Hitachi Microcomputer Support Software

H8S, H8/300 Series C Compiler

USER'S MANUAL

HITACHI

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

Preface

This manual explains the facilities and operating procedures for the H8S, H8/300 series C compiler (Ver. 1.0). The C compiler translates source programs written in C into object programs for Hitachi H8S/2600 series, H8S/2000 series, H8/300H series, H8/300 series, and H8/300L series microcomputers.

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

(1) PART I OVERVIEW AND OPERATIONS

This part overviews the C compiler functions and explains C compiler invoking, optional functions, and listings created by the C compiler.

(2) PART II 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 requirements when installing an object program generated by the C compiler on a system. They are the object program being written in ROM and memory allocation. In addition, specifications of the low-level interface routine must be made by the user when using 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.

Note: For differences from the H8/300 series C compiler (Ver. 2.0), refer to Appendix G, Differences from Older Version.

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 before pressing the key that follows.

PART I

OVERVIEW AND OPERATIONS

Section 1 Overview

The H8S, H8/300 series C compiler (Ver. 1.0) inputs source programs written in C language and outputs them as relocatable object programs or assembly source programs.

The C compiler provides functions for improving object efficiency and supporting program development.

1.1 Functions for Improving Object Efficiency

Table 1-1 lists the functions for improving object efficiency. For details, refer to the related sections.

Table 1-1 Functions for Improving Object Efficiency

Item	Specification Method	Specification	Function	Reference
Short absolute address	Option	abs8	Efficiently accesses 8-bit data in 8-bit short absolute addressing mode.	3.4, Compiler Options in Part I, Overview and Operations, and 3.1.1, Short Absolute Address Specifications in Part II, Programming
	Extended specifications	#pragma abs8		
	Option	abs16	Efficiently accesses data in 16-bit short absolute addressing mode.	
	Extended specifications	#pragma abs16		
enumeration data size reduction	Option	byteenum	Handles enumeration data as 1-byte data.	3.4, Compiler Options in Part I, Overview and Operations
switch statement output code selection	Option	case=ifthen|table	Selects whether to output code for **switch** statements in if-then method or in table method.	
Operation size expanded interpretation	Option	[no]cpuexpand	Efficiently performs data multiplication and division. Note that the range of values guaranteed by C language specifications may be exceeded.	

Table 1-1 Functions for Improving Object Efficiency (cont)

Item	Specification Method	Specification	Function	Reference
Block transfer instruction	Option	eepmov	Generates code for assignment expressions in structs and initial value assignment to local variable arrays into block transfer instructions (eepmov).	3.4, Compiler Options in Part I, Overview and Operations
Function call in memory indirect addressing mode	Option	indirect	Efficiently calls functions in memory indirect addressing mode (@@aa:8).	3.4, Compiler Options in Part I, Overview and Operations, and 3.1.3, Function Call in Memory Indirect Addressing Mode in Part II, Programming
	Extended specifications	#pragma indirect		
Optimization	Options	optimize=1|0	Selects whether or not to optimize the object program.	3.4, Compiler Options in Part I, Overview and Operations
		speed=register| shift| loop| switch| inline| struct	Selects an optimization method to accelerate the object program.	
		[no]volatile	Selects whether or not to optimize external variables	
Register variable assignment expansion	Option	[no]regexpansion	Increases the number of registers to be assigned for register variables.	
In-line expansion	Extended specifications	#pragma inline	Performs in-line expansion of called functions at compilation.	3.1.4, In-Line Expansion for Function, in Part II, Programming

Table 1-1 Functions for Improving Object Efficiency (cont)

Item	Specification Method	Specification	Function	Reference
Register save/restore code specification	Extended specifications	#pragma regsave	Generates code for saving and restoring registers other than ER0 and ER1 (R0 and R1 for H8/300) at function entry and exit.	3.1.7, Register Save/Restore Code Control, in Part II, Programming
		#pragma noregsave	Does not generate code for saving and restoring registers even when the specified function uses registers.	

1.2 Functions Supporting Program Development

Table 1-2 lists the functions supporting program development. For details, refer to the related sections.

Table 1-2 Functions Supporting Program Development

Item	Specification Method	Specification	Function	Reference
Comment nesting	Option	comment	Enables comments to be nested.	3.4, Compiler Options in Part I, Overview and Operations
Object output	Options	code=asmcode|machinecode	Specifies object file output format.	
		[no]object	Selects whether or not to output an object file.	
CPU/ operating mode	Option	cpu=2600n|2600a|2000n|2000a|300hn|300ha|300|300l	Selects CPU and operating mode	1.3, CPU/ Operating Mode Selection in Part I, Overview and Operations
Debugging	Option	[no]debug	Selects whether or not to output symbolic debugging information.	3.4, Compiler Options in Part I, Overview and Operations
Include directory specification	Option	include	Specifies which directories are to be searched for the include file.	
Limit value expansion	Option	limits	Changes limit values such as maximum number of symbols.	
Listing output	Options	[no]list	Selects whether or not to output a listing.	
		show	Specifies the contents and format of the output listing.	
Error level	Option	message	Outputs information-level messages.	

Table 1-2 Functions Supporting Program Development (cont)

Item	Specification Method	Specification	Function	Reference
Section manipulation	Options	section	Specifies an output section name.	3.4, Compiler Options in Part I, Overview and Operations, and 3.1.6, Section Switching in Part II, Programming
		string=const|data	Specifies the output destination for character string data.	
	Extended specifications	#pragma section #pragma abs8 section #pragma abs16 section #pragma indirect section	Switches the section name of the object program within the source program.	
Subcommand file	Option	subcommand	Specifies compiler options in the subcommand file.	3.4, Compiler Options in Part I, Overview and Operations
Assembly language embedding	Extended specifications	#pragma asm #pragma endasm	Embeds assembly language in a C program	3.1.2, Assembly Language Embedded in a C Program in Part II, Programming
Interrupt functions	Extended specifications	#pragma interrupt	Writes interrupt functions in C.	3.1.5, Interrupt Function Creation in Part II, Programming
Intrinsic functions	Extended specifications	Function call format Example: set_imask_ccr( )	Provides functions for processing that cannot be written in C, such as system control instruction or rotate instruction.	3.2, Intrinsic Functions in Part II, Programming

1.3 CPU/Operating Mode Selection

The C compiler supports the following CPUs.

- H8S/2600 series
- H8S/2000 series
- H8/300H series
- H8/300 series
- H8/300L series

The C compiler supports the normal and advanced modes for the H8S/2600, H8S/2000, and H8/300H.

Table 1-3 shows CPU/operating mode selection.

Table 1-3 CPU/Operating Mode Selection

CPU/Operating Mode	Standard Library	Option Specification*2
H8S/2600 in normal mode	c8s26n.lib	cpu=2600n
H8S/2600 in advanced mode	c8s26a.lib	cpu=2600a*3
H8S/2000 in normal mode	c8s26n.lib	cpu=2000n
H8S/2000 in advanced mode	c8s26a.lib	cpu=2000a*3
H8/300H in normal mode	c38hn.lib	cpu=300hn
H8/300H in advanced mode	c38ha.lib	cpu=300ha*3
H8/300*1	c38reg.lib	cpu=300

Notes: 1. The H8/300L-series instructions are the same as those for the H8/300 series; the C compiler assumes the H8/300L series to be the H8/300 series.
2. The CPU/operating mode can also be specified by environment variable H38CPU. For details, refer to section 3.2, Environment Variable H38CPU.
3. For the H8S/2600, H8S/2000, and H8/300H in advanced mode, the bit width of the address space can also be specified. For details, refer to section 3.4, Compiler Options.

Note that load module execution results cannot be guaranteed if object programs or standard libraries are created using different CPU/operating mode and then linked together.

Section 2 Developing Procedures

Figure 2-1 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.

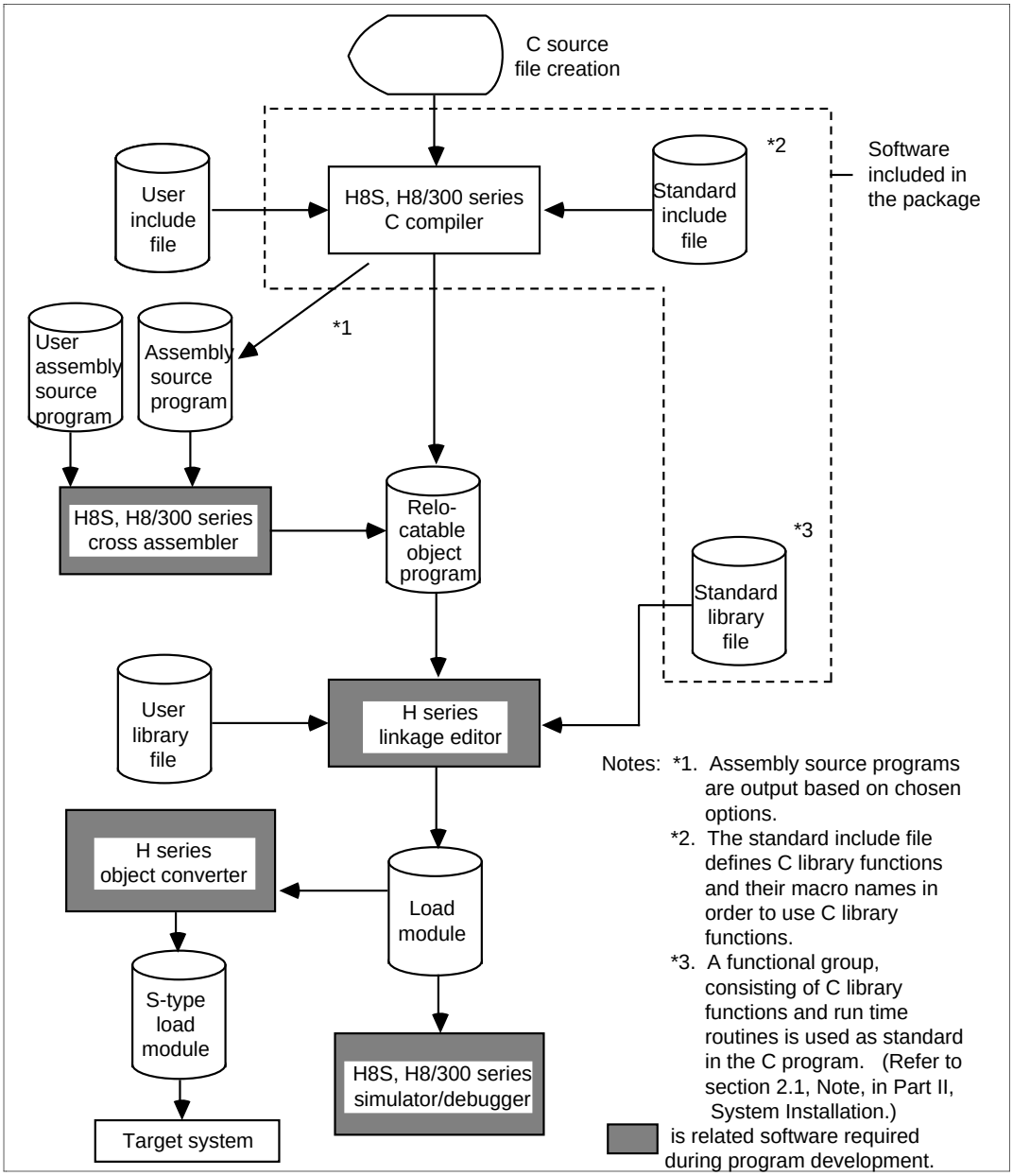


Figure 2-1 Relationship between the C Compiler and Other Software

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.

```
ch38[Δ<option>...][Δ<file name>[Δ<option>...]...]
```

The format for <option> is as follows.

```
<option>[=<suboption>[,<suboption>]...]
```

The general operations of the C compiler are described below.

Invoking C Compiler:

```
ch38 (RET)
```

Instead of compiling, the C compiler outputs the standard command line format and option list.

Compiling Programs:

```
ch38Δtest.c (RET)
```

The C source program test.c is compiled.

The CPU/operating mode must be specified for C compiler operation. If the **cpu** option is not specified, the C compiler uses the H38CPU environment variable specifications. For details on environment variable H38CPU, refer to section 3.2, Environment Variable H38CPU.

C Compiler Options:

```
ch38Δ-cpu=2600aΔ-debugΔtest.c (RET)
ch38Δ-cpu=2600aΔ-debugΔ-show=object,expansionΔtest.c (RET)
```

Insert minus (-) before options **cpu**, **debug**, and **show**. When multiple options are specified, they must be separated by a space (Δ). Multiple suboptions must be separated by a comma (,).

Note: For MS-DOS systems, either a minus or a slash (/) must be specified before options. Multiple suboptions can be enclosed by parentheses () or separated by a space (Δ).

Compiling Multiple Programs:

Several C source programs can be compiled simultaneously.

Example 1: Specifying multiple programs

```
ch38Δ-cpu=2600aΔtest1.cΔtest2.c (RET)
```

Example 2: Specifying options for all C source programs

```
ch38Δ-cpu=2600aΔ-objectΔtest1.cΔtest2.c (RET)
```

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

Example 3: Specifying options for particular special C source programs

```
ch38Δ-cpu=2600aΔtest1.cΔtest2.cΔ-object (RET)
```

The **object** 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 Environment Variable H38CPU

The C compiler uses the specifications for environment variable H38CPU when the **cpu** option is not specified at C compiler initiation. The CPU/operating mode is specified by environment variable H38CPU as follows:

UNIX System:

- C shell

```
setenv H38CPU <CPU/operating mode>[:<bit width of address space>]
```

- Bone shell

```
H38CPU=<CPU/operating mode>[:<bit width of address space>]
export H38CPU
```

MS-DOS System:

```
SET H38CPU=<CPU/operating mode>[:<bit width of address space>]
```

<CPU/operating mode> can be selected from 2600n, 2600a, 2000n, 2000a, 300hn, 300ha, 300, and 300l. When 2600a, 2000a, or 300ha is specified, the bit width of the address space can also be specified. Table 3-1 lists the specifiable bit width.

Table 3-1 Bit Width of Address Space

CPU/Operating Mode	Bit Width of Address Space	Default
2600a	20, 24, 28, or 32	32
2000a	20, 24, 28, or 32	32
300ha	20 or 24	24

The specification of this environment variable is used only when the **cpu** option is not specified.

Note: H38CPU must be specified in upper-case characters.

3.3 File Naming

A standard file extension is automatically added to the name of a file when omitted. The standard file extensions used by the C compiler and related software are shown in table 3-2.

Table 3-2 Standard File Extensions Used by the C Compiler

File Extension	Description
c	Source program file written in C
h	Include file
lst, lis	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: Listing file extension is lst on MS-DOS systems and lis on UNIX systems.

The general rules for naming files depend on the host machine. Refer to the manual of the host machine.

3.4 Compiler Options

Table 3-3 shows C compiler option formats, valid abbreviations, and defaults. Characters underlined indicate the minimum valid abbreviation. For details on option and suboption specifications on the command line, refer to section 3.1, How to Invoke the C Compiler.

Table 3-3 C Compiler Options

Item	Format	Default	Related Extended Specifications
Short absolute address	abs8 abs16	None	#pragma abs8 #pragma abs16 #pragma abs8 section #pragma abs16 section
Enumeration data size	byteenum	None	None
switch statement output code selection	case=ifthen| table	None	None
Comment nesting	comment	None	None
Object type	code=machinecode| asmcode	code=machinecode	None
CPU and operating mode (bit width of address space)	cpu=2600n| 2600a [:<bit width>] | 2000n | 2000a [:<bit width>] | 300hn | 300ha [:<bit width>] | 300 | 300l | 300reg	None	None
Operation size expanded interpretation	cpuexpand nocpuexpand	nocpuexpand	None
Debugging information	debug nodebug	nodebug	None
Macro name	define =<macro name> = <name> | <macro name> = <constant> | <macro name>	None	None
Block transfer instruction	eepmov	None	None

Table 3-3 C Compiler Options (cont)

Item	Format	Default	Related Extended Specifications
Include file	include =<path name>	None	None
Memory indirect addressing	indirect	None	#pragma indirect #pragma indirect section
Limit value expansion	limits=macro=<numeric value>| symbol=<numeric value>| <numeric value>	For UNIX limits=macro=4, symbol=8 For MS-DOS limits=1	None
Listing file	list [= <listing file name>] nolist	list	None
Message output control	message	None	None
Object file	object [= <object file name>] noobject	object	None
Optimization level	optimize =0 | 1	optimize=1	None
Register variable assignment expansion	regexpansion noregexpansion	regexpansion	None
Section name	section =program = <section name> | const = <section name>| data = <section name>| bss =<section name>	section=program=P , const=C, data=D, bss=B	#pragma section
Listings and formats	show =source |nosource | object |noobject | statistics |nostatistics | allocation |noallocation | expansion |noexpansion | width=<numeric value> | length=<numeric value>	show=source, noobject, statistics, noallocation, noexpansion, width=132, length=60	None
Optimization method	speed[=register| shift| loop| switch| inline| struct]	None	#pragma inline

Table 3-3 C Compiler Options (cont)

Item	Format	Default	Related Extended Specifications
Character string output area	string =const | data	string=const	None
Subcommand file	subcommand=<subcommand file name>	None	None
External variable optimization	volatile novolatile	novolatile	None

Each option is described below.

Short Absolute Address:

- Format: abs8
  abs16

- Description

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

  When the **abs8** option is specified, the C compiler generates code for accessing char, unsigned char, and composite data including char or unsigned char element or member in 8-bit absolute addressing mode (@aa:8).

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

  The data specified to be accessed in 8-bit absolute addressing (**abs8** option) is output to section <$ABS8 + C section name>, <$ABS8 + D section name>, or <$ABS8 + B section name>. The data specified to be accessed in 16-bit absolute addressing (**abs16** option) is output to section <$ABS16 + C section name>, <$ABS16 + D section name>, or <$ABS16 + B section name>. The section where the data is output by this option must be allocated to the short absolute address area at linkage.

  For the range of short absolute address area, refer to appendix F, Access Range of Short Absolute Addresses. For section name specification for the short absolute address area, refer to 3.1.6, Section Switching in Part II, Programming.

Enumeration Data Size:

- Format: byteenum
- Description

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

  When all members of the **enum** data is in the range from –128 to 127, the C compiler handles the data as **char** data if this option is specified.

  When this option is not specified or at least one of the members exceeds the range from -128 to 127 even if this option is specified, the **enum** data is handled as int data.

switch Statement Output Code Selection Method:

- Format: case=ifthen|table
- Description

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

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

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

  When this option is not specified, the C compiler selects the method that generates a smaller object. If the **speed** option or **speed=switch** option is specified when the **case** option is not specified, the C compiler selects the method that generates a faster object on the average.

Example:

```
  int a,b;
   :
switch(a){
    case  1:  b=3; break;
    case  2:  b=2; break;
    case  3:  b=1; break;
    default:  b=0; break;
}
```

An example of above C source program expansion is shown below (cpu=2600n):

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

(case=ifthen)

```
MOV.W @_a,R0
SUB.W #1,R0
CMP.W #2,R0
BHI   Ld
ADD.W R0,R0
MOV.W @(L,ER0),R0
JMP   @ER0
    :
L:(jump table)
```

(case=table)

	if_then Method		table Method	
case Value	**Object Size**	**Execution Cycles**	**Object Size**	**Execution Cycles**
1	22 bytes	18	28 (22 + 6) bytes	28
3		30		

Comment Nesting:

- Format: comment
- Description

  Enables nested comments to be written.

  When this option is not specified, if nested comments are written, an error will occur.

  Example:

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

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

Object Type:

- Format: code=machinecode | asmcode
- Description

  Specifies an object program type.

  When the **code=machinecode** option is specified, a relocatable object program that can be input to the linkage editor is generated.

  When the **code=asmcode** option is specified, an assembly source program that can be input to the assembler is generated.

  When the **code=asmcode** and **debug** options are both specified, the **.LINE** directive is output to the assembly source program. At assembly, C source program-level debugging is enabled by specifying the **debug** option.

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

- Note

  When the **code=asmcode** option is specified, the **show=object** option becomes invalid.

CPU/Operating Mode (Bit Width of Address Space):

- Format: cpu=2600n |
  2600a[:<address space bit width>] |
  2000n |
  2000a[:<address space bit width>] |
  300hn |
  300ha[:<address space bit width>] |
  300 | 300l | 300reg

- Description

  Specifies the CPU/operating mode for the object program to be created. Suboptions are listed in table 3-4.

Table 3-4 Suboptions for cpu Option

Suboption	Description
2600n	Creates the object for H8S/2600 in normal mode.
2600a[:<address space bit width>]	Creates the object for H8S/2600 in advanced mode. <address space bit width> must be 20, 24, 28, or 32, which specifies the 1-Mbyte, 16-Mbyte, 256-Mbyte, or 4-Gbyte address space, respectively. The default value for <address space bit width> is 32.
2000n	Creates the object for H8S/2000 in normal mode.
2000a[:<address space bit width>]	Creates the object for H8S/2000 in advanced mode. <address space bit width> must be 20, 24, 28, or 32, which specifies the 1-Mbyte, 16-Mbyte, 256-Mbyte, or 4-Gbyte address space, respectively. The default value for <address space bit width> is 32.
300hn	Creates the object for H8/300H in normal mode.
300ha[:<address space bit width>]	Creates the object for H8/300H in advanced mode. <address space bit width> must be 20 or 24, which specifies the 1-Mbyte or 16-Mbyte address space, respectively. The default value for <address space bit width> is 24.
300	Creates the object for H8/300.
300l	Creates the object for H8/300. This suboption is provided to maintain the compatibility with the cross assembler.
300reg	Creates the object for H8/300. This suboption is provided to maintain the compatibility with the older version of the C compiler.

