf _init_env;
int     _exit_code;

void callmain()
{

   /* Saves current environment by setjmp function and calls the */
   /*  main function                                             */

   /* Terminates C program if a termination code is returned from the */
   /* exit function                                                    */

if(!setjmp(_init_env))
  _exit_code = main();
}
```

D.3 Creating Abort Routine

To terminate the routine abnormally, the program must be terminated by an abort routine prepared according to the user system specifications. The following shows an example of abort routine in which an error message is output to the standard output device, closes all files, enters endless loop, and waits for reset.

Example:

```
#include <stdio.h>

extern void abort();
extern void _CLOSEALL();

void abort()
{

      printf("program is abort !!\n"); /*Outputs message     */
      _CLOSEALL();                     /*Closes all files     */
      while(1);                        /*Enters endless loop  */
}
```

Appendix E Examples of Low-Level Interface Routine

```
/************************************************************************/
/*                     lowsrc.c:                                        */
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/*         SH-series simulator debugger interface routine               */
/*   - Only standard I/O files (stdin, stdout, stderr) are supported    */
/************************************************************************/
#include <string.h>

/* file number */

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

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

/* file flag */

#define O_RDONLY 0x0001                 /* Read only                        */
#define O_WRONLY 0x0002                 /* Write only                       */
#define O_RDWR   0x0004                 /* Both read and write              */

/* special character code */

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

/* size of area managed by sbrk */

#define HEAPSIZE 1024

/************************************************************************/
/* Declaration of reference function                                    */
/* Reference of assembly program in which the simulator debugger input or  */
/* output characters to the console                                     */
/************************************************************************/
extern void charput(char);              /* One character input              */
extern char charget(void);              /* One character output             */

/************************************************************************/
/* Definition of static variable:                                       */
/* Definition of static variables used in low-level interface routines     */
/************************************************************************/

char flmod[FLMAX];                      /* Open file mode specification area */

static  union  {
  long  dummy ;                         /* Dummy for 4-byte boundary        */
  char heap[HEAPSIZE];                  /* Declaration of the area managed  */
                                        /* by sbrk                          */
} heap_area ;

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

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

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

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

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

     else
           return -1;                    /* Error                        */
}


/*************************************************************************/
/*   close:File close                                                    */
/*         Return value:0 (Pass)                                         */
/*                      -1 (Failure)                                     */
/*************************************************************************/
int close(int fileno)                     /* File number                 */
{
      if(fileno<FLMIN || FLMAX<fileno)    /* File number range check     */
           return -1;

      flmod[fileno]=0;                    /* File mode reset             */
      return 0;
}
```

```
/************************************************************************/
/* read:Data read                                                       */
/*      Return value:Number of read characters (Pass)                   */
/*                   -1                          (Failure)              */
/************************************************************************/
int read(int  fileno,                       /* File number              */
        char *buf,                          /* Destination buffer address  */
        unsigned int  count)                /* Number of read characters   */
{
      unsigned int i;

   /*Check mode according to file name and store each character in buffer */

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

/************************************************************************/
/* write:Data write                                                     */
/*       Return value:Number of write characters  (Pass)                */
/*                    -1                          (Failure)             */
/************************************************************************/
int write(int  fileno,                      /* File number              */
         char *buf,                         /* Destination buffer address  */
         unsigned int  count)               /* Number of write characters  */
{
       unsigned int  i;
       char c;

       /* Check mode according to file name and output each character     */

       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:Definition of file read/write position                          */
/*      Return value:Offset from the top of file read/write position(Pass)*/
/*                   -1             (Failure)                           */
/*      (lseek is not supported in the console input/output)             */
/*************************************************************************/
long lseek(int  fileno,                   /* File number                 */
           long offset,                   /* Read/write position         */
           int  base)                     /* Origin of offset            */
{
     return -1;
}


/*************************************************************************/
/*     sbrk:Data write                                                   */
/*          Return value:Start address of the assigned area (Pass)       */
/*                       -1                                 (Failure)    */
/*************************************************************************/
char  *sbrk(unsigned long size)              /* Assigned area size       */
{
      char  *p ;

      if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size             */
           return (char *)-1 ;

      p=brk ;                                /* Area assignment          */
      brk += size ;                          /* End address update       */
      return p ;
}
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;         lowlvl.src                                                     |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;         SH SERIES SIMULATOR DEBUGGER INTERFACE ROUTINE                 |
;         Input/output one character-                                    |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          .EXPORT     _charput
          .EXPORT     _charget
SIM_IO:   .EQU        H'0080            ;Specifies TRAP_ADDRESS
          .SECTION    P, CODE, ALIGN=4

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;         _charput: One character output                                 |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
          MOV.L       A_DATA, R0        ;Specifies data
          MOV.B       R4, @R0
          MOV.L       A_PARM, R1        ;Specifies parameter block address
          MOV.L       R0, @(4,R1)       ;Specifies data buffer start address
          MOV.L       A_FNO, R0         ;Specifies file number
          MOV.B       @R0, R0
          MOV.B       R0, @(1, R1)
          MOV.L       F_putc, R0        ;Specifies function code
          MOV.L       N_IO, R2
          JSR         @R2
          NOP
          RTS
          NOP
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;         _charget: One character input                                  |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
          MOV.L        A_PARM, R1       ;Specifies parameter block address
          MOV.L        A_DATA, R0       ;Specifies data buffer start address
          MOV.L        R0, @(4,R1)
          MOV.L        A_FNO, R0        ;Specifies file number
          MOV.B        @R0, R0
          MOV.B        R0, @(1, R1)
          MOV.L        F_getc, R0       ;Specifies function code
          MOV.L        N_IO, R2
          JSR          @R2
          NOP
          MOV.L        A_PARAM, R1      ;References data
          MOV.L        @(4,R1), R0
          MOV.B        @R0, R0
          RTS
          NOP
          .ALIGN       4
A_DATA:   .DATA.L      DATA             ;Data buffer start address
A_PARM:   .DATA.L      PARM             ;Parameter block address
A_FNO: .  .DATA.L      FILENO           ;File number area address
F_putc:   .DATA.L      H’01280000       ;fputc function number
F_getc:   .DATA.L      H’01270000       ;fgetc function number
N_IO:     .DATA.L      SIM_IO           ;Trap address
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;          buffer definition                                            |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION    B,DATA,ALIGN=4
PARM:      .RES.L     1               ; Parameter block area
FILENO:    .RES.B     1               ; File number area
DATA:      .RES.B     1               ; Data assign area
           .END
```

Appendix F ASCII Codes

PARITY BIT				b_8								
				b_7	√	√	√	√	—	—	—	—
				b_6	√	√	—	—	√	√	—	—
				b_5	√	—	√	—	√	—	√	—
b_4	b_3	b_2	b_1	MSB / LSB	0	1	2	3	4	5	6	7
√	√	√	√	0	NUL	DC_0	SP	0	@	P	'	p
√	√	√	—	1	SOM	X-ON	!	1	A	Q	a	q
√	√	—	√	2	EOA	TAPE	"	2	B	R	b	r
√	√	—	—	3	EOM	X-OFF	#	3	C	S	c	s
√	—	√	√	4	EOT	TAPE	$	4	D	T	d	t
√	—	√	—	5	WRU	ERROR	%	5	E	U	e	u
√	—	—	√	6	RU	SYNC	&	6	F	V	f	v
√	—	—	—	7	BELL	LEM	'	7	G	W	g	w
—	√	√	√	8	FE_0	CAN	(	8	H	X	h	x
—	√	√	—	9	TAB	S_1	)	9	I	Y	i	y
—	√	—	√	A	LF	EOF	*	:	J	Z	j	z
—	√	—	—	B	VT	ESC	+	;	K	[	k	{
—	—	√	√	C	FF	S_4	'	<	L	\	l	|
—	—	√	—	D	CR	S_5	-	=	M	]	m	}
—	—	—	√	E	S0	S_6	.	>	N	^	n	~
—	—	—	—	F	S1	S_7	/	?	O	-	o	RUB OUT

Notes: √ : Yes
— : No

Index

D

E

F

G

H

I

P

Q

R

S

T

U

V

W

X

Y

Z

_

Symbol

SH Series C Compiler User's Manual

Publication Date: 1st Edition, April 1997
Published by: Semiconductor and IC Div.
Hitachi, Ltd.
Edited by: Technical Documentation Center
Hitachi Microcomputer System Ltd.

SEGA™

1. Intrinsic functions

SH C Ver.5.0 Release0x, Release1x Supplement

Table 1 Intrinsic Functions

No	Item	Functions	Specification	Description
1	Floating-point unit system/control register(FPSCR)	Writes the floating-point unit system/control register	void set_fpscr(int cr)	Writes cr (32bits) to the floating-point unit system/control register.
2		Reads to the floating-point unit system/control register	int get_fpscr()	Returns the floating-point unit system/control register. Returns the FPSCR value.
3	Single-precision floating-point vectors operations	Calculates inner product of vectors	float fipr(float vect1[4], float vect2[4])	Returns the inner product of vect1 and vect2.