- Notes

  — When the **cpu** option is not specified, the C compiler uses the H38CPU environment variable specifications. When the **cpu** option and H38CPU environment variable are specified, the C compiler uses the **cpu** option specifications.

  — When neither the **cpu** option nor the H38CPU environment variable is specified, an error will occur.

Operation Size Expanded Interpretation:

- Format: cpuexpand
  nocpuexpand

- Description

  Expands the ANSI C language standard to generate code for multiplication and division of the variables conforming to a volatile attribute.

  When the **nocpuexpand** option is specified, the C compiler generates multiplication and division code conforming to the ANSI C language standard. Table 3-5 shows examples of multiplication and division code generated by specifying this option.

Table 3-5 cpuexpand Option Specifications

	Operation Size of us1 * us2 (for H8S/2600)	
Operation	**cpuexpand Is Specified**	**nocpuexpand Is Specified**

volatile unsigned short us1,us2;
unsigned long ul;
ul=us1*us2;
	Executed as unsigned long.
	Output example:
	`MOV.W @_us1,Rd`
	`MOV.W @_us2,Rs`
	`MULXU.W Rs,ERd`
	`MOV.L ERd,@_ul`
	4-byte result of us1 * us2 is assigned to ul.
		Executed as unsigned short.
		Output example:
		`MOV.W @_us1,Rd`
		`MOV.W @_us2,Rs`
		`MULXU.W Rs,ERd`
		`EXTU.L ERd`
		`MOV.L ERd,@_ul`
		Low-order two bytes of us1 * us2 result are zero-extended and assigned to ul.

volatile unsigned short us1,us2,us3;
unsigned short us;
us=us1*us2/us3;
	Executed as unsigned long.
	Output example:
	`MOV.W @_us1,Rd`
	`MOV.W @_us2,Rs`
	`MULXU.W Rs,ERd`
	`MOV.W @_us3,Rs`
	`DIVXU.W Rs,ERd`
	`MOV.W Rd,@_us`
	4-byte result of us1 * us2 is used as the dividend.
		Executed as unsigned short.
		Output example:
		`MOV.W @_us1,Rd`
		`MOV.W @_us2,Rs`
		`MULXU.W Rs,ERd`
		`EXTU.L ERd`
		`MOV.W @_us3,Rs`
		`DIVXU.W Rs,ERd`
		`MOV.W Rd,@_us`
		Low-order two bytes of us1 * us2 result are zero-extended and used as the dividend.

- Notes

  — Specify the volatile attribute for the variable that is used for expanded interpretation. Even if the **cpuexpand** option is specified, expanded interpretation is not performed for variables that are not volatile.

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

Debug Information:

- Format: debug
  nodebug

- Description

  When the **debug** option is specified, the C compiler outputs debugging information to the object file so that C source program-level debugging can be performed.

  Debugging information is directly output to the relocatable object program. The .**LINE** directives are added to the assembly source program. Therefore, the assembly program output by the C compiler can be debugged at the C source level.

  This option is valid even when the optimizing option is specified.

  When the **nodebug** option is specified, the C compiler does not output debugging information to the object file.

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

- Note

  To perform C source-level debugging, the **debug** option must also be specified at assembly and linkage.

Macro Name Definition:

- Format: define=<macro name>=<name> |
  <macro name>=<constant> |
  <macro name>

- Description

  Defines macro names.

  Table 3-6 shows the macro names, names, and constants that can be specified using a suboption. Up to 31 characters are valid for each macro name, name, and constant.

  When suboption <macro name>=<name> or <macro name>=<constant> is specified, <name> or <constant> is defined as a macro name. When only <macro name> is specified as a suboption, the macro name is assumed to be defined.

  This option enables macro definition on the command line in the same way as that in C source programs.

  Up to 16 suboptions can be specified.

Table 3-6 Macro Names, Names, and Constants Specified by define Option

Item	Description
Macro name	A character string beginning with an alphabetic character or an underscore followed by zero or more alphabetic characters, underscores, and numbers (0 to 9)
Name	A character string beginning with an alphabetic character or an underscore followed by zero or more alphabetic characters, underscores, and numbers (0 to 9)
Constant	A character string of one or more numbers, or a character string of one or more numbers followed by a period (.) and zero or more numbers.

Block Transfer Instruction:

- Format: eepmov
- Description

  Expands to the block transfer instruction, EEPMOV, the assignment statements of structures and initial value assignment expressions for the arrays declared by local variables.

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

- Note

  If an NMI interrupt occurs during EEPMOV execution, control moves to the next instruction after the interrupt processing, and therefore, EEPMOV operation cannot be guaranteed. Precautions must be taken against NMI interrupts when this option is used.

Include File:

- Format: include=<path name>
- Description

  Specifies the name of the path where the include file referred to by the C source program is stored.

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

  For details on how to retrieve the include file, refer to descriptions on the preprocessor specifications in appendix A.1, C Compiler Language Specifications.

Memory Indirect Addressing Mode:

- Format: indirect

- Description

  Uses the memory indirect addressing mode (@@aa:8) for all function calls in the C source program.

  When this option is specified, the C compiler outputs an address table for memory-indirect calls of the functions defined in the C source program to section <$INDIRECT + section name> in addition to the functions themselves. For details on how to specify the section name, refer to section 1.6, Section Switching in Part II, Programming.

- Notes

  — When the **indirect** option is specified, the standard library functions are also called in memory indirect addressing mode. In this case, the addresses of the standard library functions must be set in the address table.

  Example: Address table assembly program

```
          .IMPORT   _printf                  (a)
          .EXPORT   $printf                  (b)
          .SECTION  $INDIRECT,DATA,ALIGN=2
$printf   .DATA.L   _printf                  (c)
          .END
```

  (a): Standard function external reference declaration (_+standard function name)
  (b): Address table external definition declaration ($+standard function name)
  (c): Address table definition

  — The address table can be stored in the area from 0x0000 to 0x00FF. Check that the address table section is located in the area from 0x0000 to 0x00FF using the linkage map at linkage.

Limit Value Expansion:

- Format: limits=macro=<numeric value> |
  symbol=<numeric value> |
  <numeric value>

- Description

  Expands the limit values of macro names using **#define** statements, external identifiers, and internal identifiers.

  The **limits=macro=<numeric value>** option expands the limit value of **#define** macros to 1024 × <numeric value>.

  The **limits=symbol=<numeric value>** option expands the limit value of external identifiers to 512 × <numeric value> - 1 and that of internal identifiers to 512 × <numeric value>.

  The **limits=<numeric value>** option expands the limit value of **#define** macros to 1024 × <numeric value>, that of external identifiers to 512 × <numeric value> - 1, and that of internal identifiers to 512 × <numeric value>.

  1 to 24 can be specified as <numeric value>.

  When this option is not specified, the C compiler assumes that the following is specified:
  limits=macro=4, symbol=8 (UNIX)
  limits=1 (MS-DOS)

Listing File Specification:

- Format: list=[=<listing file name>]
  nolist

- Description

  The **list** option specifies <listing file name>.

  When the **nolist** option is specified, no listing file is created.

  <listing file name> must satisfy the rules described in section 3.3, File Naming.

  If <listing file name> is not specified in the **list** option, the listing file name body becomes the same as that of the source file and the extension becomes lis for UNIX and lst for MS-DOS.

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

Message Output Control:

- Format: message

- Description

  Outputs information-level messages for the C source program.

  When this option is not specified, the C compiler does not output information-level messages.

Object File:

- Format: object [=<object file name>]
  noobject

- Description

  The object file name can be specified with the **object** option.

  When the **noobject** option is specified, no object file is created.

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

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

- Note

  When the **noobject** option is specified, the following options become invalid:
  case,
  code,
  cpuexpand/nocpuexpand,
  debug,
  eepmov,
  optimize,
  regexpansion/noregexpansion,
  section,
  show=object, statistics, allocation,
  speed,
  string,
  volatile/novolatile

Optimization Level:

- Format: optimize=0 | 1
- Description

  Specifies the optimization level for the object program.

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

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

  When this option is not specified, the C compiler assumes that **optimize=1** is specified.

- Note

  When the **optimize=0** option is specified, the following option becomes invalid:
  speed=inline, loop

Register Variable Assignment Expansion:

- Format: regexpansion
  noregexpansion
- Description

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

  When the **noregexpansion** option is specified, the C compiler does not increases the number of registers.

  Generally, variable access speed increases when the registers are increased.

  For details on register variable assignment, refer to the description on registers in appendix A.1, C Compiler Language Specifications.

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

Section Name:

- Format: section=program=<section name> |
  const=<section name> |
  data=<section name> |
  bss=<section name>

- Description

  Specifies section names in the object program.

  For details on section names, refer to section 2.1, Structure of Object Program in Part II, Programming. <section name> must be a character string consisting of alphabetic, or numeric characters, underscores (_), or dollar marks ($), and it cannot begin with a numeric character. Up to 32 characters can be used for a section name.

  When this option is not specified, the C compiler assumes that **section=program=P, const=C, data=D, bss=B** is specified.

Listing and Formats:

- Format: show=source | nosource |
  object | noobject |
  statistics | nostatistics|
  allocation | noallocation |
  expansion | noexpansion
  width=<value> |
  length=<value>

- Description

  Specifies the output listing contents and format or cancels the output. Table 3-7 lists suboptions.

Table 3-7 Suboptions for show Option

Suboption	Description
source	Outputs source program listing.
nosource	Does not output source program listing.
object	Outputs object program listing.
noobject	Does not output object program listing.
statistics	Outputs statistic information listing.
nostatistics	Does not output statistic information listing.
allocation	Outputs symbol assignment information listing.
noallocation	Does not output symbol assignment information listing.
expansion	Outputs source program listing after the include files and macros are expanded. When **nosource** is also specified, **expansion** becomes invalid and the source program listing is not output.
noexpansion	Outputs source program listing before the include files and macros are expanded. When **nosource** is also specified, **expansion** becomes invalid and the source program listing is not output.
width=<value>	Sets the maximum number of characters in one line of listings to <value>. <value> must be 0, or 80 to 132 in decimal. When <value> is 0, the maximum number of characters is not specified.
length=<value>	Sets the maximum number of lines in one page of listings to <value>. <value> must be 0, 20 to 255 in decimal. When <value> is 0, the maximum number of lines is not specified.

For listing examples, refer to section 3.5, C Compiler Listings. When this option is not specified, the C compiler assumes that **show=source, noobject, statistics, noallocation, noexpansion, width=132, length=60** is specified.

Optimization Method:

- Format: speed[=register |
  shift |
  loop |
  switch |
  inline |
  struct]

- Description

  Optimizes the execution speed of the object generated by the C compiler.

  When the **speed=register** option is specified, the C compiler uses the PUSH and POP instruction to save and restore the register contents at entry and exit of functions, without using run-time routines, for the 300ha, 300hn, or 300 CPU/operating mode.

  When the **speed=shift** option is specified, the C compiler outputs a faster object code for shift operations.

  When the **speed=loop** or **speed=switch** option is specified, the C compiler generates a faster code for the for statement or switch statement, respectively.

  When the **speed=inline** option is specified, the C compiler performs in-line expansion of called functions. For details on in-line expansion conditions, refer to section 3.1.4, In-line Expansion of Functions in Part II, Programming.

  When the **speed=struct** option is specified, the C compiler performs in-line expansion of a struct or double assignment statement, without using run-time routines.

  When **speed** is specified without suboptions, all the above optimization methods are used. When this option is not specified, the C compiler reduces the object code size without considering the execution speed.

- Note

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

Character String Output Area:

- Format: string=const | data
- Description

  When the **string=const** option is specified, the C compiler outputs character strings in the C source program to the constant area. The character strings output to the constant area do not need to be transferred to the RAM.

  When the **string=data** is specified, the C compiler outputs the character strings to the initialization data area. The character string output to the initialization data area can be modified at program execution.

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

Subcommand File:

- Format: subcommand=<subcommand file name>
- Description

  Specifies the subcommand file where options used at compiler initiation are stored. The command format in the subcommand file is the same as that on the command line.

  Example:

  When the subcommand file (opt.sub) contents are as follows:

  ```
  -show=object,length=0 -debug -byteenum
  ```

  and when the C compiler is initiated by the following command line:

  ```
  ch38 -cpu=2600a -subcommand=opt.sub test.c
  ```

  the C compiler operates in the same way as when the following is specified:

  ```
  ch38 -cpu=2600a -show=object,length=0 -debug -byteenum test.c
  ```

External Variable Optimization:

- Format: volatile | novolatile
- Description

  When the **volatile** option is specified, the C compiler optimizes external variables. When the **novolatile** option is specified, the C compiler does not optimize them. When this option is not specified, the C compiler assumes that **novolatile** is specified.

3.5 C Compiler Listings

This section deals with C compiler listings and their formats.

Structure of C Compiler Listings: Table 3-8 shows the structure and contents of C compiler listings.

Table 3-8 Structure and Contents of C Compiler Listings

List Structure	Contents	Option Specification Method	Default
Source listing information	Listing consists of source programs *1	show = source show = nosource	Output
	Source program listing after include file and macro expansion *2	show = expansion show = noexpansion	No output
Error information	Errors detected during compilation	—	Output
Symbol allocation information	Variables allocated to stack frames for a function	show = allocation show = noallocation	No output
Object information	Machine code used in object programs and the assembly code	show = object show = noobject	No output
Statistics information	Length of each section (byte), the number of symbols, and object types	show = statistics show = nostatistics	Output

Notes: 1. Source program listings may be output in the object information, depending on the suboption combination. Table 3-9 shows the output destinations of the source program listings depending on the suboption combination.

2. The source program listing after include file and macro expansion is only valid when show = source is specified.

Table 3-9 Source Program Listing Output Destinations

Source Program Listing Output Destination	Suboption Specifications		
	source	expansion	object
Source listing information	Specified	Specified	—
Object information	Specified	Not specified	Specified

Source Listing: The source listing may be output in two ways. When **show = noexpansion** is specified, the unpreprocessed source program is output. When **show = expansion** is specified, the preprocessed source program is output. Figures 3-1 and 3-2 show these output formats, respectively. In addition, figure 3-2 shows the differences between them with bold characters.

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

FILE NAME: m0260.c

  Seq File         Line Pi  0----+----1----+----2----+----3----+----4----+----5--
»
    1 m0260.c         1      #include "header.h"
    4 m0260.c         2
    7 m0260.c         5      int sum2(void)
    8 m0260.c         6      {   int j;
   10 m0260.c         8
   11 m0260.c         9      #ifdef SMALL
   12 m0260.c        10          j=SML_INT;
   13 m0260.c        11      #else
   14 m0260.c        12          j=LRG_INT;
   15 m0260.c        13      #endif
   16 m0260.c        14
   17 m0260.c        15          return j; /*
continue12345678901234567890123456789»
   (1)    (2)         (3)     +2345678901234567890   */
   18 m0260.c        16      (6)
                              }
```

Figure 3-1 Source Listing Output for show=noexpansion

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

FILE NAME: m0260.c

  Seq File          Line Pi 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 m0260.c          2
    7 m0260.c          5     int sum2(void)
    8 m0260.c          6     {   int j;
   10 m0260.c          8
   11 m0260.c          9     #ifdef SMALL
   12 m0260.c         10  X       j=SML_INT;
   13 m0260.c         11  (4) #else
   14 m0260.c         12   E     j=100;
   15 m0260.c         13  (5) #endif
   16 m0260.c         14
   17 m0260.c         15         return j; /*
continue12345678901234567890123456789012345678≫
   (1)   (2)           (3)    +2345678901234567890   */
   18 m0260.c         16     (6)
                              }
```

Figure 3-2 Source Listing Output for show=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) If **show=expansion** is specified and conditional directives such as **#ifdef** and **#elif** are to be compiled, source program lines that are not to be compiled are marked with an X.

(5) If **show=expansion** is specified and **#define** directives are used to expand macros, lines containing a macro expansion are marked with an E.

(6) 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.

Error Information: Figure 3-3 shows an example of error information.

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

FILE NAME: m0260.c

  Seq File          Line Pi 0----+----1----+----2----+----3----+----4----+----5-》
    1 m0260.c          1     #include "header.h"
    4 m0260.c          2
    5 m0260.c          3     extern int sum3(int);
    6 m0260.c          4
    7 m0260.c          5     sum3(int x)
    8 m0260.c          6     {
    9 m0260.c          7         int i;
   10 m0260.c          8         int j;
   11 m0260.c          9
   12 m0260.c         10         j=0;
   13 m0260.c         11         for (i=0; i<=x; i++){
   14 m0260.c         12             j+=k;              ← Error in this line
   15 m0260.c         13         }
   16 m0260.c         14         return j;
   17 m0260.c         15     }
   18 m0260.c         16
   19 m0260.c         17
   20 m0260.c         18


*********** ERROR INFORMATION **********

FILE NAME: m0260.c

File       Line  Erno Lvl  Message
m0260.c      12  2225 (E)  UNDECLARED NAME:  "k"
  (1)        (2)   (3)  (4)      (5)

NUMBER OF ERRORS:          1    (6)
NUMBER OF WARNINGS:        0    (6)
NUMBER OF INFORMATIONS:    0    (7)
```

Figure 3-3 Source Listing Including Errors and Error Information

Description

(1) The name of the source program in which the error occurred is indicated within the limit of ten characters.

(2) The line number containing the error is listed, respectively.

(3) The error number identifies the error message.

(4) (I) Information level
(W) Warning level
(E) Error level
(F) Fatal level

(5) A message is given in upper-case letters.

(6) Total number of error-level messages and the total number of warning-level messages.

(7) Total number of information-level messages (only when the **message** option is specified).

Symbol Allocation Information: Symbol allocation information is the information of function parameters and local variables. Figure 3-4 shows an example of symbol allocation information when a program is compiled in H8S/2600 advanced mode.

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

FILE NAME: m0280.c

  Seq File          Line Pi  0----+----1----+----2----+----3----+----4----+----5-
»
    1 m0280.c          1      extern int h(char, char *, double);
    2 m0280.c          2
    3 m0280.c          3      int
    4 m0280.c          4      h(char a, register char *b,double c)
    5 m0280.c          5      {
    6 m0280.c          6          char   *d;
    7 m0280.c          7
    8 m0280.c          8          d= &a;
    9 m0280.c          9          h(*d, b, c);
   10 m0280.c         10          {
   11 m0280.c         11              register int i;
   12 m0280.c         12
   13 m0280.c         13              i= *d;
   14 m0280.c         14              return i;
   15 m0280.c         15          }
   16 m0280.c         16      }

******* STACK FRAME INFORMATION ********

FILE NAME: m0280.c

Function (File m0280.c   , Line     4): h
               (1)
  Parameter Allocation
    a                                  0xffffffff7 saved from R0L   ┐
    b                                REG ER5       saved from ER1   ┤ (2)
    c                                  0x00000008                   ┘

  Level 1 (File m0280.c   , Line     5) Automatic/Register Variable Allocation  (3)
    d                                  0xffffffff2
  Level 2 (File m0280.c   , Line    10) Automatic/Register Variable Allocation  (3)
    i                                REG R4

Parameter Area Size     : 0x00000008 Byte(s)   ┐
Linkage Area Size       : 0x00000008 Byte(s)   │
Local Variable Size     : 0x00000006 Byte(s)   ┤ (4)
Temporary Size          : 0x00000000 Byte(s)   │
Register Save Area Size : 0x00000008 Byte(s)   │
Total Frame Size        : 0x0000001e Byte(s)   ┘
```

Figure 3-4 Symbol Allocation Information (cpu = 2600a)

Description

(1) File name in which the function is defined, line number, and function name

(2) Parameter allocation

A saved from B: A parameter passed with B is copied to A at the entry of the function.

REG ERx: If a parameter is allocated to a register, REG is indicated.

0xffffffxx: If a parameter is allocated to a stack, the offset from the address by the frame pointer (ER6) is indicated.

(3) Local variable allocation information

This indicates where the local variables declared by a compound statement are stored. If they are allocated to stacks, the offset from the address indicated by ER6 is indicated. If they are allocated to registers, REG is displayed.

(4) Information on stack frames used in functions

Parameter Area Size: The total size of the area for parameters allocated to the stack and return value address area

Linkage Area Size: The total size of the linkage area (return address area and frame pointer stack area)

Local Variable Size: The total size of the parameter save area which is used when the local variable area in the function and parameters passed in registers are allocated to stacks.

Temporary Size: The size of the temporary area used by the C compiler in the function.

Register Save Area Size: The size of the amount of memory required to stack the register contents before the function is executed.

Total Frame Size: The total size of stack frames allocated in the function.

Note: The following message is output instead of parameter allocation and local variable allocation information when the option **optimize = 1** is specified.

```
Optimize Option Specified:  No Allocation Information Available
```

Figure 3-5 shows an example of stack allocation corresponding to the symbol allocation information shown in figure 3-4.

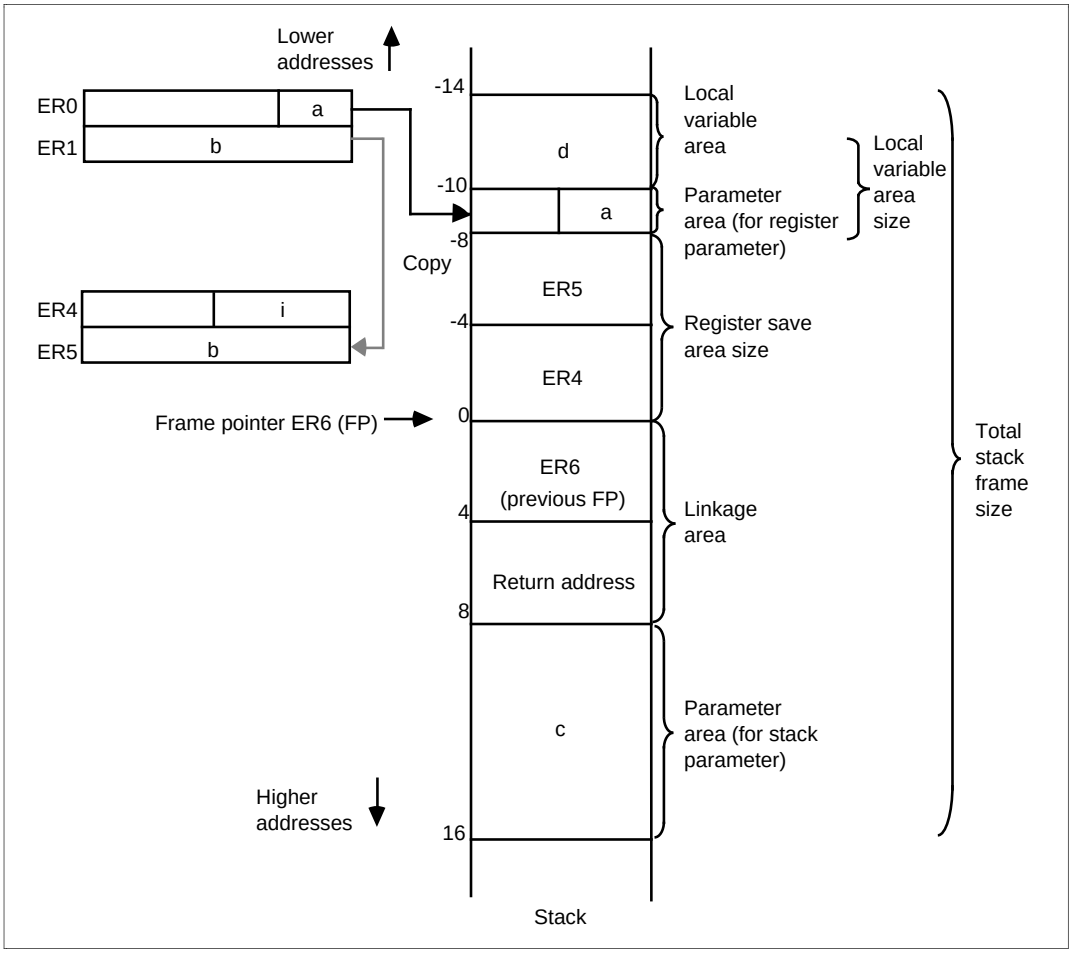


Figure 3-5 Stack Allocation Example (cpu = 2600a)

Object Information: Figures 3-6 and 3-7 show object listing examples when the source program listing is output to the object information and when it is not output.

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

FILE NAME: m0251.c

SCT OFFSET CODE       C LABEL     INSTRUCTION OPERAND    COMMENT
(1)  (2)    (3)                        (4)
P                                                        ;section
                    1:    extern int sum(int);
                    2:          (5)
                    3:    int
                    4:    sum(int x)
 00000000              _sum:                             ; function: sum
                    5:    {
                    6:        int i;
                    7:        int j;
                    8:
                    9:        j=0;
 00000000  1911                SUB.W       R1,R1
                   10:        for (i=0; i<=x; i++){
 00000002  1988                SUB.W       E0,E0
 00000004  4004                BRA         L8:8
 00000006              L7:
                   11:        j+=i;
 00000006  0981                ADD.W       E0,R1
 00000008  0B58                INC.W       #1,E0
 0000000A              L8:
 0000000A  1D08                CMP.W       R0,E0
 0000000C  4FF8                BLE         L7:8
                   12:       }
                   13:       return j;
 0000000E  0D10                MOV.W       R1,R0
                   14:   }
 00000010  5470                RTS
```

Figure 3-6 Object Information When Source Program Listing Is Output (show=source,object, cpu=2600a)

Description

(1) Section attribute (P, C, D, B) of each section

(2) The offset indicates the offset address relative to the beginning of each section

(3) Contents of the offset address of each section

(4) Assembly code corresponding to machine language

(5) Line number and contents of source program

Note: When the **show=expansion** option is specified, the object listing is always output in the format shown in figure 3-7.

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

FILE NAME: m0251.c

SCT OFFSET CODE        C LABEL     INSTRUCTION OPERAND    COMMENT
(1)  (2)    (3)                        (4)
P                                                         ; section
                         ;*** File m0251.c   , Line 4     ; block
 00000000                _sum:                            ; function: sum
                         ;*** File m0251.c   , Line 5     ; block
                         ;*** File m0251.c   , Line 9     ; expression statement
 00000000  1911                  SUB.W       R1,R1
                         ;*** File m0251.c   , Line 10    ; expression statement
 00000002  1988                  SUB.W       E0,E0
                         ;*** File m0251.c   , Line 10    ; for
 00000004  4004                  BRA         L8:8
 00000006                L7:
                         ;*** File m0251.c   , Line 10    ; block
                         ;*** File m0251.c   , Line 11    ; expression statement
 00000006  0981                  ADD.W       E0,R1
                         ;*** File m0251.c   , Line 10    ; expression statement
 00000008  0B58                  INC.W       #1,E0
 0000000A                L8:
 0000000A  1D08                  CMP.W       R0,E0
 0000000C  4FF8                  BLE         L7:8
                         ;*** File m0251.c   , Line 13    ; return
 0000000E  0D10                  MOV.W       R1,R0
                         ;*** File m0251.c   , Line 14    ; block
 00000010  5470                  RTS
```

Figure 3-7 Object Information When Source Program Listing Is Not Output (show=nosource,object, cpu=2600a)

Description

(1) Section attribute (P, C, D, B) of each section

(2) The offset indicates the offset address relative to the beginning of each section

(3) Contents of the offset address of each section

(4) Assembly code corresponding to machine language

Statistics Information: Figure 3-8 shows an example of statistics information.

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                              0x00000012 Byte(s)
CONSTANT SECTION(C):                              0x00000000 Byte(s)
DATA     SECTION(D):                              0x00000000 Byte(s)
BSS      SECTION(B):                              0x00000000 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000012 Byte(s)                              (1)
TOTAL CONSTANT SECTION: 0x00000000 Byte(s)
TOTAL DATA     SECTION: 0x00000000 Byte(s)
TOTAL BSS      SECTION: 0x00000000 Byte(s)

    TOTAL PROGRAM SIZE: 0x00000012 Byte(s)


** ASSEMBLER/LINKAGE EDITOR LIMITS INFORMATION **

NUMBER OF EXTERNAL REFERENCE  SYMBOLS:        0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:        1    (2)
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:          3

********* CPU MODE INFORMATION ********

cpu=2600a    (3)
```

Figure 3-8 Statistics Information

Description

(1) Size of each section and total size of sections

(2) Number of external reference symbols, number of external definition symbols, and total number of internal and external labels

(3) CPU/operating mode specified by the **cpu** option or environment variable.

Note: Statistics information is not output if an error-level error or fatal-level error has occurred or when option **noobject** is specified. In addition, SECTION SIZE INFORMATION is not output when option **code = asmcode** is specified.

PART II

PROGRAMMING

Section 1 Limitations of the C compiler

Table 1-1 shows the limits on source programs that can be handled by the C compiler. Source programs must fall within these limits. The underlined values are those when the **limit** option is not specified. For details, refer to section 3.3, Compiler Options in Part I, Overview and Operations.

Table 1-1 Limitation of the C Compiler

Classification	Item	Limit
Invoking the C compiler	Number of source programs that can be compiled at one time	16
	Total number of macro names that can be specified using the define option	16
	Length of file name	128 characters
Source programs	Length of one line	4096 characters
	Number of source program lines	65535
Preprocessing	Nesting level of files in an **#include** directive	30
	Total number of macro names that can be specified in a **#define** directive *	24576 1024 (MS-DOS) 4096 (UNIX)
	Number of parameters that can be specified using a macro definition or a macro call operation	63
	Depth of the recursive expansion of a macro name	32
	Nesting level 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 external identifiers used for external linkage	12287 511 (MS-DOS) 4095 (UNIX)
	Number of internal identifiers that can be used in one function	12288 512 (MS-DOS) 4096 (UNIX)
	Total number of pointers, arrays, and functions that qualify the basic type	16

Note: The C compiler defines six macro names (_ _ LINE _ _, _ _ FILE _ _, _ _ DATE _ _, _ _ TIME _ _, _ _ STDC _ _ , and _ _ CPU _ _).

Table 1-1 Limitation of the C Compiler (cont)

Classification	Item	Limit
Declarations	Array dimensions	6
(cont)	Array or structure size (byte) * • H8S/2600 normal mode, H8S/2000 normal mode, H8/300H normal mode, or H8/300 • H8/300H advanced mode • H8S/2600 advanced mode or H8S/2000 advanced mode	 65535 16777215 4294967295
Statements	Nesting levels of compound statements	32
	Levels of statement nesting 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	Character array length	512 characters
	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	Approximately 500
C library functions	Number of files that can be opened simultaneously by the open function	20

Note: When the bit width of the address space is specified in advanced mode, if the address space corresponding to the specified bit width is smaller than the above limits, the address space becomes the limit.

Section 2 Executing a C Program

This section covers object programs which are generated by the C compiler. In particular, this section explains what items are required to link C programs with assembly programs and how to install programs on the H8S/2600 system, H8S/2000 system, H8/300H system, or H8/300 system (see Part III, System Installation). This section consists of the following three parts.

Section 2.1 Structure of Object Programs

This section discusses the characteristics of memory areas used for C source programs and standard library functions.

Section 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 referred between C programs, hardware, and assembly programs.

Section 2.3 Linkage with Assembly Programs

This section explains the rules for variable 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. The above information is required for C program functions calling assembly program routines or assembly program routines calling C program functions.

Refer to respective hardware manuals for details on H8S/2600, H8S/2000, H8/300H, and H8/300 hardware.

2.1 Structure of Object Programs

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

- Sections
  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** or **#pragma section**.
- Write Operation
  Indicates whether write operations are enabled at program execution.
- Initial Value
  Shows whether there is an initial value when program execution starts.
- Alignment
  Restricts addresses to which data is allocated.

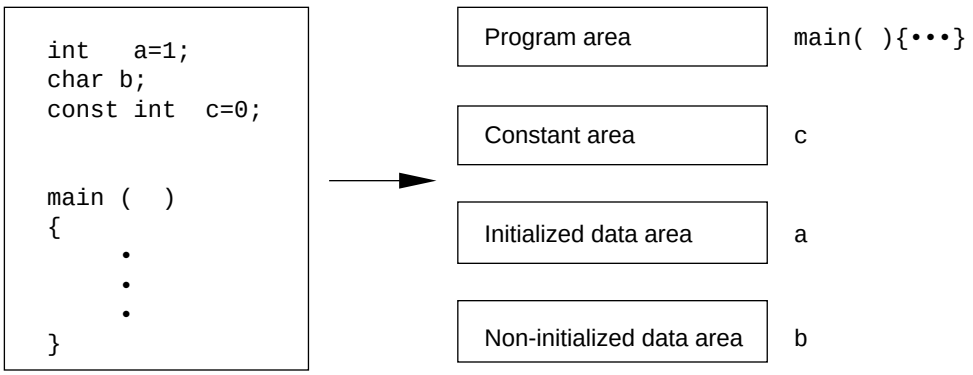
Table 2-1 shows the types and characteristics of those memory areas.

Table 2-1 Memory Area Types and Characteristics

Memory Area Name	Section		Write	Initial		
	Name *	Type	Operation	Value	Alignment	Contents
Program area	P	code	Disabled	Yes	2 bytes	This area stores machine codes.
Constant area	C	data	Disabled	Yes	2 bytes	This area stores const data.
Initialized data area	D	data	Enabled	Yes	2 bytes	This area stores data whose initial values are specified.
Non-initialized data area	B	data	Enabled	No	2 bytes	This area stores data whose initial values are not specified.
Stack area	—	—	Enabled	No	—	This area is allocated at run time and is required for C program execution. Refer to section 2.2, Dynamic Area Allocation, in Part III, System Installation.
Heap area	—	—	Enabled	No	—	This area is used by a C library function (**malloc, realloc,** or **calloc**). Refer to section 2.2, Dynamic Area Allocation, in Part III, System Installation.

Note: Section name shown is the default generated by the C compiler when a specific name is not specified by the compiler option **section** or expanded language specification **#pragma section**.

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



C program Section generated by C compiler

2.2 Internal Data Representation

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

- Size
  Shows the amount of memory needed to store the data.
- Alignment
  Restricts the addresses to which data is allocated. There are two types of alignment, 1-byte alignment in which data can be allocated to any address and 2-byte alignment in which data is allocated to an even byte address.
- Data range
  Shows the range of scalar-type data.
- Data allocation example
  Shows how the elements of aggregate-type data are allocated.

Scalar-Type Data: Table 2-2 shows the internal representation of scalar-type data used in C.

Table 2-2 Internal Representation of Scalar-Type Data

Data Type	Size (bytes)	Alignment (bytes)	Sign Bit	Data Range	
				Minimum Value	Maximum Value
char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
signed char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
unsigned char	1	1	Unused	0	$2^8 - 1$ (255)
short	2	2	Used	-2^{15} (–32768)	$2^{15} - 1$(32767)
unsigned short	2	2	Unused	0	$2^{16} - 1$ (65535)
int	2	2	Used	-2^{15} (–32768)	$2^{15} - 1$ (32767)
unsigned int	2	2	Unused	0	$2^{16} - 1$ (65535)
long	4	2	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
unsigned long	4	2	Unused	0	$2^{32} - 1$ (4294967295)
enum *1	2	2	Used	-2^{15} (–32768)	$2^{15} - 1$ (32767)
float	4	2	Used	– ∞	+ ∞
double, long double	8	2	Used	– ∞	+ ∞
Pointer (H8S/2600 normal, H8S/2000 normal, H8/300H normal, or H8/300)	2	2	Unused	0	$2^{16} - 1$ (65535)
Pointer (H8/300H advanced)	4	2	Unused	0	$2^{24} - 1$ (16777215)*2
Pointer (H8S/2600 advanced or H8S/2000 advanced)	4	2	Unused	0	$2^{32} - 1$ (4294967295)

Notes: 1. When the **byteenum** option is specified, the size and alignment become 1 byte.
2. The lower three bytes indicate address data and the highest byte has an indefinite value.

Aggregate-Type Data: This part explains the internal representation of array, structure, and union data types. Table 2-3 shows the internal data representation of aggregate-type data.

Table 2-3 Internal Representation of Aggregate-Type Data

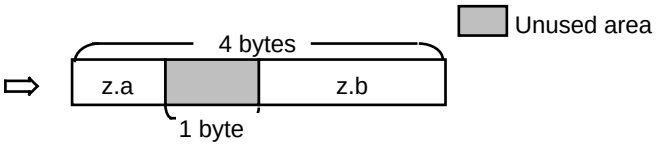
Data type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array type	Array element alignment	(Number of array elements) x (Element size)	`char a [10] ;` Alignment : 1 byte Size : 10 bytes

Structure type	Maximum structure member alignment	Total member size*1	`struct {`
			`  char a, b;`
			`};`
			Alignment : 1 byte
			Size : 2 bytes

Union type	Maximum union member alignment	Maximum value of member size*2	`union {`
			`  char a, b;`
			`};`
			Alignment : 1 byte
			Size : 1 byte

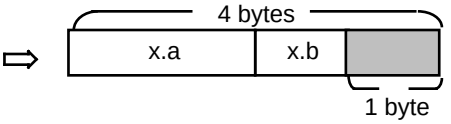
Notes: 1. When structure members are allocated, 1-byte unused area may be generated between structure members to align data types.

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



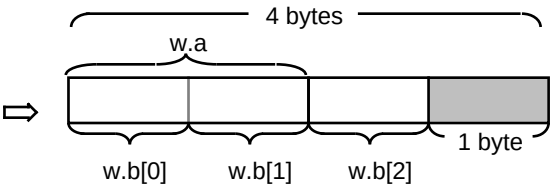
If a structure has 2-byte alignment and the last member ends at an odd-byte address, the following 1-byte is included in this structure.

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



2 When an union has 2-byte alignment and its maximum member size is odd, the following 1-byte is included in this union.

```
union {
     int a;
     char b[3];
  }w;
```



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

- Bit field members
  Table 2-4 shows the specifications of bit field members.

Table 2-4 Bit Field Member Specifications

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

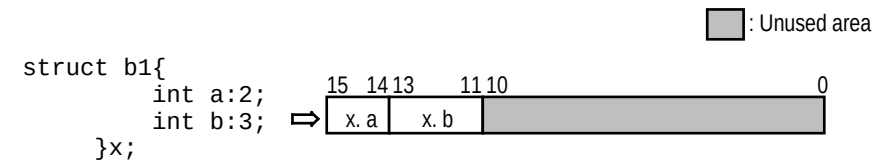
Notes: 1. To use a member of a bit field, data in the bit field is expanded to the declared type.
2. Zero extension: Zeros are written to the high-order bits during extension.
Sign extension: The most significant bit of a bit field is used as a sign and is written to the high order bits during extension.

Note: 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.

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

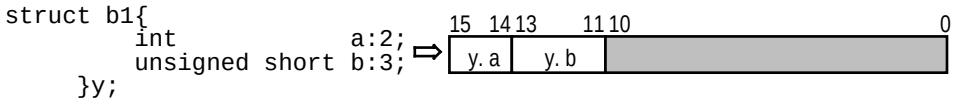
  — Bit field members are placed in an area beginning from the left, that is, the most significant bit.

  Example:



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

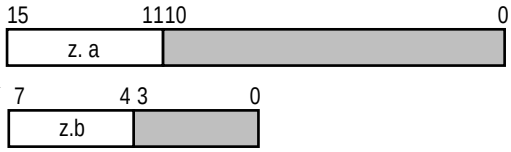
  Example:



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

Example:

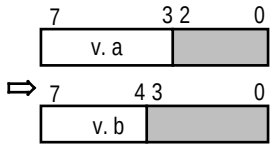
```
struct b1{
         int  a:5;
         char b:4;
      }z;
```



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

Example:

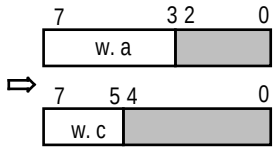
```
struct b2{
       char a:5;
       char b:4;
     }v;
```



— If an anonymous bit field member or 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;
```



2.3 Linkage with Assembly Programs

Because C is suitable for writing system programs, it can be used in almost every program in application systems for microcomputers. Especially, this C compiler supports an assembly language embedding function and intrinsic functions to enable all programs to be written in C.

However, when the required specifications, such as hardware timing or memory size limitation, are severe, some processes must be written in assembly language, and then linked to the C program.