4		Multiplies a vector and a matrix	void ftrv (float vect1[4], float vect2[4])	Calculates vect2 =vect1·MTX. MTX is a 4x4 Matrix. Load MTX's data to extension registers using ld_ext(See No.12), before using this function.
5		Multiplies a vector and a matrix and adds another vector	void ftrvad (float vect1[4], float vect2[4], float vect3[4])	Calculates vect3 =vect1·MTX+vect2. MTX is a 4x4 Matrix. Load MTX's data to extension registers using ld_ext(See No.12), before using this function.

No	Item	Functions	Specification	Description
6		Multiplies a vector and a matrix and subtracts another vector	void ftrvsub (float vect1[4], float vect2[4], float vect3[4])	Calculates vect3 =vect1·MTX-vect2. MTX is a 4x4 Matrix. Load MTX's data to extension registers using ld_ext(See No.12), before using this function.
7		Adds two vectors	void add4 (float vect1[4], float vect2[4], float vect3[4])	Calculates vect3 =vect1+vect2.
8	Single-precision floating-point vectors operations	Calculates difference of vectors	void sub4 (float vect1[4], float vect2[4], float vect3[4])	Calculates vect3 =vect1-vect2.
9	Single-precision floating-point 4x4 matrix operation	Multiplies matrices	void mtrx4mul (float mtrx1[4][4], float mtrx2[4][4])	Calculates mtrx2 =mtrx1·MTX. MTX is a 4x4 Matrix. Load MTX's data to extension register using ld_ext(See No.12), before using this function.
10		Multiplies matrices and adds another matrix	void mtrx4muladd (float mtrx1[4][4], float mtrx2[4][4], float mtrx3[4][4])	Calculates mtrx3 =mtrx1·MTX+mtrx2. MTX is a 4x4 Matrix. Load MTX's data to extension register using ld_ext(See No.12), before using this function.
11		Multiplies matrices and subtracts another matrix	void mtrx4mulsub (float mtrx1[4][4], float mtrx2[4][4], float mtrx3[4][4])	Calculates mtrx3 =mtrx1·MTX-mtrx2. MTX is a 4x4 Matrix. Load MTX's data to extension register using ld_ext(See No.12), before using this function.
12	Floating-point extension registers	Loads a matrix data to extension registers	void ld_ext (float mtrx[4][4])	Loads a mtrx to extension registers.
13		Stores a matrix data from extension registers	void st_ext (float mtrx[4][4])	Stores a mtrx from extension registers.

Private intrinsic functions

Table 2 lists private intrinsic functions. These functions are supported by Ver.5.0 Release01.

Usage: <private.h> must be specified when using private intrinsic functions.

Table 2 Private Intrinsic Functions

No	Item	Functions	Specification	Description
1	Sine and cosine	FSCA instruction	void fsca (long rad, float *sinval, float *cosval)	Calculates the approximate value of the sine and cosine of the rad (signed fixed-point number). The sinval is the approximate value of the sine. The cosval is the approximation value of the cosine.
2	Reciprocal of square root	FSSCA instruction	float fssca (float val)	Returns approximate value of the reciprocal of square root of the single-precision floating-point number of val.

Library

There are 4 types of standard library combination for SH4. Link a library listed in table according to the combination of a cpu, pic or endian option.

fpu specification	double=float			
endian specification	Endian = big		endian = little	
Pic specification	pic=0	pic=1	pic=0	pic=1
cpu=sh4	sh4nbfzz.lib	sh4pbfzz.lib	sh4nlfzz.lib	sh4plfzz.lib

Compile Options

Specify -cpu=sh4 -double=float -endian=little options. When you need workaround code (avoids to a right judgement logical bug in SH4-CPU. The bug is SPC(Saved Program Counter)=0 on Set4) for SH4, specify -cpu=sh4, -double=float, -endian=little, -extra=a=400 options.

#pragma aligndata8

The float type array used by an intrinsic function needs to be assigned to 8-byte alignment. These variables must be specified as "#pragma aligndata8 (variable name,...). Refer to the sample below.

Example

```
#pragma aligndata8(vect1,vect2)
float vect1[4]={1.0,2.0,3,0,4.0};
float vect2[4]={4.0,3.0,2.0,1.0};
float func(){
        float retval = fipr(vect1,vect2);
        reutrn retval;
}
```

Note: "$8" is added to the section name. In this example, variables vect1 and vect2 belong to "D$8"section.