This section explains the 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 or function 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 or function name (up to 31 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=2
_a: .DATA.W 1
_b: .DATA.W 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)

```
char a,b;
```

Assembly program (reference)

```
.IMPORT  _a,_b
.SECTION P,CODE,ALIGN=2

MOV.B    @_a,R5L
MOV.B    R5L,@_b
RTS
.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:

- Stack Pointer
- Allocating and Deallocating Stack Frames
- Registers
- 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, since the data may be destroyed by an interrupt process.

Allocating and Deallocating Stack Frames: In a function call (right after a JSR or a BSR instruction has been executed), the stack pointer indicates the return address area. Allocating and setting data to a higher address than this area is a role of the calling function. When control returns from a function, the called function deallocates the return address area, usually with a RTS instruction, while the calling side deallocates the area having an address higher than the return value address and the parameter area.

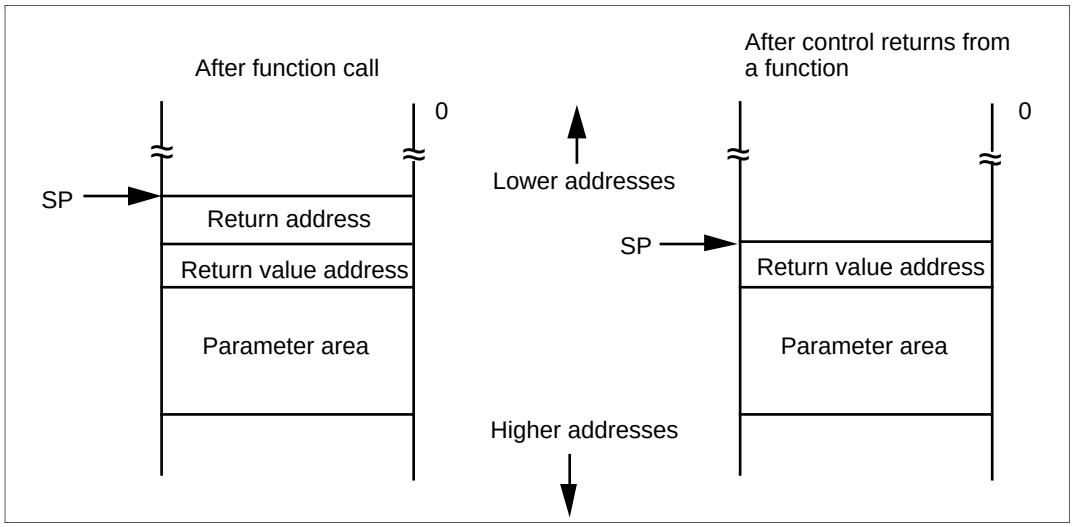


Figure 2-1 Allocation and Deallocation of a Stack Frame

Registers: Some registers change after a function call, while some do not. Table 2-5 shows how registers of each CPU type change according to the rules.

Table 2-5 Rules on Changes in Registers After a Function Call

	CPU Type and Registers Used in a Function		
Item	**H8S/2600, H8S/2000, and H8/300H**	**H8/300**	**Notes on Programming**
Non-guaranteed registers	ER0 and ER1	R0 and R1	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.
Guaranteed registers	ER2 – ER6	R2 – R6	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.

The following examples show the rules governing register changes that take place in H8S/2600 advanced mode.

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

Assembly program (called program)

```
      .EXPORT  _sub
      .SECTION P,CODE,ALIGN=2
_sub: STM.L    (ER4-ER6),@-SP
      SUB.L    #10,SP
            .
            .
            .
      ADD.L    #10,SP
      LDM.L    @SP+,(ER4-ER6)
      RTS
      .END
```

The data in the registers used in the function is pushed on to the stack.

Function processing
(ER0 and ER1 register data is not guaranteed, and they can be used without stacking them.)

Register data is popped from the stack.

C program (calling program)

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

- A subroutine 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
            .
            .
            .
      MOV.L    ER1,@(4,SP)
      MOV.L    ER0,ER6
      JSR      @_sub
            .
            .
            .
      .END
```

If registers ER0 and ER1 contain valid data, the data is pushed onto the stack or stored in unused registers.

The sub function is called.

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

The rest of this section explains the general rules concerning parameters and return values, 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 after parameters are copied to the parameter area. 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 transferred or a return value is returned. This section explains the rules on type conversion.

  (a) Return value type conversion
  A return value is converted to the data type returned by the function.

  (b) Type conversion of parameters whose types are declared
  Parameters whose types are declared by prototype declaration are converted to the declared types.

  (c) 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:
  - Parameters whose types are **c**har or unsigned char are converted to int.
  - Parameters whose types are float are converted to double.
  - Other parameters are not converted.

Example :

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

The return value is converted to long.

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

c is converted to int because no type is declared for the parameter.

1.0 is converted to int because int type is declared for the parameter.

Note: When parameter types are not declared by a prototype declaration, the correct specifications must be made by the calling and called functions so that parameters are correctly transferred. Otherwise, correct operation is not guaranteed.

Example :

```
f(X)
float x;
{
   .
   .
   .
}

main()
{
   float x;
   f(x)
}
```

Incorrect specification

```
f(float x)
{
   .
   .
   .
}

main()
{
   float x;
   f(x)
}
```

Correct specification

Since the parameter type belonging to function f is not declared by a prototype declaration in the incorrect specification above, parameter x is converted to double when function main calls function f. Function f cannot receive the parameter correctly because the parameter type is declared as float in function f. Use the prototype declaration to declare the parameter type, or make the parameter declaration double in function f.

The parameter type is declared by a prototype declaration in the correct specification above.

- Parameter area allocation
  Parameters are allocated to either registers or a stack parameter area. Figure 2-2 shows the parameter area allocation for each object type. Table 2-6 lists the general parameter area allocation rules.

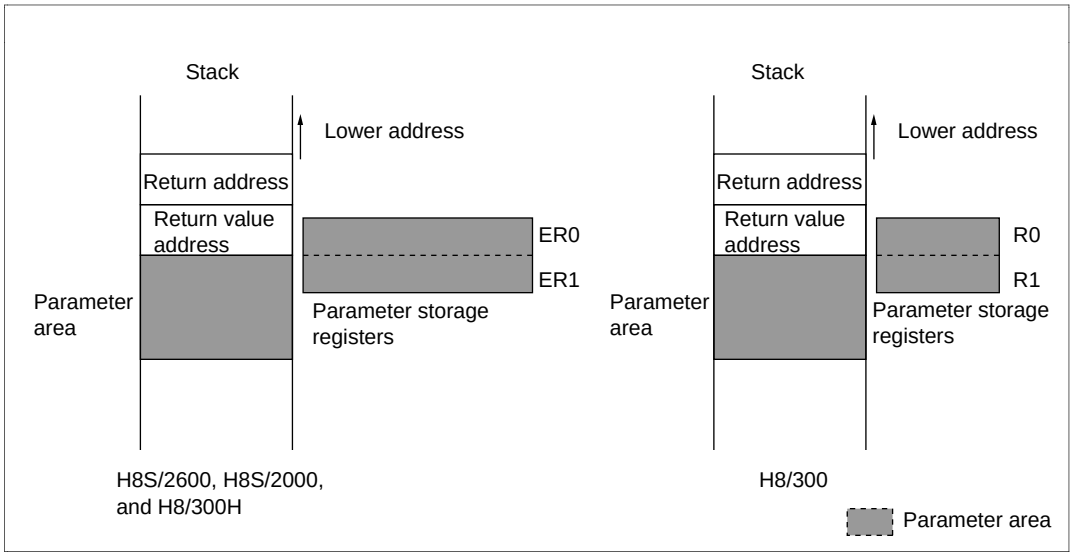


Figure 2-2 Parameter Area Allocation

Table 2-6 General Rules on Parameter Area Allocation

	Allocation Rules		
	Parameters Allocated to Registers		
CPU Type	Parameter Storage Registers	Target Type	Parameters Allocated to a Stack
H8S/2600, H8S/2000, and H8/300H	ER0 and ER1	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, and pointer	• Parameters whose types are other than target types for register passing • Parameters of a function which has been declared by a prototype declaration to have variable-number parameters* • Parameters which could not be allocated to registers because multiple parameters are allocated
H8/300	R0 and R1	char, unsigned char, short, unsigned short, int, unsigned int, and pointer	

Note: If a function has been declared to have variable-number parameters by a prototype definition, 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,...);
      :
      f2(x,y,z);
      :
```

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 from the LSB of the smallest numbered register. Figure 2-3 shows an example of parameter allocation to storage registers.

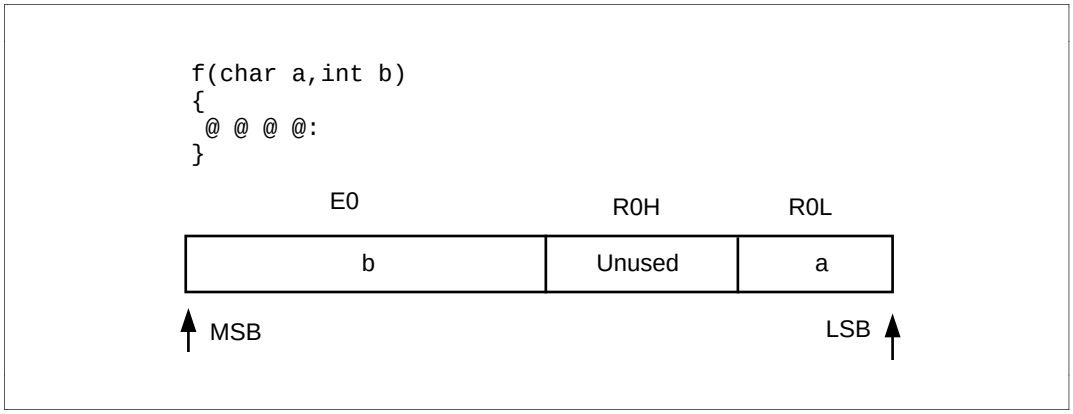


Figure 2-3 Example of Allocation to Parameter Storage Registers (H8S/2600)

  — 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, structure type or union type parameters are allocated using 2-byte alignment and an even numbered byte area for each parameter. This is because the H8S and H8/300-series stack pointer is incremented or decremented in 2-byte units.

Refer to appendix B, Parameter Allocation Examples, for examples of parameter allocation for each CPU/operating mode.

- Return value setting location
  The return value is written to either a register or memory depending on its type. Refer to table 2-7 for the relationship between the return value type and setting location.

  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 calling side must allocate this 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. The return value is not written if its type is void.

Table 2-7 Return Value Type and Setting Location

	Return Value Setting Location	
Return Value Type	**H8S/2600, H8S/2000, and H8/300H**	**H8/300**
char and unsigned char	Register(R0L)	Register (R0L)
short, unsigned short, int, and unsigned int	Register(R0)	Register (R0)
Pointer	Registers: Normal mode: (R0) Advanced mode: (ER0)	Register(R0)
long, unsigned long, and float	Register (ER0)	Return value setting area (memory)
double, long double, structure, and union	Return value setting area (memory)	Return value setting area (memory)

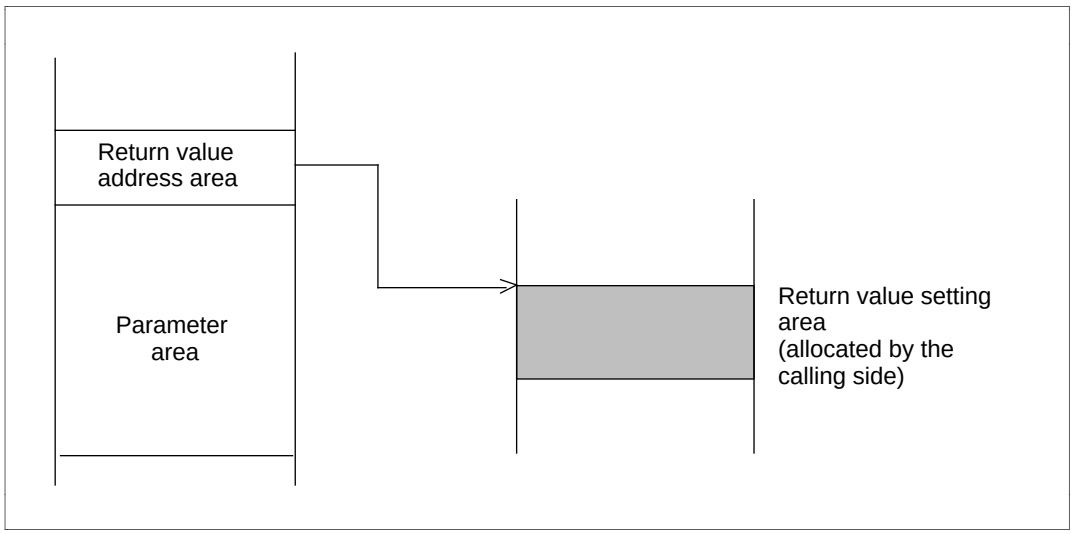


Figure 2-4 Return Value Setting Area Used When Written to Memory

Section 3 Extended Specifications

The C compiler supports the following two kinds of extended specifications:

- **#pragma** extension
- Intrinsic functions

3.1 #pragma Extension

Table 3-1 lists **#pragma** extension specifiers.

Table 3-1 #pragma Extension Specifier List

#pragma Extension Specifier	Function
#pragma abs8, #pragma abs16	Specifies the short absolute addressing mode
#pragma asm, #pragma endasm	Embeds assembly language instructions in a C source program
#pragma indirect	Calls functions in memory indirect addressing mode
#pragma inline	Performs in-line expansion of functions
#pragma interrupt	Writes an interrupt function in C
#pragma section, #pragma abs8 section, #pragma abs16 section, #pragma indirect section	Switches sections in a C source program
#pragma regsave, #pragma noregsave	Saves and restores register contents at the entry and exit of functions

3.1.1 Short Absolute Address Specifications

#pragma abs8 and **#pragma abs16** generate code for accessing variables allocated in 8-bit and 16-bit absolute address area in short absolute addressing mode. For details on the short absolute addressing area, refer to appendix F, Access Area in Short Absolute Addressing.

The **abs8** and **abs16** options can generate code for accessing in short absolute addressing mode in compilation units. However, **#pragma abs8** and **#pragma abs16** can specify short absolute addressing in variable units.

Description Format:

```
#pragma abs8 (<variable name> [, <variable name>],...)
#pragma abs16 (<variable name> [, <variable name>],...)
```

Note: **#pragma abs8** and **#pragma abs16** musts be declared before the variables used are declared.

Explanation:

The variables declared in **#pragma abs8** are output to sections <$ABS8 + C section name>, <$ABS8 + D section name>, and <$ABS8 + B section name>, and the code for accessing them in 8-bit absolute addressing mode (@aa:8) is generated.

The variables declared in **#pragma abs16** are output to sections <$ABS16 + C section name>, <$ABS16 + D section name>, and <$ABS16 + B section name>, and the code for accessing them in 16-bit absolute addressing mode (@aa:16) is generated.

For details on specifying the C section name, D section name, and B section name, refer to description on **#pragma abs8 section** and **#pragma abs16 section** in section 3.1.6, Section Switching.

Notes:

1. Only the variables to be allocated to the static area can be specified with **#pragma abs8** and **#pragma abs16**.

2. Up to 63 variables can be specified in one **#pragma abs8** or **#pragma abs16** statement.

3. The variables specified with **#pragma abs8** and **#pragma abs16** are output to section $ABS8C, $ABS8D, $ABS8B, $ABS16C, $ABS16D, or $ABS16B when the section switching function is not used. Allocate the section to the 8-bit or 16-bit absolute addressing area at linkage.

Example:

```
#pragma abs8(c)
#pragma abs16(i)
char c;
int i;
long l;
f( ){
    c=10;
    i=100;
    l=1000;
}
```

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .EXPORT   _c
      .EXPORT   _i
      .EXPORT   _l
      .EXPORT   _f
      .SECTION  P,CODE,ALIGN=2
_f:
      MOV.B     #10:8,R0L
      MOV.B     R0L,@_c:8       ;Accesses c in 8-bit short absolute addressing mode
      MOV.W     #100:16,R0
      MOV.W     R0,@_i:16       ;Accesses i in 16-bit short absolute addressing mode
      MOV.L     #1000:32,ER0
      MOV.L     ER0,@_l:32      ;Accesses l in 32-bit absolute addressing mode
      RTS
      .SECTION  $ABS8B,DATA,ALIGN=2
_c:
      .RES.B    1               ;Allocates c to section $ABS8B
      .SECTION  $ABS16B,DATA,ALIGN=2
_i:
      .RES.W    1               ;Allocates i to section $ABS16B
      .SECTION  B,DATA,ALIGN=2
_l:
      .RES.L    1               ;Allocates l to section B
      .END
```

3.1.2 Assembly Language Embedded in a C Program

#pragma asm and **#pragma endasm** embed assembly language instructions in a C program.

Description Format:

```
#pragma asm
     <assembly language program>
#pragma endasm
```

Explanation:

The assembly language instructions must be preceded by **#pragma asm** and be followed by **#pragma endasm**.

The C compiler inserts the assembly language instructions enclosed by **#pragma asm** and **#pragma endasm** into the object code generated by the C compiler.

Notes:

1. Specify assembly program output with **code=asmcode** when compiling. If not specified, the assembly language instructions enclosed by **#pragma asm** and **#pragma endasm** are ignored.

2. The C compiler checks neither the syntax of the assembler instructions, nor their influence over the code generated by the C compiler. When the **optimize=1** or **speed** option is specified when compiling, the expanded code or location of the assembler instructions may differ from that specified using **#pragma asm** and **#pragma endasm**. Check the output code and program operation by yourself, when using this function.

3. The **#pragma asm** and **#pragma endasm** specification cannot be nested. If attempted, an error will occur.

4. If **#pragma asm** and **#pragma endasm** are specified in a conditional or loop statement, the assembly language instructions including **#pragma asm** and **#pragma endasm** must be enclosed by { }. If not, results are not guaranteed.

   Example:

   ```
   while(a==0)
   {                                   Must be specified.
   #pragma asm
      <assembly language program>
   #pragma endasm
   }                                   Must be specified.
   ```

Example:

```
void main()
#pragma asm
     MOV.L #H'FFFFFFFE,SP     ;The stack address is set before
#pragma endasm                ;the main function
{
        :
}
```

3.1.3 Function Call in Memory Indirect Addressing Mode

#pragma indirect calls functions in memory indirect addressing mode (@@aa:8).

The **indirect** option can also call functions in memory indirect addressing mode in compilation units. However, **#pragma indirect** can specify it in function units.

Description Format:

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

Note: **#pragma indirect** must be declared before the functions used are declared or defined.

Explanation:

#pragma indirect specifies the functions to be called in memory indirect addressing mode.

The function declared in the **#pragma indirect** statement is called in the format JSR @@<$ + function name>:8. When the function declared in the **#pragma indirect** statement is defined, the <$ + function name> label and the function address are stored in section <$INDIRECT + section name> as the address table for function call.

For detail on how to specify the section name, refer to the description on **#pragma indirect section** in section 3.1.6, Section Switching.

Notes:

1. Up to 63 functions can be specified in one **#pragma indirect** statement.
2. Up to 64 functions can be specified in total when the CPU/operating mode is 2600a, 2000a, or 300ha, and up to 128 when the CPU/operating mode is 2600n, 2000n, 300hn or 300.
3. The address table for the functions specified with **#pragma indirect** is output to section $INDIRECT when the section switching function is not used.
4. Allocate the address table section within addresses H'0x0000 to 0x00FF at linkage.

Example:

```
#pragma indirect (f)      /*Declares that f is accessed in memory indirect addressing*/
#define A *((unsigned char *)0xffffff00)
unsigned char a;
unsigned char f(void)
{
    sub( );
    return(A&1);
}
sub ( )
{
    a=f( );               /* Function call */
}
```

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .EXPORT   _a
      .EXPORT   _f
      .EXPORT   $f              ;Externally defines address table label
      .EXPORT   _sub
      .SECTION  P,CODE,ALIGN=2
_f:
      BSR       _sub:8
      MOV.B     @-256:8,R0L
      AND.B     #1:8,R0L
      RTS
_sub:
      JSR       @@$f:8          ;Calls a function in memory indirect addressing
      MOV.B     R0L,@_a:32
      RTS
      .SECTION  $INDIRECT,DATA,ALIGN=2
$f:                             ;Address table label
      .DATA.L   _f              ;Address table constant
      .SECTION  B,DATA,ALIGN=2
_a:
      .RES.B    1
      .END
```

3.1.4 In-Line Expansion for Function

#pragma inline performs in-line expansion for the specified functions.

The **speed=inline** option can also perform in-line expansion. However, **#pragma inline** can specify it in function units regardless of optimization specification.

Description Format:

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

Explanation:

#pragma inline specifies the functions for which in-line expansion is performed.

The code for the function declared in the **#pragma inline** statement is directly generated at the location where the function is called. The code for calling the function by the JSR or BSR instruction is not generated.

Notes:

1. Up to 63 functions can be specified in one **#pragma inline** statement.

2. When the function declared by **#pragma inline** satisfies one of the following conditions, in-line expansion will not be performed:
   - A variable parameter is used.
   - A parameter address is referenced.
   - The actual parameter type does not match the formal parameter type.
   - A **switch** statement is included.
   - Another function for which in-line expansion is specified is called.
   - The maximum size of in-line expansion is exceeded.

3. Even when the function declared by **#pragma inline** is expanded at all calling locations, the code for the function definition itself is also output.

Example:

```
#pragma inline (f)        /*Declares that in-line expansion is performed for function f  */
int a,b,c;
int f(int x,int y)
{
    return x+y;
}
sub ( )
{
    a=f(b,c);             /* Directly expanded to a = b + c  */
}
```

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .EXPORT   _a
      .EXPORT   _b
      .EXPORT   _c
      .EXPORT   _f
      .EXPORT   _sub
      .SECTION  P,CODE,ALIGN=2
_f:
      MOV.B     R0,R1
      ADD.W     E0,R1
      MOV.W     R1,R0
      RTS
_sub:
      MOV.W     @_c:32,E0       ;
      MOV.W     @_b:32,R0       ; Directly expanded
      ADD.W     E0,R0           ; to the code for
      MOV.W     R0,R1           ; a = b + c
      MOV.W     R1,@_a:32       ;
      RTS
      .SECTION  B,DATA,ALIGN=2
_a:
      .RES.W    1
_b:
      .RES.W    1
_c:
      .RES.W    1
      .END
```

3.1.5 Interrupt Function Creation

#pragma interrupt enables an external (hardware) interrupt function to be written in C.

Description Format:

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

Table 3-2 lists interrupt specifications.

Table 3-2 Interrupt Specifications

Item	Form	Options	Specifications
Stack switching	sp=	<variable> | &<variable> | <constant> | <variable>+<constant> | &<variable>+<constant>	The address of a new stack is specified with a variable or a constant. <variable>: Variable (pointer type) &<variable>: Variable (object type) address <constant>: Constant value
Trap-instruction return	tn=	<constant>	Termination is specified by the TRAPA instruction <constant>: Constant value (trap vector number)
Interrupt function termination	sy=	<function name> | <constant> | $<function name>	Termination is specified by a jump instruction to an interrupt function <function name>: Interrupt function name <constant>: Absolute address $<function name>: Interrupt function name without an underscore (_)

Explanation:

#pragma interrupt declares an interrupt function.

An interrupt function declared by **#pragma interrupt** will preserve register values before 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 (tn=) is specified, the TRAPA instruction is executed at the end of the function. If the interrupt function termination (sy=) is specified, the JMP instruction is executed to jump to the specified address.

As the function name for interrupt function termination specification, $ + <function name> can be specified as well as <function name>. If $ + <function name> is specified, no underscore (_) is added before the function name, which is used as an external identifier.

An interrupt function with no specifications is processed in the usual procedure.

Trap-instruction return and interrupt function termination cannot be specified at the same time, but stack switching and trap-instruction return, and stack switching and interrupt function termination can be specified at the same time.

Example:

```
extern int  STK[100];
#pragma interrupt ( f(sp=STK+100, tn=2) )
                              (1)      (2)
```

Explanation:

(1) Stack switching specification
STK+100 is set as the stack pointer used by interrupt function f.

(2) Trap-instruction return specification
After the interrupt function has completed its processing, TRAPA #2 is executed. The SP at the beginning of trap exception processing is shown in the figure below. After the previous PC and CCR (condition code register (EXR (extend register) in H8S series)) are popped from the stack by the RTE instruction in the trap routine, control is returned from the interrupt function.

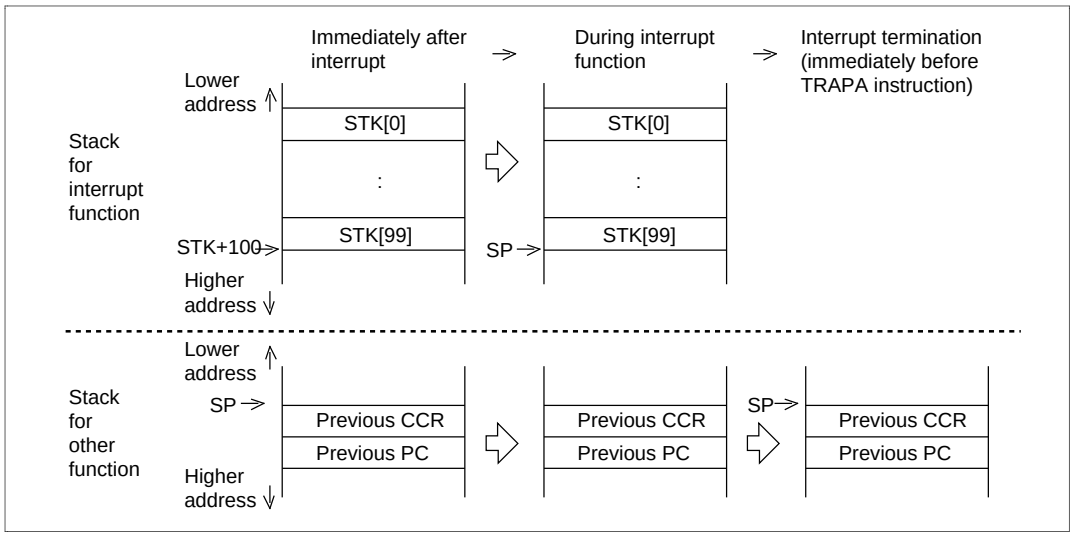


Figure 3-1 Stack Processing by an Interrupt Function

Example:

```
#pragma interrupt (f1(sy=$f2))
char x;
void f1(void)
{
    x=0;
}
```

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .IMPORT   f2              ; Underscore (_) is not added
      .EXPORT   _f1
      .EXPORT   _x
      .SECTION  P,CODE,ALIGN=2
_f1:
      PUSH.W    R0
      SUB.B     R0L,R0L
      MOV.B     R0L,@_x:32
      POP.W     R0
      JMP       @f2:24          ; Return by JMP @f2
      .SECTION  B,DATA,ALIGN=2
_x:
      .RES.B    1
      .END
```

Notes:

1. The storage class specifier of the interrupt function 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)
   ```

   Explanation:

   (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 cannot be guaranteed.

   Example (An interrupt function is declared):

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

   Explanation:

   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 f2(){ f1();} ..... (b)
```

   Explanation:

   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 not to be compiled in the same file as f2, correct program execution is not guaranteed.

3. A function declared as an interrupt function cannot be referenced in the same file.

   Example:

```
#pragma interrupt(f1)
main(){
          void (*a)(void);
          a=f1; ..... (a)
}
```

   Explanation:

   Since the address of interrupt function f1 cannot be referenced at (a), an error is output.

   If an interrupt function is referenced to set, for example, a vector table, it must not be declared as an interrupt function in the same file.

Examples:

```
#pragma interrupt(f1)
      :
void f1(void)
{
      :
   }
```

File with an interrupt function definition

```
extern void f1(void); .... (b)
main()
{
     void (*a)(void);
     a=f1;
}
```

File referencing an interrupt function

Explanation:

To reference the address of interrupt function f1 at (b), f1 is not declared as an interrupt function.

4. Up to 63 functions can be declared in one **#pragma interrupt** statement.

5. When stack switching is specified, the linkage area size in symbol assignment information output in the compile listing includes the size of the area for saving previous SP and ER0 (R0 in H8/300) required for SP calculation.

3.1.6 Section Switching

#pragma section switches sections output by the C compiler within a C program. Using conventional compilers, a program must be separated into several files to assign addresses in function or variable units. Using the section switching function, a program need not be separated.

Description Format:

```
#pragma section [<name>|<value>]
#pragma abs8 section [<name>|<value>]
#pragma abs16 section [<name>|<value>]
#pragma indirect section [<name>|<value>]
```

Explanation:

#pragma section <name> or **#pragma section <value>** specifies a section name. The name of the section after the declaration in the source program is "P section name + <name> (<value>)", "C section name + <name> (<value>)", "D section name + <name> (<value>)", or "B section name + <name> (<value>)". When <name> or <value> is omitted, the default name is used for the rest of the section.

#pragma abs8 section <name> or **#pragma abs8 section <value>** specifies a section name for 8-bit absolute address area. The name of the section for the 8-bit absolute address area after the declaration in the source program is "$ABS8C + <name> (<value>)", "$ABS8D + <name> (<value>)", or "$ABS8B section name + <name> (<value>)". When <name> or <value> is omitted, $ABS8C, $ABS8D, or $ABS8B is used for the rest of the section.

#pragma abs16 section <name> or **#pragma abs16 section <value>** specifies a section name for 16-bit absolute address area. The name of the section for 16-bit absolute address area after the declaration in the source program is "$ABS16C + <name> (<value>)", "$ABS16D + <name> (<value>)", or "$ABS16B section name + <name> (<value>)". When <name> or <value> is omitted, $ABS16C, $ABS16D, or $ABS16B is used for the rest of the section.

#pragma indirect section <name> or **#pragma indirect section <value>** specifies a section name for outputting the vector table for the functions called in memory indirect addressing mode. The name of the section after the declaration in the source program is "$INDIRECT + <name> (<value>)". When <name> or <value> is omitted, $INDIRECT is used for the rest of the section.

Notes:

1. Declare **#pragma section**, **#pragma abs8 section**, **#pragma abs16 section**, and **#pragma indirect section** outside the function definition.

2. Up to 64 section names can be declared for each of **#pragma section**, **#pragma abs8 section**, **#pragma abs16 section**, and **#pragma indirect section** in one file.

Example:

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

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .EXPORT   _a
      .EXPORT   _c
      .EXPORT   _f
      .EXPORT   _b
      .EXPORT   _g
      .SECTION  Pabc,CODE,ALIGN=2
_f:
      MOV.W     @_c:32,R0
      MOV.W     R0,@_a:32
      RTS
      .SECTION  P,CODE,ALIGN=2
_g:
      MOV.W     @_c:32,R0
      MOV.W     R0,@_b:32
      RTS
      .SECTION  Cabc,DATA,ALIGN=2
_c:
      .DATA.W    H'0001
      .SECTION  Babc,DATA,ALIGN=2
_a:
      .RES.W     1
      .SECTION  B,DATA,ALIGN=2
_b:
      .RES.W     1
      .END
```

If the **section=p=PROG** compile option is specified in the above example, f is allocated to PROGabc, and g to section PROG.

3.1.7 Register Save/Restore Code Control

#pragma regsave saves and restores the registers that are not used in functions at the entry and exit of the functions.

#pragma noregsave suppresses the code for saving and restoring the registers that are not used in functions.

Description Format:

```
#pragma regsave (<function name> [, <function name> ...])
#pragma noregsave (<function name> [, <function name> ...])
```

Note: **#pragma regsave** and **#pragma noregsave** must be declared before the functions used are declared.

Explanation:

#pragma regsave specifies the functions, for which the registers other than ER0 and ER1 (R0 and R1 in H8/300), that is, ER2 to ER6 (R2 to R6 in H8/300) when optimization is specified and ER2 to ER5 (R2 to R5 in H8/300) when optimization is not specified are all saved at the entry of the functions and restored at the exit of the functions, whether or not the register is used.

#pragma noregsave does not generate the code for saving and restoring the registers.

Example:

```
#pragma regsave (f)          /* Declared that the registers are saved and restored*/
f(){};
```

When the above C source program is compiled specifying **cpu=2600a**, the following object program will be generated:

```
      .CPU      2600A
      .EXPORT   _f
      .SECTION  P,CODE,ALIGN=2
_f:
      STM.L     (ER2-ER3),@-SP       ; Code for
      STM.L     (ER4-ER6),@-SP       ; saving registers
      LDM.L     @SP+,(ER4-ER6)       ; Code for
      LDM.L     @SP+,(ER2-ER3)       ; restoring registers
      RTS
      .END
```

Note:

Up to 63 functions can be declared in one **#pragma regsave/noregsave** statement.

3.2 Intrinsic Functions

The C compiler provides functions that cannot be written in C, such as system control instructions, as intrinsic functions. The following functions can be specified by intrinsic functions.

- Setting and referencing the condition code register
- Setting and referencing the extend register
- Special instructions (TRAPA, SLEEP, MOVFPE, MOVTPE, EEPMOV, and MAC)
- Rotation
- Operation reflecting the result in the condition code
- Decimal operation

Table 3-3 lists intrinsic functions.

Table 3-3 Intrinsic Functions

Item	Specification	Function
Condition code register	void set_imask_ccr(unsigned char)	Sets the interrupt mask
	unsigned char get_imask_ccr(void)	References the interrupt mask
	void set_ccr(unsigned char)	Sets the condition code register
	unsigned char get_ccr(void)	References the condition code register
	void and_ccr(unsigned char)	ANDs the condition code register
	void or_ccr(unsigned char)	ORs the condition code register
	void xor_ccr(unsigned char)	Exclusively ORs the condition code register
Extend register	void set_imask_exr(unsigned char)	Sets the interrupt mask
	unsigned char get_imask_exr(void)	References the interrupt mask
	void set_exr(unsigned char)	Sets the extend register
	unsigned char get_exr(void)	References the extend register
	void and_exr(unsigned char)	ANDs the extend register
	void or_exr(unsigned char)	ORs the extend register
	void xor_exr(unsigned char)	Exclusively ORs the extend register

Table 3-3 Intrinsic Functions (cont)

Item	Specification	Function
Special instructions	void trapa(unsigned int)	TRAPA instruction
	void sleep(void)	SLEEP instruction
	void movfpe(char*,char)	MOVFPE instruction
	void movtpe(char,char*)	MOVTPE instruction
	void tas(char*)	TAS instruction
	void eepmov(char*,char*,unsigned char) void eepmov(char*,char*,unsigned int)	EEPMOV instruction
	long mac(long,int*,int*,unsigned long) long macl(long, int*,int*,unsigned long, unsigned long)	MAC instruction
Rotation	char rotlc(int,char)	1-byte left rotation
	int rotlw(int,int)	2-byte left rotation
	long rotll(int,long)	4-byte left rotation
	char rotrc(int,char)	1-byte right rotation
	int rotrw(int,int)	2-byte right rotation
	long rotrl(int,long)	4-byte right rotation
Condition code operation	int ovfaddc(char,char,char*)	1-byte addition + reflecting the results in the condition code
	int ovfaddw(int,int,int*)	2-byte addition + reflecting the results in the condition code
	int ovfaddl(long,long,long*)	4-byte addition + reflecting the results in the condition code
	int ovfsubc(char,char,char*)	1-byte subtraction + reflecting the results in the condition code
	int ovfsubw(int,int,int*)	2-byte subtraction + reflecting the results in the condition code
	int ovfsubl(long,long,long*)	4-byte subtraction + reflecting the results in the condition code
	int ovfshalc(char,char*)	1-byte left shift + reflecting the results in the condition code
	int ovfshalw(int,int*)	2-byte left shift + reflecting the results in the condition code
	int ovfshall(long,long*)	4-byte left shift + reflecting the results in the condition code

Table 3-3 Intrinsic Functions (cont)

Item	Specification	Function
Condition code operation (cont)	int ovfnegc(char,char*)	1-byte negation + reflecting the results in the condition code
	int ovfnegw(int,int*)	2-byte negation + reflecting the results in the condition code
	int ovfnegl(long,long*)	4-byte negation + reflecting the results in the condition code
Decimal operation	void dadd(unsigned char,char*,char*, char*)	Decimal addition
	void dsub(unsigned char,char*,char*, char*)	Decimal subtraction

3.2.1 How to Use Intrinsic Functions

Intrinsic functions can be called in the same way as other functions except only after **#include <machine.h>** is declared.

3.2.2 Intrinsic Function Descriptions

Setting and Referencing Condition Code Register:

Example:

```
#include <machine.h>
main()
{
    set_imask_ccr(0)
         :
}
```

- set_imask_ccr

  — Calling procedure

  ```
  #include <machine.h>
  void set_imask_ccr(unsigned char mask);
  ```

  — Description

  Sets the mask value (0 or 1) to the interrupt mask bit (I) of the condition code register (CCR).

- get_imask_ccr

  — Calling procedure

  ```
  #include <machine.h>
  unsigned char get_imask_ccr(void)
  ```

  — Description

  References the mask value (0 or 1) in the interrupt mask bit (I) of the condition code register (CCR).

- set_ccr
  - Calling procedure

    ```
    #include <machine.h>
    void set_ccr(unsigned char ccr);
    ```

  - Description

    Sets the ccr value (8 bits) to the condition code register (CCR).

- get_ccr
  - Calling procedure

    ```
    #include <machine.h>
    unsigned char get_ccr(void);
    ```

  - Description

    References the condition code register (CCR).

- and_ccr
  - Calling procedure

    ```
    #include <machine.h>
    void and_ccr(unsigned char ccr);
    ```

  - Description

    ANDs the condition code register (CCR) with the ccr value and stores the results in the CCR.

- or_ccr
  - Calling procedure

    ```
    #include <machine.h>
    void or_ccr(unsigned char ccr);
    ```

  - Description

    ORs the condition code register (CCR) with the ccr value and stores the results in the CCR.

- xor_ccr
  - — Calling procedure

    ```
    #include <machine.h>
    void xor_ccr(unsigned char ccr);
    ```

  - — Description

    Exclusively ORs the condition code register (CCR) with the ccr value and stores the results in the CCR.

Setting and Referencing Extend Register:

Example:

```
#include <machine.h>
main()
{
    set_imask_exr(0)
         :
}
```

- set_imask_exr

  — Calling procedure

  ```
  #include <machine.h>
  void set_imask_exr(unsigned char mask);
  ```

  — Description

  Sets the mask value (0 to 7) to the interrupt mask bits (I2 to I0) of the extend register (EXR). This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- get_imask_exr

  — Calling procedure

  ```
  #include <machine.h>
  unsigned char get_imask_exr(void)
  ```

  — Description

  References the mask value (0 to 7) in the interrupt mask bits (I2 to I0) of the extend register (EXR). This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- set_exr
  - — Calling procedure

    ```
    #include <machine.h>
    void set_exr(unsigned char exr);
    ```

  - — Description

    Sets the exr value (8 bits) to the extend register (EXR). This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- get_exr
  - — Calling procedure

    ```
    #include <machine.h>
    unsigned char get_exr(void)
    ```

  - — Description

    References the extend register (EXR). This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- and_exr
  - — Calling procedure

    ```
    #include <machine.h>
    void and_exr(unsigned char exr);
    ```

  - — Description

    ANDs the extend register (EXR) with the exr value and stores the result in the EXR. This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- or_exr
  - — Calling procedure

    ```
    #include <machine.h>
    void or_exr(unsigned char exr);
    ```

  - — Description

    ORs the extend register (EXR) with the exr value and stores the result in the EXR. This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- xor_exr
  - — Calling procedure

    ```
    #include <machine.h>
    void xor_exr(unsigned char exr);
    ```

  - — Description

    Exclusively ORs the extend register (EXR) with the exr value and stores the result in the EXR. This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

Special Instructions:

Example:

```
#include <machine.h>
f()
{
          :
    trapa(0)
}
```

- trapa

  — Calling procedure

  ```
  #include <machine.h>
  void trapa(unsigned int trap_no);
  ```

  — Description

  Expanded to an unconditional trap instruction, TRAPA #trap_no. The trap_no must be a constant from 0 to 3. This function cannot be used in 300 CPU/operating mode.

- sleep

  — Calling procedure

  ```
  #include <machine.h>
  void sleep(void);
  ```

  — Description

  Expanded to the low power-consumption instruction, SLEEP.

- movfpe

  — Calling procedure

  ```
  #include <machine.h>
  void movfpe(char*addr,char data);
  ```

  — Description

  Expanded to an E clock-synchronous data transfer instruction, MOVFPE. Moves the contents of addr to data synchronously with the E clock.

- movtpe

  — Calling procedure

  ```
  #include <machine.h>
  void movtpe(char data,char* addr);
  ```

  — Description

  Expanded to an E clock-synchronous data transfer instruction, MOVTPE. Moves data to addr in the timing synchronous with the E clock.

- tas

  — Calling procedure

  ```
  #include <machine.h>
  void tas(char* addr);
  ```

  — Description

  Expanded to the test and set instruction, TAS. Compares the contents of addr with 0, reflects the result in the condition code register (CCR), and changes the highest-order bit of the addr contents to 1. This function can be used in 2600a, 2000a, 2600n, and 2000n CPU/operating modes.

- eepmov

  — Calling procedure

  ```
  #include <machine.h>
  void eepmov(char* dst, char* src, unsigned char size);
  ```
  or
  ```
  void eepmov(char* dst, char* src, unsigned int size);
  ```

  — Description

  Expanded to the block transfer instruction, EEPMOV. Transfers the bytes whose number is specified by size from the address specified by src to the address specified by dst.

  The size must be a constant. The maximum size is 255 in 300 CPU/operating mode and 65535 in other modes. However, when the size is in the range of 256 to 65535, this function is expanded to EEPMOV.W. In this case, precautions must be taken against NMI interrupts.

- mac and macl

  — Calling procedure

```
#include <machine.h>
long mac(long val,int* ptr1,int* ptr2,unsigned long count);
long macl(long val,int* ptr1,int* ptr2,unsigned long count,
                                          unsigned long mask);
```

  — Description

  Expanded to the multiply-and-accumulate instruction, MAC.

  The function mac sets val to the MAC register as the initial value, multiplies two bytes ptr1 and ptr2 with sign, adds the 4-byte result to the MAC register contents, and adds two to ptr1 and ptr2. This operation is repeated for the times specified by count.

  The function macl logically ANDs ptr2 with mask to use ptr2 repeatedly.

  These functions can be used in 2600a and 2600n CPU/operating modes.

  — Note

  The boundary of the table pointed to by ptr2 in the macl function must be aligned to a multiple of the mask value's complement. For example, in the following case, the linkage map must confirm that ptr2 is allocated to the address of a multiple of eight.

  Example:

```
#include <machine.h>
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
long l1,l2;
          :
l1=mac(100,ptr1,ptr2,4);
/* l1=100+0*9+1*8+2*7+3*6 */
l2=macl(100,ptr1,ptr2,4,-4);
/* l2=100+0*9+1*8+2*9+3*8 */
```

Rotation:

Example:

```
#include <machine.h>
int i,data;
f()
{
    i=rotlw(5,data);
}
```

- rotlc, rotlw, and rotll

  — Calling procedure

  ```
  #include <machine.h>
  char rotlc(int count,char data);
  int rotlw(int count,int data);
  long rotll(int count,long data);
  ```

  — Description

  The functions rotlc, rotlw, and rotll rotate 1-byte, 2-byte, and 4-byte data to the left according to the bits specified by count, and then return the results.

- rotrc, rotrw, and rotrl

  — Calling procedure

  ```
  #include <machine.h>
  char rotrc(int count,char data);
  int rotrw(int count,int data);
  long rotrl(int count,long data);
  ```

  — Description

  The functions rotrc, rotrw, and rotrl rotate 1-byte, 2-byte, and 4-byte data to the right by the bits specified by count, respectively, and return the results.

Operation Reflecting Results to Condition Code:

Example:

```
#include <machine.h>
int dst,src;
f()
{
if(ovfaddw(dst,src,0))
          :
else
          :
}
```

- ovfaddc, ovfaddw, and ovfaddl

  — Calling procedure

  ```
  #include <machine.h>
  int ovfaddc(char dst,char src,char* rst);
  int ovfaddw(int dst,int src,int* rst);
  int ovfaddl(long dst,long src,long* rst);
  ```

  — Description

  The functions ovfaddc, ovfaddw, and ovfaddl add 1-byte, 2-byte, and 4-byte data dst and src, respectively, store the results to the area specified by rst only when rst is not 0, return 0 when the results do not overflow and return a value other than 0 when they do overflow.

  These functions can be used only in the conditional statements such as the **if**, **do**, **while**, and **for** statements.

- ovfsubc, ovfsubw, and ovfsubl

  — Calling procedure

  ```
  #include <machine.h>
  int ovfsubc(char dst,char src,char* rst);
  int ovfsubw(int dst,int src,int* rst);
  int ovfsubl(long dst,long src,long* rst);
  ```

— Description

The functions ovfsubc, ovfsubw, and ovfsubl subtract 1-byte, 2-byte, and 4-byte data src from dst, respectively, store the results to the area specified by rst only when rst is not 0, and return 0 when the results do not overflow and a value other than 0 when they do overflow.

These functions can be used only in the conditional statements such as the if, do, while, and for statements.

- ovfshalc, ovfshalw, and ovfshall

  — Calling procedure

  ```
  #include <machine.h>
  int ovfshalc(char dst,char* rst);
  int ovfshalw(int dst,int* rst);
  int ovfshall(long dst,long* rst);
  ```

  — Description

  The functions ovfshalc, ovfshalw, and ovfshall arithmetically shift 1-byte, 2-byte, and 4-byte data dst to the left by one bit, respectively, store the results to the area specified by rst only when rst is not 0, and return 0 when the results do not overflow and a value other than 0 when they do overflow.

  These functions can be used only in the conditional statements such as the **if**, **do**, **while**, and **for** statements.

- ovfnegc, ovfnegw, and ovfnegl

  — Calling procedure

  ```
  #include <machine.h>
  int ovfnegc(char dst,char* rst);
  int ovfnegw(int dst,int* rst);
  int ovfnegl(long dst,long* rst);
  ```

  — Description

  The functions ovfnegc, ovfnegw, and ovfnegl calculate the 2's complements of 1-byte, 2-byte, and 4-byte data dst, respectively, store the results to the area specified by rst only when rst is not 0, and return 0 when the results do not overflow and a value other than 0 when they do overflow.

  These functions can be used only in the conditional statements such as the **if**, **do**, **while**, and **for** statements.

Decimal Operation:

Example:

```
#include <machine.h>
char ptr1[10]={0,1,2,3,4,5,6,7,8,9};
char ptr2[10]={0,1,2,3,4,5,6,7,8,9};
char rst[10];
              :
dadd((char)10,ptr1,ptr2,rst);
/* rst=0x0,0x2,0x4,0x6,0x8,0x10,0x12,0x14,0x16,0x18 */
```

- dadd
  - — Calling procedure

    ```
    #include <machine.h>
    void dadd(unsigned char size,char* ptr1,char* ptr2,char* rst);
    ```

  - — Description

    Adds size-byte data stored in the area starting from ptr1 to size-byte data stored in the area starting from ptr2 in decimal and stores the result to the size-byte area starting from rst. The size must be a constant from 1 to 255.

- dsub
  - — Calling procedure

    ```
    #include <machine.h>
    void dsub(unsigned char size,char* ptr1,char* ptr2,char* rst);
    ```

  - — Description

    Subtracts size-byte data stored in the area starting from ptr2 from size-byte data stored in the area starting from ptr1 in decimal and stores the result to the size-byte area starting from rst. The size must be a constant from 1 to 255.

Section 4 Notes on Programming

This section contains notes on coding programs for the C compiler and notes on developing programs at compilation or when debugging.

4.1 Coding Notes

Functions with float Parameters: For a function that declares float for parameters, either a prototype must be declared or parameters must be declared as double. Correct processing is not guaranteed if a function that has float parameters is called without a prototype declaration.

Example:

```
void f(float); .............(1)

g( )
{
    float a;
    f(a);
}

void
f(float x)
{
    :
}
```

Since function f has a float parameter, a prototype must be declared as shown at (1).

Program Whose Evaluation Order is Not Regulated: Correct execution of a process 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.

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. */

}
```

Note: When the **cpuexpand** option is specified, no overflow or underflow error message will be output.

Assignment to const Variables: Even if a variable is declared with const attribute, if assignment is done to a variable other than const converted from const attribute 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 file 2 */
   i=10;                /* is not an error.                            */
```

4.2 Notes on Program Development

This section contains notes on developing programs at compilation or when debugging.

4.2.1 Notes on CPU/Operating Mode Selection

(1) Unify CPU/operating modes specified at compilation. If an object program generated by different CPU/operating modes is linked, correct object program execution is not guaranteed.

(2) Specify the same CPU/operating mode for assembling and compiling. When using an H8/300 Series Assembler (Ver. 3.x) or an H8S,H8/300 Series Assembler (Ver. 1.0) to assemble an assembly program generated by the C compiler, specify the same CPU/operating mode with the one specified at compilation by the **cpu** option.

(3) Link the standard library that matches the CPU/operating mode at linkage. The C compiler supplies seven standard libraries, one for each of the CPU/operating mode. The library which matches the CPU/operating mode must be specified. If an unmatched library is specified, linkage is not guaranteed.

Refer to section 1.3, CPU/Operating Mode Selection, in Part I, Overview and Operation.

4.2.2 Notes on Bit Manipulation Instructions

The C compiler generates the BSET, BCLR, BNOT, BST, and BIST bit manipulation instructions. These instructions read data in byte units, manipulate bits, and write data in byte units. If a read operation is attempted for a write-only register, the CPU fetches value 0xFF regardless of the register contents. Therefore, if a bit manipulation instruction is executed for a write-only register, the bits other than the target bit may be changed. The following shows an example of bit manipulation for a write-only register.

Example:

Include file (300x.h) contents

```
struct S_p4ddr{
        unsigned char p7:1;
                :
        unsigned char p0:1;
};
union SS{
        unsigned char Schar;
        struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

C source program contents

```
#include "300x.h"
sub()
{

   unsigned char DDR=P4DDR.Schar;
   DDR &=~P0;
   P4DDR.Schar=DDR;

}
```

4.2.3 Troubleshooting

Table 4-1 Troubleshooting

Trouble	Check Points	Solution	References
Embedded assembly language is not indicated in the object program output by the C compiler.	Is **code=asmcode** specified with a compiler option ?	Specify **code=asmcode** at compilation.	3.1 in Part II, Programming
Error 314, cannot found section, is output at linkage	Is the section name which is output by the C compiler specified in capitals at start option of linkage editor?	Specify the correct section name.	2.1 in Part II, Programming, and 3.4 in Part I, Overview and Operation
Error 105, undefined external symbol, is output at linkage	Is underscore attached to the symbol in the assembly program if identifiers are mutually referenced by a C program and an assembly program ?	Reference symbols with the correct symbol names.	2.3.1 in Part II, Programming
	Is a C library function used in a C program?	Specify a standard library as the input library at linkage.	Standard library specification: 4.2.1 in Part II, Programming
	Does an undefined reference symbol identifier start with a $? (A run time routine in a standard library must be used.)		Execution routine in a standard library: 2.1 in Part III, System Installation,
	Is a standard I/O library in a C library function used?	Create low level interface routines for linking.	4 in Part III, System Installation
Error 108, relocation size overflow, is output at linkage	Is **abs8** or **abs16** option specified? Is **#pragma abs8** or **#pragma abs16** specified?	Allocate the 8-bit and 16-bit absolute address area correctly.	3.4 in Part I, Overview and Operation, and 3.1.1 in Part II, Programming
C source-level debugging cannot be performed	Is **debug** specified at compilation, assembly, and linkage?	Specify **debug** at compilation, assembly, and linkage	3.4 in Part I, Overview and Operation

PART III

SYSTEM INSTALLATION

Section 1 Overview

Part III describes how to install an object program generated by the C compiler on an H8S/2600, H8S/2000, H8/300H or H8/300 system. Before installation, memory allocation and execution environment for the object program must be specified.

- Memory allocation

  Stack area, heap area, each section of a C-compiler-generated object program must be allocated in ROM or RAM on a H8/300, H8/300H, H8S/2000, or H8S/2600 system.

- Execution environment setting for C-compiler-generated object program

  The execution environment can be specified by the register initialization processing, memory area initialization, and C program initiation processing. These must be written by assembly language.

  If C library functions for I/O operations are used, library must be initialized according to the execution environment specification. Specifically, if I/O operation function (stdio.h) and memory allocation function (std1ib.h) are used, the user must create low-level I/O routines and memory allocation routines appropriate to the user system.

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.

Section 2 Allocating Memory Areas

To install an object program generated by the C compiler on a system, each memory area size must be determined, then the areas must be allocated in memory.

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

2.1 Static Area Allocation

2.1.1 Data to be Allocated in Static Area

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

2.1.2 Static Area Size Calculation

The static area size is calculated by adding the size of C-compiler-generated object program and that of library functions used by the C program. After object program linkage, the static area size can be determined from each section size including library size output on a linkage map listing. Before object program linkage, the static area size can be approximately determined from the section size information on a compile listing. Figure 2-1 shows an example of section size information.

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                          0x00000080 Byte(s)
CONSTANT SECTION(C):                          0x00000004 Byte(s)
DATA     SECTION(D):                          0x00000004 Byte(s)
BSS      SECTION(B):                          0x00000004 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000080 Byte(s)
TOTAL CONSTANT SECTION: 0x00000004 Byte(s)
TOTAL DATA     SECTION: 0x00000004 Byte(s)
TOTAL BSS      SECTION: 0x00000004 Byte(s)

    TOTAL PROGRAM SIZE: 0x0000008C Byte(s)
```

Figure 2-1 Section Size Information

If the standard library is not used, the static area size can be calculated by adding memory area size used by library functions and memory area size used by sections to the size shown in section size information. The standard library includes C library functions based on C language specifications and arithmetic operation routines required for C program execution. Accordingly, the standard library may be required even if library functions are not used in the C source program.

Note: The standard library supplied by the C compiler includes C library functions (based on C language specification), and arithmetic routines (run time routines) which are 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 user can see the run time routine names used by the C programs through the symbol allocation information in compiler listing.

The following shows the example of C program and symbol allocation information.

C program

```
long a,b;
main()
{
      a *= b;
}
```

Symbol allocation information output by C compiler

```
******* STACK FRAME INFORMATION *******

FILE NAME: main.c

Function (File main.c    , Line    2): main

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000000 Byte(s)
Total Frame Size        : 0x00000008 Byte(s)

Used Runtime Library Name
$MULL$3                          ;Run time routine
```

2.1.3 ROM and RAM Allocation

When installing a program to memory, static areas must be allocated 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, 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 is allocated to ROM 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, only a ROM area needs to be allocated.

2.1.5 Example: Memory Area Allocation and Address Specification at Program Linkage

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

Figure 2-2 shows an example of allocating static areas in H8S/2600 advanced mode.

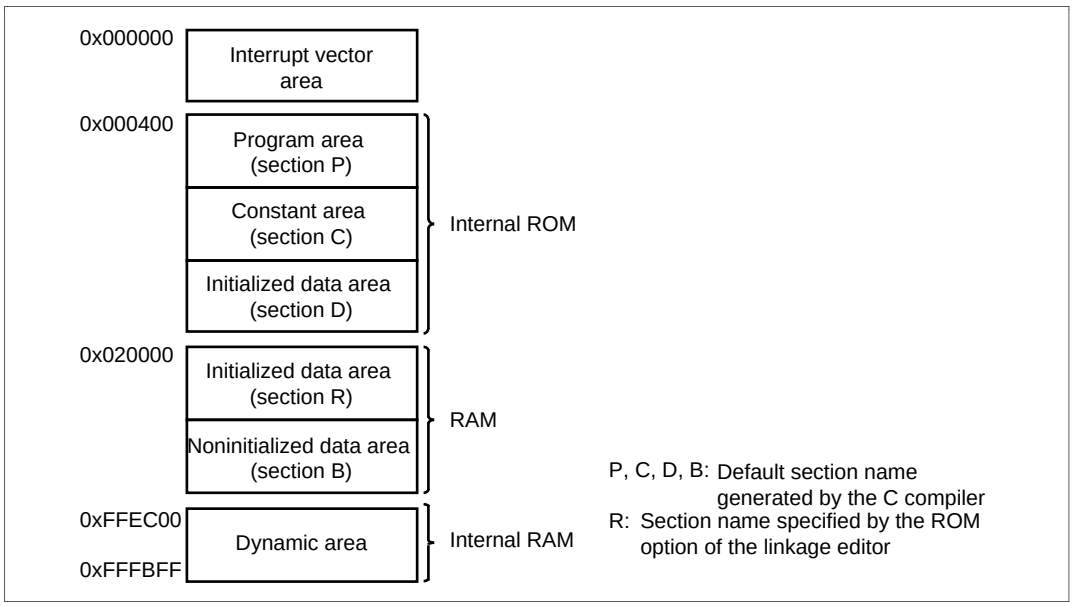


Figure 2-2 Static Area Allocation

Specify the following subcommand when allocating the static area as shown in figure 2-2.

```
        :
ROMΔ(D,R)                          --------(1)
STARTΔP,C,D(400),R,B(20000)  --------(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, relocate the symbol to the address 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 0x20000.

Note: The linkage editor of Version. 5.0 or upper supports subcommand ROM.

2.2 Dynamic Area Allocation

2.2.1 Dynamic Areas

Two types of dynamic areas are used:

- Stack area
- 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 size of stack used by each function can be determined from the symbol allocation information (Total Frame Size) of the compiler listings.

  Example:

  The following shows the symbol allocation information and stack size calculation in a C program. In this example, the H8S/2600 advanced mode is specified.

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

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

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

```
******* STACK FRAME INFORMATION ********
FILE NAME: m0280.c

Function (File m0280.c   , Line    3): h

  Parameter Allocation
    a                                0xfffffff7 saved from R0L
    b                             REG ER5       saved from ER1
    c                                0x00000008

  Level 1  (File m0280.c   ,Line   4) Automatic/Register Variable Allocation
    d                                0xfffffff2

  Level 2  (File m0280.c   ,Line   9) Automatic/Register Variable Allocation
    i                             REG R4

Parameter Area Size      : 0x00000008 Byte(s)
Linkage Area Size        : 0x00000008 Byte(s)
Local Variable Size      : 0x00000006 Byte(s)
Temporary Size           : 0x00000000 Byte(s)
Register Save Area Size  : 0x00000008 Byte(s)
Total Frame Size         : 0x0000001e Byte(s)
```

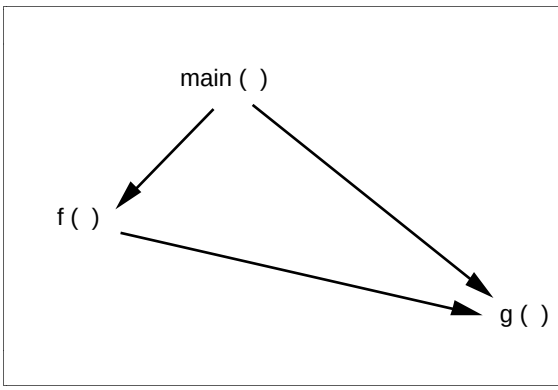
The size of stack used by a function is shown by Total Frame Size 0x1e, that is, 30 bytes.

- Stack size calculation

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

  Example:

  Figure 2-3 illustrates the function call nestings and stack size.



Function Name	Stack Size (Bytes)
main	30
f	32
g	24

Figure 2-3 Nested Function Calls and Stack Size

If function g is called via function f, stack area size is calculated according to the formula listed in table 2-1.

Table 2-1 Stack Size Calculation Example

Function Calling Route	Total Stack Size
main (30) —> f(32) —> g(24)	86 bytes (max)
main (30) —> g(24)	54 bytes

As can be seen from table 2-1, the maximum size of stack area required for longest function calling route should be determined (86 bytes in this example) and this size of memory should be allocated in RAM.

When using standard library functions, the stack frame sizes for library functions must also be accounted for. Refer to the Standard Library 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 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 2 or 4 bytes must be summed because a 2-byte (**cpu = 2600n**, **cpu = 2000n**, **cpu = 300hn**, **cpu = 300**) or 4-byte (**cpu = 2600a**, **cpu = 2000a**, **cpu = 300ha**) management area is used every time a memory management library function allocates an area. The C compiler manages the heap area in 1028-byte units (**cpu = 2600n**, **cpu = 2000n**, **cpu = 300hn**, **cpu = 300**) or 1032-byte units (**cpu = 2600a**, **cpu = 2000a**, **cpu = 300ha**). The heap area size (HEAPSIZE) can be calculated as follows:

HEAPSIZE = 1028 × n or HEAPSIZE = 1032 x n ($n \geq 1$)
(Areas allocated by memory management library) + management area size ≤ HEAPSIZE

An input/output 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:

- **cpu = 2600n**, **cpu = 2000n**, **cpu = 300hn**, **cpu = 300**

  Size of area allocated in an I/O: 514 bytes × (maximum number of simultaneously open files)

- **cpu = 2600a**, **cpu = 2000a**, **cpu = 300ha**

  Size of area allocated in an I/O: 516 bytes × (maximum number of simultaneously open files)

Note: Areas released by the free function, 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:

— If possible, allocate the largest area first after program execution is started.

— If possible, specify data area size to be reused as a 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 SP in the initial specification (INIT) at program initiation. 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 3.2, Initialization (INIT), and section 4.6, Creating Low-Level Interface Routine, 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 system specification because the C program execution environment differs depending on the user systems. 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 for details on program execution.

Figure 3-1 shows an example of program configuration.

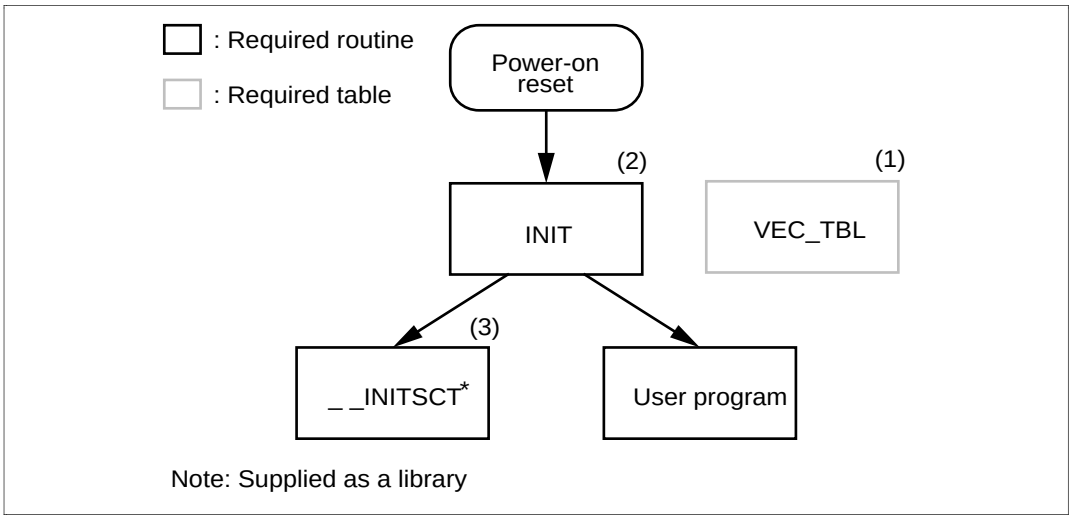


Figure 3-1 Program Configuration (No C Library Function is Used)

Each routine is described below.

(1) Vector table setting (VEC_TBL)

Sets vector table so as to initiate register initialization program INIT by power-on reset.

(2) Initialization (INIT)

Initializes registers and sequentially calls initialization routines.

(3) Section initialization (_ _INITSCT)

Clears the non-initialized data area with zeros and copies the initialized data area in ROM to RAM. This routine is supplied as a standard library function _ _INITSCT.

How to create the above routines are described below.

3.1 Vector Table Setting (VEC_TBL)

To call register initialization routine INIT at power-on reset, set the start address of function INIT to address 0 in the vector table. When the user system executes interrupt handlings or memory-indirect function calls, interrupt vector and address table settings are also performed in the VEC_TBL routine. The VEC_TBL coding examples in C and assembly language are shown below.

Examples:

(C program example)

```
extern void INIT(void);  /* #pragma interrupt is not declared in this file     */
extern void IRQ0(void);  /* because the interrupt function references addresses*/
#pragma section vect1    /* Outputs vec_table to Cvect1 section                */
                         /* Specify that Cvect0 section is allocated to        */
                         /*             address 0 with option start at linkage */
const void (*const vec_table1[])(void)={INIT};
#pragma section vect2    /* Outputs vec_table2 to Cvect2 section               */
                         /* Specify that Cvect2 section is allocated to        */
                         /* the specified address with option start at linkage */
const void(*const vec_table2[])(void)={IRQ0};
                         /* Allocates the address table created by option      */
                         /*    indirect or #pragma indirect to an address that */
                         /*                                        is not used */
```

(Assembly program examples)

- 2600n, 2000n, 300hn, and 300 CPU/operating mode

```
       .EXPORT  $IRQ0
       .IMPORT  _INIT
       .IMPORT  _IRQ0
       .SECTION Cvect1,DATA,LOCATE=H'0000 ;Allocates Cvect1 section to address 0
       .DATA.W  _INIT                     ;Allocates the start of _INIT to
                                          ;addresses 0 to 1
       .SECTION Cvect2,DATA,LOCATE=H'0040 ;Allocates Cvect2 section to address 0x40
$IRQ0: .DATA.W  _IRQ0                     ;Allocates the start of _IRQ0 to
       .END                               ;addresses 0x40 to 0x41
```

- 2600a, 2000a, and 300ha CPU/operating mode

```
       .EXPORT  $IRQ0
       .IMPORT  _INIT
       .IMPORT  _IRQ0
       .SECTION Cvect1,DATA,LOCATE=H'0000 ;Allocates Cvect1 section to address 0
       .DATA.L  _INIT                     ;Allocates the start of _INIT to
                                          ;addresses 0 to 3
       .SECTION Cvect2,DATA,LOCATE=H'0080 ;Allocates Cvect2 section to address 0x80
$IRQ0: .DATA.L  _IRQ0                     ;Allocates the start of _IRQ0 to
       .END                               ;addresses 0x80 to 0x83
```

3.2 Initialization (INIT)

INIT initializes registers, calls initialization routine sequentially, and then calls main function. The coding examples of this routine are shown below.

Examples:

- 2600n, 2000n, 300hn, and 300 CPU/operating mode

```
#include <machine.h>
#pragma noregsave INIT
void main(void)
void _INITSCT(void);
void INIT(void)
#pragma asm
    MOV.W #H'FFFE,R7     ;Specifies the stack address before INIT function starts
#pragma endasm
{
    set_imask_ccr(0);    /*Sets the interrupt mask bit to 0 */
    _INITSCT();          /*Calls section initialization routine _INITSCT */
    main();              /*Calls main function */
    sleep();             /*Expands to sleep instruction */
}
```

- 2600a, 2000a, and 300ha CPU/operating mode

```
#include <machine.h>
#pragma noregsave INIT
void main(void)
void _INITSCT(void);
void INIT(void)
#pragma asm
    MOV.L #H'FFFBFE,SP   ;Specifies the stack address before INIT function starts
#pragma endasm
{
    set_imask_ccr(0);    /*Sets the interrupt mask bit to 0 */
    _INITSCT();          /*Calls section initialization routine _INITSCT */
    main();              /*Calls main function */
    sleep();             /*Expands to sleep instruction */
}
```

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. Since this routine is supplied as a standard, this routine can be executed by only calling the _ _INITSCT function in the initialization routine as described in section 3.2, Initialization (INIT). However, note that the following conditions must be satisfied to execute the _ _INITSCT function.

1. Section names for the non-initialized data area and initialized data area must be B and D, respectively. (Default section names B and D are created by the C compiler when no name is specified by the compiler option **section** or **#pragma section**.)

2. No **abs8** or **abs16** option, or **#pragma abs8** or **#pragma abs16** must be specified.

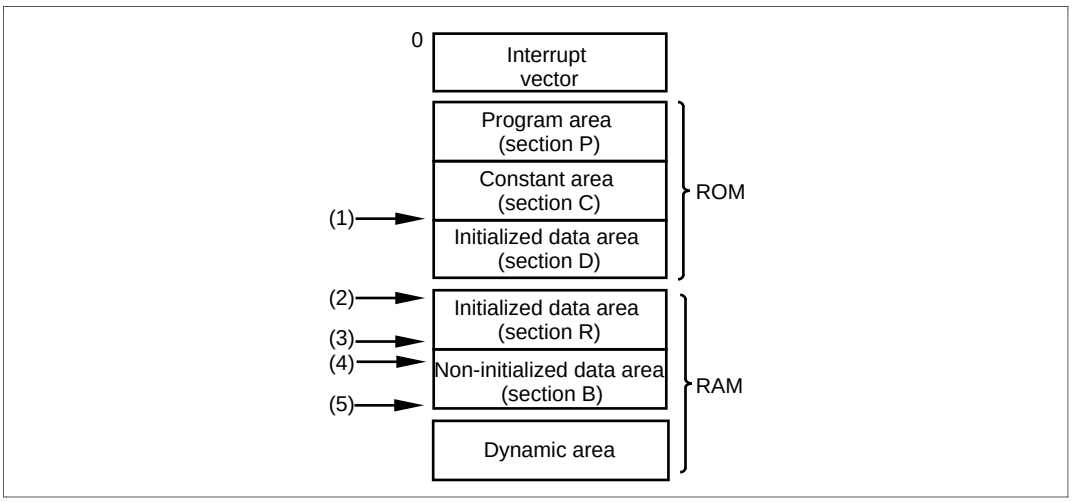
3. R is assumed to be section name of initialized data area in RAM. Accordingly, **ROM=(D,R)** option or subcommand must be specified by the linkage editor during load module creation.

4. Linkage editor of version 5.0 or higher must be used.

When the standard library function _ _INITSCT is not used, create _ _INITSCT using the following procedure.

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



To obtain the above addresses, create the following assembly programs and link them together.

• H8S/2600, H8S/2000, and H8/300H in normal mode, and H8/300

```
            .SECTION  D,DATA,ALIGN=2
            .SECTION  R,DATA,ALIGN=2
            .SECTION  B,DATA,ALIGN=2
            .SECTION  C,DATA,ALIGN=2

_ _D_ROM    .DATA.W   (STARTOF D)                  ;(1) Start address of section D
_ _D_BGN    .DATA.W   (STARTOF R) *1               ;(2) Start address of section R
_ _D_END    .DATA.W   (STARTOF R) + (SIZEOF R)*2   ;(3) End address of section R

_ _B_BGN    .DATA.W   (STARTOF B)                  ;(4) Start address of section B
_ _B_END    .DATA.W   (STARTOF B) + (SIZEOF B)     ;(5) End address of section B

            .EXPORT   _ _D_ROM
            .EXPORT   _ _D_BGN
            .EXPORT   _ _D_END
            .EXPORT   _ _B_BGN
            .EXPORT   _ _B_END
            .END
```

• H8S/2600, H8S/2000, and H8/300H in advanced mode

```
            .SECTION  D,DATA,ALIGN=2
            .SECTION  R,DATA,ALIGN=2
            .SECTION  B,DATA,ALIGN=2
            .SECTION  C,DATA,ALIGN=2

_ _D_ROM    .DATA.L   (STARTOF D)                  ;(1) Start address of section D
_ _D_BGN    .DATA.L   (STARTOF R) *1               ;(2) Start address of section R
_ _D_END    .DATA.L   (STARTOF R) + (SIZEOF R)*2   ;(3) End address of section R

_ _B_BGN    .DATA.L   (STARTOF B)                  ;(4) Start address of section B
_ _B_END    .DATA.L   (STARTOF B) + (SIZEOF B)     ;(5) End address of section B

            .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.

2. Section name R must be the section name in RAM area specified with the ROM option at linkage.

3. Underlined parts *1 and *2 are supported in the linkage editor with version 5.0 or higher. If a linkage editor with version lower than 5.0 is used, addresses actually allocated must be specified.

   Example: Initialized data area is allocated from address 0x8000 in RAM.

```
_ _D_BGN    .DATA.L    H'8000 *1                ;(2)
_ _D_END    .DATA.L    H'8000 + (SIZEOF D) *2   ;(3)
```

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

```
extern char *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;
extern void _INITSCT( );

void _INITSCT( )
{
       char *p, *q ;

        /* Non-initialized 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 ;

}
```

Section 4 Setting the C Library Function Execution Environment

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

This section describes how to set C program execution environment when C library functions are used.

Figure 4-1 shows a program configuration when C library functions are used.

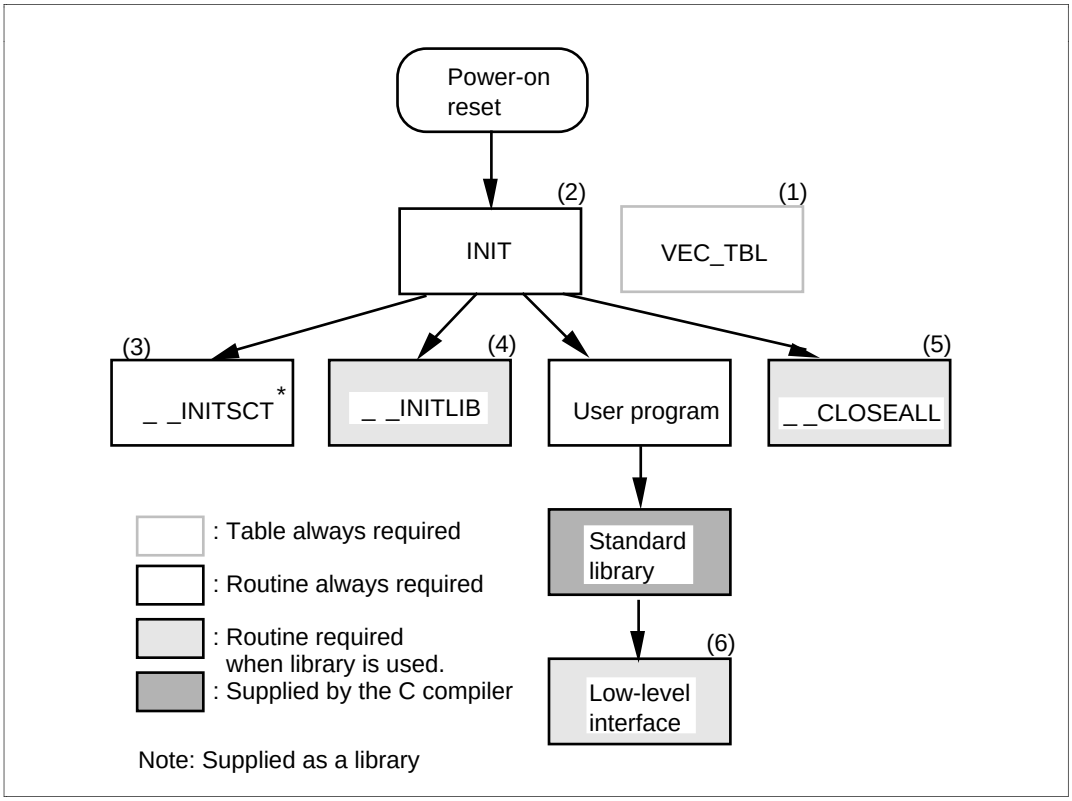


Figure 4-1 Program Configuration When C Library Function Is Used

Each routine required to execute library functions as follows.

(1) Setting vector table (VEC_TBL)

Sets vector table to initiate register initialization program (INIT) at power-on reset.

(2) Initialization (INIT)

Initializes registers and sequentially calls the initialization routines. INIT is written in assembly language so that it can access hardware register directly.

(3) Initializing sections (_ _INITSCT)

Clears non-initialized data area with zeros and copies the initialized data area in ROM to RAM. This routine is supplied as a standard library function.

(4) Initializing C library functions (_ _INITLIB)

Initializes C library functions required to be initialized and prepares standard I/O functions.

(5) Closing files (_ _CLOSEALL)

Closes all files with open status.

(6) 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.

Note: When using the C library functions that terminates program execution such as exit, onexit, or abort, the C library function must be created according to the user system. For details, refer to appendix D, Termination Processing Function Example.

In addition, when using C library function assert macro, the abort function must be created.

4.1 Setting Vector Table (VEC_TBL)

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

4.2 Initialization (INIT)

Initializes registers and sequentially calls the initialization routine _ _INITLIB and file closing routine _ _CLOSEALL. The coding example of INIT is shown below.

- 2600n, 2000n, 300hn, or 300 CPU/operating mode

```
#include <machine.h>
#pragma noregsave(INIT)      /* Suppresses register save/restore code */
void main(void);
void _INITSCT(void);
void _INITLIB(void);
void _CLOSEALL(void);
void INIT(void);
#pragma asm
    MOV.W #H'FFFE,R7         ;Specifies stack address before INIT function starts
#pragma endasm
{
    set_imask_ccr(0);        /* Sets interrupt mask bit to 0 */
    _INITSCT( );             /* Calls section initialization routine _INITSCT */
    _INITLIB( );             /* Calls library initialization routine _INITLIB */
    main( );                 /* Calls main function _INITSCT */
    _CLOSEALL( );            /* Calls file closing routine _CLOSEALL */
    sleep( );                /* Expands to sleep instruction */
}
```

- 2600a, 2000a, or 300ha CPU/operating mode

```
#include <machine.h>
#pragma noregsave(INIT)      /* Suppresses register save/restore code */
void main(void);
void _INITSCT(void);
void _INITLIB(void);
void _CLOSEALL(void);
void INIT(void);
#pragma asm
    MOV.L #H'FFFBFE,SP       ;Specifies stack address before INIT function starts
#pragma endasm
{
    set_imask_ccr(0);        /* Sets interrupt mask bit to 0 */
    _INITSCT( );             /* Calls section initialization routine _INITSCT */
    _INITLIB( );             /* Calls library initialization routine _INITLIB */
    main( );                 /* Calls main function _INITSCT */
    _CLOSEALL( );            /* Calls file closing routine _CLOSEALL */
    sleep( );                /* Expands to sleep instruction */
}
```

4.3 Initializing Sections (_ _INITSCT)

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

4.4 Initializing C Library Functions (_ _INITLIB)

Initializes related C library functions. 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.

(3) The user low-level interface routine must be initialized according to the user low-level initialization routine specification if required.

(4) When using the rand and strtok functions, library functions other than I/O must be initialized.

Library function initialization program example is shown below.

Example:

```
#include <stdlib.h>

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

void _INITLIB(void)       /* Deletes an underscore from the assembly routine */
                          /* symbol name */
{
    errno=0;              /* Initialization for all library functions */
    _INIT_LOWLEVEL( );    /* Calls low-level interface initialization routine */
    _INIT_IOLIB( );       /* Calls standard I/O initialization routine */
    _INIT_OTHERLIB( );    /* Calls other initialization routine */
}
```

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

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

The standard I/O library function initialization routine 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.

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 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 4-2 FILE-Type Data

4.4.2 Creating Other Library Function Initialization Routine (_INIT_OTHERLIB)

The following gives an example of the routine initializing standard library functions other than standard I/O.

Example:

```
#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 may be lost.

When an program is installed in a device, the program is not terminated normally. 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( )  /*Deletes an underline 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 opened files*/

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

4.6 Creating Low-Level Interface Routines

Low-level interface routines must be created for C programs that use the standard input/output or memory management library functions. Table 4-1 shows the low-level interface routines used by standard library functions.

Table 4-1 Low-Level Interface Routines

Name	Explanation
open	Opens files
close	Closes files
read	Reads data from a file
write	Writes data to a file
lseek	Sets the file read/write address for data
sbrk	Allocates a memory area

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 H8S, H8/300-series simulator debugger.

Note: The open, close, read, write, lseek, and sbrk are reserved words for low-level interface routines. Do not use these words in C programs.

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 uniquely recognized by the open routine.)
- Information such as the size and 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 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.

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)	

(a) open routine				
Purpose	Opens a file			

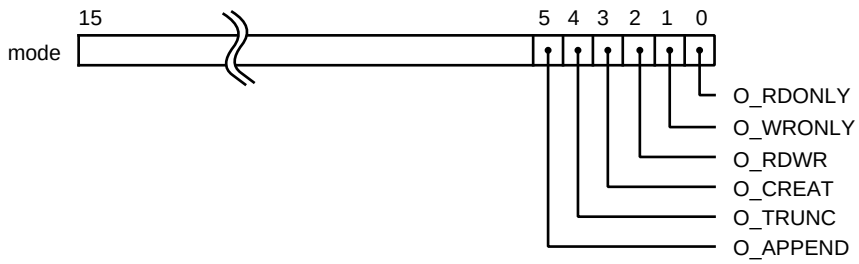
Interface	`int open (char *name,`
	`int mode,`
	`int flg);`			

Parameters	No.	Name	Type	Meaning
	1	name	Pointer to char	String literal indicating a file name
	2	mode	int	Processing specification
	3	flg	int	Processing specification (always 0777)
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 below:



O_RDONLY (bit 0): If this bit is 1, the file becomes read only.

O_WRONLY (bit 1): If this bit is 1, the file becomes write only.

O_RDWR (bit 2): If this bit is 1, the file becomes read/write.

O_CREAT (bit 3): If this bit is 1 and the file indicated by the file name does not exist, a new file is created.

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.

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.

(b) 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.

The area of memory allocated by the open routine for file management information is freed, 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.

(c) 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.

(d) 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 must be 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.

(e) 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:

- Base = 0
  The second parameter gives the new offset relative to the beginning of the file.
- Base = 1
  The second parameter is added to the current position to give the new offset.
- 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 Base=0 or Base=1, 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.

(f) sbrk routine				
Purpose	Allocates a memory area			
Interface	`char *sbrk(int size);`			
Parameters	No.	Name	Type	Meaning
	1	size	int	Size of the area to be allocated
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 Output by the Compiler

The C compiler checks C source programs and options specified at initiation for a variety of possible errors. This section explains the format and meaning of error messages that may be generated during compile time, and gives the appropriate programmer response.

1.1 Error Message Format

Error messages are output to the listing file and the standard output file for MS-DOS systems or the standard error output file for UNIX systems. Figures 1-1 and 1-2 show the format for error messages that are output to the standard output file and standard error output file. For details on the files used to list compiled program information, refer to section 3.5 (3), Error Information, in Part II, Overview and Operation.

```
SAMPLE. C      23      2011   (E)    LINE TOO LONG
   ①           ②       ③      ④          ⑤
```

Figure 1-1 Error Messages Format (MS-DOS Systems)

```
"sample. c"   line 23 :     2011   (E)    LINE TOO LONG
    ①      ②  ③     ④          ⑤
```

Figure 1-2 Error Messages Format (UNIX Systems)

Explanation:

① File name
This gives the name of the source program file in which the error was detected. In this example, the file specification is **sample.c**.

② Line number
This gives the line number where the error was detected. In this example, an error was detected on line 23.

③ Error number
This number is unique to the error message. Section 1.4, List of Error Messages, explains error messages in the order of their error numbers. See this section for details about the errors and appropriate programmer responses.

④ Message level
This indicates how serious the error was. There are four error levels: I, W, E, and F. See section 1.3, Message Levels, for details.

⑤ Message text
This gives a description of the error. In this example, line 23 in file sample.c is too long.

1.2 Error Location

The file name and line number indicate the position of the error in the source program. These items are not output if the error is not in the source program.

(1) File name
The file name indicates the source program file in which the error was detected.

Note: If the file name is too long, only the first 10 characters are given in the error message output to the listing file. This means that error positions cannot be identified in files which have different directories but the same file names, or at least the same first 10 characters.

(2) Line number
The line number indicates where in the source program the error was detected.

1.3 Message Levels

Error messages are classified into the following four levels according to their severity:

① (I) .... Information level

② (W) ... ... Warning level

③ (E) ... ... Error level

④ (F) .... Fatal level

Information-level errors indicate unfavorableness in program coding, but no violation of the language specifications. Warning-level errors indicate violation of the language specifications. The C compiler can recover from a warning-level error. Error-level errors indicate violation of the language specification. Fatal-level errors indicate that the limits of the C compiler have been exceeded.

Note: In addition to the above errors, internal errors may occur. They have the following message format: INTERNAL ERROR, and error-number: 4000 to 4999. These error messages are

output if an error occurs in the C compiler. If such an error occurs, contact your local Hitachi dealer.

If an error occurs, the action taken by the C compiler depends on the error level. (The effect of an internal error is the same as the effect of a fatal-level error.)

(1) Error Message Output
If no **message** option is used, an error message is output for a warning-level, error-level, or fatal-level error. To output a message for an information-level error, specify the **message** option.

(2) Object Program Output
If only information-level or warning-level errors occur during compile time, the C compiler performs error recovery and outputs the object program. In this case, check that the error recovery performed by the C compiler matches with the user's intention by referencing the list of error messages.

The object program is not output if an error-level or fatal-level error is detected.

(3) Continuing Compile Processing
The C compiler continues processing in order to detect any additional errors in the source program when a warning-level error or an error-level error is encountered. The C compiler terminates processing immediately if a fatal-level error is detected.

Table 1-1 shows C compiler action for each level of errors.

Table 1-1 C Compiler Action and Programmer Response for Each Level of Error

No.	Error Level	Symbol	Error Number	Object Program Output	Processing Continues	Programmer Response
1	Informa-tion	(I)	0000 to 0999	Yes	Yes	Check the list of error messages to decide whether error recovery performed by the C compiler is correct. If required, modify and recompile the source program.
2	Warning	(W)	1000 to 1999	Yes	Yes	Check the list of error messages to decide whether error recovery performed by the C compiler is correct. If required, modify and recompile the source program.
3	Error	(E)	2000 to 2999	No	Yes	Correct the error and recompile the source program.
4	Fatal	(F)	3000 to 3999	No	No	Correct the error and recompile the source program.

1.4 List of Error Messages

This section gives lists of error messages in order of error number. A list of error messages is provided for each level of errors.

Example:

Error No.	Message	Explanation
① 2226	② SCALAR REQUIRED	③ The binary operator && or || is used in a non-scalar expression. ④ S: Assumes that the result type is **int** and continues processing. ⑤ P: Use scalar expressions as operands.

① Error Number

② Error Message
This message is sent to the standard output file or standard error output file. It is also output to the listing file unless the **nolist** option is specified.

③ Explanation
This gives more details about the error.

④ System Action
This indicates the reaction of the C compiler to the error.

⑤ Programmer Response
This indicates to the programmer how to resolve the error.

(1) Information-Level Messages

Error No.	Message	Explanation
0001	CHARACTER COMBINATION "/*" IN COMMENT	A comment has character string "/*." P: Check that "/*" and "*/" in the comment are correctly specified.
0002	NO DECLARATION	There is a declaration that has no declarator. P: Check that the declaration has a declarator.
0003	UNREACHABLE STATEMENT	There is a statement that is not executed. P: Check that the statement is not required to be executed.
0004	CONSTANT AS CONDITION	A constant expression is specified as an expression indicating the condition of the **if** or **switch** statement. P: Check that the **if** and **switch** statements are used correctly.
0006	CONVERSION IN ARGUMENT	The expression of a function parameter is converted to the parameter type specified in the prototype declaration. P: Check that the correct expression type of the parameter is specified.
0008	CONVERSION IN RETURN	A return statement is converted to the value type returned by the function. P: Check that the correct return statement is specified.
0010	ELIMINATION OF NEEDLESS EXPRESSION	The left part of the assignment expression is not used. P: Check that the assignment expression can be eliminated.
0011	USED BEFORE SET SYMBOL: **"variable name"**	A local variable is referred in which no value is set. P: Check that the local variable requires no value.
0015	NO RETURN VALUE	A return statement does not return values or there is no return statement in the function that returns a type other than the **void** type. P: Check that the function returns values.

Error No.	Message	Explanation
0100	FUNCTION NOT OPTIMIZED: **"function name"**	Optimization processing cannot be applied to a function. P: A function with a large program cannot be optimized. However, the function can be optimized by dividing the function.
0200	NO PROTOTYPE FUNCTION	There is no prototype declaration of a called function. P: Check that the function type and the type and number of parameters are correct.

(2) Warning-Level Messages

Error No.	Message	Explanation
1000	ILLEGAL POINTER ASSIGNMENT	The pointer is assigned to a pointer with a different data type. S: Sets the right hand side to the internal representation of the left hand side pointer. The result type is the same as the data type of the left pointer. P: Use the **cast** operator to specify explicit type conversion.
1001	ILLEGAL COMPARISON	The operands of the binary operator == or != are a pointer and an integer other than 0. S: Selects an internal representation for the operands. P: Specify the correct type for the operands.
1002	ILLEGAL POINTER REQUIRED	The operands of the binary operator ==, !=, >, <, >=, or <= are pointers assigned to different types. S: Assumes that the operands are pointers assigned to the same type. P: Use the **cast** operator so that the same operand type will be used.
1005	UNDEFINED ESCAPE SEQUENCE	An undefined escape sequence (a character following a backslash) is used in a character constant or string literal. S: Ignores the backslash. P: Remove the backslash or specify the correct escape sequence.
1007	LONG CHARACTER CONSTANT	The length of a character constant is 2 characters. S: Uses the specified characters. P: Check that the correct character constant is specified.
1008	IDENTIFIER TOO LONG	An identifier's length exceeds 31 characters. S: Uses the first 31 characters and ignores the rest. P: Use identifiers with 31 or less characters.

Error No.	Message	Explanation
1010	CHARACTER CONSTANT TOO LONG	The length of a character constant exceeds two characters. S: Uses the first two characters and ignores the rest. P: Use character constant with two or less characters.
1012	FLOATING POINT CONSTANT OVERFLOW	The value of a floating-point constant exceeds the limit. S: Assumes the internally represented value corresponding to +∞ or –∞ depending on the sign of the result. P: Specify floating-point constants within their limits.
1013	INTEGER CONSTANT OVERFLOW	The value of **unsigned long** integer constant exceeds the limit. S: Ignores the overflow and uses the remaining bits. P: Specify integer constants within their limits.
1014	ESCAPE SEQUENCE OVERFLOW	The value of an escape sequence indicating a bit pattern in a character constant or string exceeds 255. S: Uses the low order byte. P: Change the value of the escape sequence to 255 or lower.
1015	FLOATING POINT CONSTANT UNDERFLOW	The absolute value of a floating-point constant is less than the lower limit. S: Assumes 0.0 as the value of the constant. P: Change the value of the constant to 0.0 or specify a constant whose value can be represented.

Error No.	Message	Explanation
1016	ARGUMENT MISMATCH	The data type assigned to a pointer specified as a parameter in a prototype declaration differs from the data type assigned to a pointer used as the corresponding parameter in a function call. S: Uses the internal representation of the pointer used for the function call parameter. P: Use the **cast** operator for the function call parameter to convert the parameter to the type specified in the prototype declaration.
1017	RETURN TYPE MISMATCH	The function return type and the expression type in a **return** statement are pointers, however, the data types assigned to these pointers are different. S: Uses the internal representation of the pointer specified in the **return** statement expression. P: Use the **cast** operator on the expression specified in the **return** statement expression to obtain the function return type.

Error No.	Message	Explanation
1018	TYPE MISMATCH	The same name variables or functions in the **extern** storage class are declared in different scopes, however, the data type of the variables or functions are different. S: The declared variable or function is used in the visible scope. However, if the variable or function is linked with another file, the data type is interpreted depending on the declaration as follows: (1) If it is declared for definition: The declared and defined data type are effective. (2) If it is not declared for definition: – If the current declaration is in the function, the first declared type is used. – If the current declaration is outside the function, the current declared type is used. P: Always ensure that variables or functions having the same name in the **extern** storage class have the same data type.
1019	ILLEGAL CONSTANT EXPRESSION	The operands of the relational operator <, >, <=, or >= in a constant expression are pointers to different data types. S: Assumes 0 as the result value. P: Use an expression other than a constant expression to obtain the correct result.
1020	ILLEGAL CONSTANT EXPRESSION	The operands of the binary operator – in a constant expression are pointers to different data types. S: Assumes 0 as the result value. P: Use an expression other than a constant expression to obtain the correct result.
1200	DIVISION BY FLOATING POINT ZERO	Division by the floating-point number 0.0 is carried out in the evaluation of a constant expression. S: Assumes the internal representation of the value corresponding to +∞ or –∞ depending on the sign of the operands. P: Specify the correct constant expression.

Error No.	Message	Explanation
1201	INEFFECTIVE FLOATING POINT OPERATION	Invalid floating-point operations such as ∞ – ∞ or 0.0/0.0 are carried out in a constant expression. S: Assumes the internal representation of not-a-number to indicate the result of an ineffective operation. P: Correct the constant expression.
1300	COMMAND PARAMETER SPECIFIED TWICE	The same C compiler option is specified more than once. S: Uses the last specified compiler option. P: Check that options are specified correctly.
1301	TOO MANY DEFINE OPTIONS	The number of macro names specified as suboptions in the **define** option exceeds 16. S: Uses the first 16 suboptions. P: Define the 17th and subsequent macro names using **#define** directives at the beginning of the source program.
1302	'FRAME' OR 'NOFRAME' OPTION IGNORED	The **frame** option is specified when optimization is specified, or the **noframe** option is specified when no optimization is specified. S: Ignores the specified option. P: Frame pointer is not used when optimization is specified; the frame pointer is always used when optimization is not specified. Cancel the **frame** and **noframe** specifications.
1303	COMPLETED FILE NAME TOO LONG	The length of a file name including the path name starting from the route directory exceeds 251 characters. S: Outputs the file name specified at the command line to the **debug** information. P: Reduce the work directory nesting level or change the file name so that the length of the file name, including the path name, is less than or equal to 251 characters.

Error No.	Message	Explanation
1305	"SHOW=OBJECT" OPTION IGNORED	When assembly source program output is specified, the **show=object** option is specified. S: Ignores the specified option. P: The object list is not output to the listing file when assembly source program output is specified. Cancel the **show=object** or **code=asmcode** option.
1306	"SPEED=INLINE" OPTION IGNORED	The **speed=inline** option is specified when no optimization is specified. S: Ignores the specified option. P: Function inline expansion is not performed when optimization is not specified. Cancel the **speed=inline** option or specify the **optimize=1** option.
1307	SECTION NAME TOO LONG	The length of a section name exceeds 32 characters. S: Uses the first 32 characters and ignores the rest. P: Set the length of section names specified by **#pragma section** to 32 characters or less.
1308	"SPEED=LOOP" OPTION IGNORED	The **speed=loop** option is specified when no optimization is specified. S: Ignores the specified option. P: Loop code expansion with speed priority is not performed when optimization is not specified. Cancel the **speed=loop** option or specify the **optimize=1** option.
1400	#PRAGMA INLINE IS NOT EXPANDED	Function inline expansion cannot be performed. S: Ignores the **#pragma inline** specification. P: Inline expansion cannot be performed when inline expansion conditions are not satisfied. Meet inline expansion conditions.

Error No.	Message	Explanation
1401	#PRAGMA ABS16 IGNORED	The **#pragma abs16** is specified when the CPU/operating mode is the 2600n, 2000n, 300hn, or 300 mode. S: Ignores the **#pragma abs16** specification. P: The **#pragma abs16** is valid only when the CPU/operating mode is the 2600a, 2000a, or 300ha mode. Change the CPU/operating mode or cancel the **#pragma abs16** specification.
1402	ILLEGAL CPUEXPAND EXPESSION	The expression for **cpuexpand** (expansion interpretation of operation size) has a variable other than the **volatile** variable. S: Does not use **cpuexpand** (expansion interpretation of operation size). P: Specify the **volatile** variable.
1403	#PRAGMA ASM IGNORED	The **#pragma asm** is specified when the object format is a relocatable object program. S: Ignores the **#pragma asm** specification. P: The **#pragma asm** is valid only when the object format is an assembly source program. Specify the **code=asmcode** option.

(3) Error-Level Messages

Error No.	Message	Explanation
2000	ILLEGAL PREPROCESSOR KEYWORD	An illegal keyword is used in a preprocessor directive. S: Ignores the line containing the preprocessor directive. P: Correct the keyword in the preprocessor directive.
2001	ILLEGAL PREPROCESSOR SYNTAX	There is an error in a preprocessor directive or in a macro call specification. S: Ignores the line containing the preprocessor directive or macro call. If there is an error in a constant expression used in the preprocessor directive, the system assumes that the constant expression is zero. P: Specify the correct preprocessor directive or macro call.
2002	' , ' NOT FOUND	A comma (,) is not used to delimit two parameters in a **#define** directive. S: Assumes that there is a comma. P: Insert a comma.
2003	') ' NOT FOUND	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. S: Assumes that there is a right parenthesis. P: Insert a right parenthesis.
2004	' > ' NOT FOUND	A right angle bracket (>) does not follow a file name in an **#include** directive S: Assumes that there is a right angle bracket. P: Insert a right angle bracket.
2005	CANNOT OPEN INCLUDE	The file specified by an **#include** directive cannot be opened. S: Ignores the **#include** directive. P: Specify the correct file name. If the file name is correct, check if the file reading is disabled.

Error No.	Message	Explanation
2006	MULTIPLE #DEFINE'S	The same macro name is redefined by **#define** directives. S: Ignores the second **#define** directive. P: Modify one of the macro names or delete one of the **#define** directives.
2008	#ELIF MISMATCHES	There is no **#if, #ifdef, #ifndef,** or **#elif** directive corresponding to an **#elif** directive. S: Ignores the **#elif** directive P: Insert the corresponding preprocessor directive or delete the **#elif** directive.
2009	#ELSE MISMATCHES	There is no **#if, #ifdef,** or **#ifndef** directive corresponding to an **#else** directive S: Ignores the **#else** directive. P: Insert the corresponding preprocessor directive or delete the **#else** directive.
2010	MACRO PARAMETERS MISMATCH	The number of macro call parameters is not equal to the number of macro definition parameters. S: Ignores the excess parameters if there are too many, or assumes blank character strings if the number of parameters is insufficient. P: Specify the correct number of macro parameters.
2011	LINE TOO LONG	A source program line exceeds 4096 characters after macro expansion. S: Ignores the 4097th and subsequent characters. P: Separate the line so that the length of each resulting line contains 4096 or less characters after macro expansion.
2012	KEYWORD AS A MACRO NAME	A preprocessor keyword is used as a macro name in the **#define** or **#undef** directive. S: Ignores the **#define** or **#undef** directive. P: Change the macro name.

Error No.	Message	Explanation
2013	#ENDIF MISMATCHES	There is no **#if, #ifdef**, or **#ifndef** directive corresponding to the **#endif** directive. S: Ignores the **#endif** directive. P: Check that the **#endif** directive is used correctly.
2014	#ENDIF EXPECTED	There is no **#endif** directive corresponding to an **#if, #ifdef,** or **#ifndef** directive, and the end of the file is detected. S: Assumes that there is an **#endif** directive. P: Insert an **#endif** directive.
2016	PREPROCESSOR CONSTANT EXPRESSION TOO COMPLEX	The total number of operators and operands in a constant expression specified by the **#if** or **#elif** directive exceeds 512. S: Assumes the value of the constant expression is 0. P: Correct the constant expression so that the number of operators and operands in the constant expression is less than or equal to 512.
2017	MISSING "	A closing double quotation mark (") does not follow a file name in the **#include** directive. S: Assumes that there is a closing double quotation mark. P: Insert a closing double quotation mark.
2018	ILLEGAL #LINE	The line count specified by the **#line** directive exceeds 32767. S: Ignores the **#line** directive. P: Modify the program so that the line count is less than or equal to 32767.
2019	FILE NAME TOO LONG	The length of a file name exceeds 128 characters. S: Uses the first 128 characters as the file name. P: Change the file name so that the length is less than or equal to 128 characters.

Error No.	Message	Explanation
2020	SYSTEM IDENTIFIER REDEFINED: **"name"**	A symbol having the same name as an intrinsic function name is defined. S: Continues processing as a symbol. P: Define the symbol having a name different from the intrinsic function name.
2021	SYSTEM IDENTIFIER MISMATCH: **"name"**	An intrinsic function not corresponding to the specified CPU/operating mode is used. S: Ignores the intrinsic function and continues processing. P: Use the intrinsic function corresponding to the CPU/operating mode.
2100	MULTIPLE STORAGE CLASSES	Two or more storage class specifiers are used in a declaration. S: Uses the first storage class specifier and ignores others. P: Specify the correct storage class specifier.
2101	ADDRESS OF REGISTER	The unary operator & is used on a **register** variable. S: Assumes that the **auto** storage class is specified for the variable and continues processing. P: Modify the declaration so that the storage class of the variable is **auto**.
2102	ILLEGAL TYPE COMBINATION	A combination of type specifiers is illegal. S: Uses the first and longest legal combination of type specifiers and ignores the rest. P: Change the type specifiers to a legal combination.
2103	BAD SELF REFERENCE STRUCTURE	A **struct** or **union** member has the same data type as its parent. S: Assumes the data type of the member is **int**. P: Declare the correct data type for the member.
2104	ILLEGAL BIT FIELD WIDTH	A constant expression indicating the width of a bit field is not a positive integer. S: Ignores the bit field width specification and assumes that the member is not a bit field. P: Specify the correct width for the bit field.

Error No.	Message	Explanation
2105	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. S: Assumes that the incomplete or undeclared tag name is an **int**. P: Declare the incomplete or undeclared tag name.
2106	EXTERN VARIABLE INITIALIZED	A compound statement specifies an initial value for an **extern** storage class variable. S: Ignores the initial value. P: Specify the initial value for the external definition of the variable.
2107	ARRAY OF FUNCTION	An array with a function member type is specified. S: Ignores the function or array type. P: Specify the correct type.
2108	FUNCTION RETURNING ARRAY	A function with an array return value type is specified. S: Ignores the function or array type. P: Specify the correct type.
2109	ILLEGAL FUNCTION DECLARATION	A storage class other than **extern** is specified in the declaration of a function variable used in a compound statement. S: Assumes **extern** as the storage class. P: Specify the correct storage class.
2110	ILLEGAL STORAGE CLASS	The storage class in an external definition is specified as **auto** or **register**. S: Assumes that the storage class is **extern.** P: Specify the correct storage class.
2111	FUNCTION AS A MEMBER	A member of a **struct** or **union** is declared as a function S: Assumes **int** as the member type. P: Specify the correct storage class.

Error No.	Message	Explanation
2112	ILLEGAL BIT FIELD TYPE	The type specifier for a bit field is illegal. **char, unsigned char, int, unsigned int, short, unsigned short**, or a combination of **const** or **volatile** with one of the above types is allowed as a type specifier for a bit field. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Specify the correct type.
2113	BIT FIELD TOO WIDE	The width of a bit field is greater than the size (8 or 16 bits) indicated by its type specifier. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Specify the correct bit field width.
2114	MULTIPLE VARIABLE DECLARATIONS	A variable name is declared more than once in the same scope. S: Uses the first declaration and ignores subsequent declarations. P: Keep one of the declarations and delete or modify the rest.
2115	MULTIPLE TAG DECLARATIONS	A **struct**, **union**, or **enum** tag name is declared more than once in the same scope. S: Uses the first declaration and ignores subsequent declarations. P: Keep one of the tag name declarations and delete or modify the rest.
2117	EMPTY SOURCE PROGRAM	There are no external definitions in the source program. S: Terminates processing. P: Specify and compile the correct source program.
2118	PROTOTYPE MISMATCH: **"function name"**	A function type differs from the one specified in the declaration. S: Ignores the current declaration if the function prototype declaration is being processed. Ignores the previous declaration if the declaration of an external function definition is being processed. P: Correct the declaration so that the function types match.

Error No.	Message	Explanation
2119	NOT A PARAMETER NAME: **"parameter name"**	An identifier not in the function parameter list is declared as a parameter. S: Ignores the parameter declaration. P: Check that the function parameter list matches all parameter declarations.
2120	ILLEGAL PARAMETER STORAGE CLASS	A storage class other than **register** is specified in a function parameter declaration. S: Ignores the storage class specifier. P: Delete the storage class specifier.
2121	ILLEGAL TAG NAME	The combination of a tag name and **struct**, **union**, or **enum** differs from the declared combination. S: Assumes **struct, union**, or **enum** depending on the tag name type. P: Specify the correct combination of a tag name and a **struct, union**, or **enum**.
2122	BIT FIELD WIDTH 0	The width of a bit field which is a member of a **struct** or **union** is zero. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Delete the member name or specify the correct bit field width.
2123	UNDEFINED TAG NAME	An undefined tag name is specified in an **enum** declaration. S: Ignores the declaration. P: Specify the correct tag name.
2124	ILLEGAL ENUM VALUE	A non-integral constant expression is specified as a value for an **enum** member. S: Ignores the value specification. P: Change the expression to an integer constant expression.
2125	FUNCTION RETURNING FUNCTION	A function with a function return value is specified. S: Ignores one of the function types. P: Specify the correct type.

Error No.	Message	Explanation
2126	ILLEGAL ARRAY SIZE	The value that specifies the number of arrays exceeds the limit. The limit is 65535 for a CPU/operating mode of 2600n, 2000n, 300hn, or 300, 1048575 for a CPU/operating mode of 2600a:20, 2000a:20, or 300ha:20, 16777215 for a CPU/operating mode of 2600a:24, 2000a:24, or 300ha:24, 268435455 for a CPU/operating mode of 2600a:28 or 2000a:28, and 4294967295 for a CPU/operating mode of 2600a:32 or 2000a:32. S: Assumes that one is the number of array elements. P: Specify an allowable number of array elements.
2127	MISSING ARRAY SIZE	The number of elements in an array is not specified where it is required. S: Assumes that the number of array elements is one. P: Specify the number of array elements.
2128	ILLEGAL POINTER DECLARATION	A type specifier other than **const** or **volatile** is specified following an asterisk (*), which indicates a pointer declaration. S: Ignores the type specifier following the asterisk. P: Specify the correct type specifier following the asterisk.
2129	ILLEGAL INITIALIZER TYPE	The initial value specified for a variable is not a type that can be assigned to the variable. S: Does not initialize the variable. P: Specify the correct type initial value.
2130	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. S: Does not initialize the variable. P: Specify a constant expression as the initial value.
2131	NO TYPE NOR STORAGE CLASS	Storage class and type specifiers are not given in an external data definition. S: Assumes **int** as the type specifier. P: Insert the storage class or type specifier.

Error No.	Message	Explanation
2132	NO PARAMETER NAME	A parameter is declared even though the function parameter list is empty. S: Ignores the parameter declaration. P: Insert the parameter name in the function parameter list or delete the parameter declaration.
2133	MULTIPLE PARAMETER DECLARATIONS	Either a parameter name is declared in a function definition parameter list more than once or a parameter is declared inside and outside the function declarator. S: Uses the first declaration if a parameter is declared more than once in the function parameter list. Uses the declaration inside the function declarator if a parameter is declared inside and outside the function declarator. P: Keep one of the declarations and delete the rest.
2134	INITIALIZER FOR PARAMETER	An initial value is specified in the declaration of a parameter. S: Does not use the initial value specification. P: Delete the initial value specification.
2135	MULTIPLE INITIALIZATION	A variable is initialized more than once. S: Ignores the second and subsequent initializations. P: Delete any redundant declarations.
2136	TYPE MISMATCH	An **extern** or **static** variable or function is declared more than once with different data types. S: Uses the type specified in the definition declaration where a definition is declared. Otherwise, the data type specified in the first declaration is used. P: Use the same data type in the declarations.

Error No.	Message	Explanation
2137	NULL DECLARATION FOR PARAMETER	An identifier is not specified in the function parameter declaration. S: Ignores the corresponding parameter declaration. P: Delete the parameter declaration or insert the correct parameter name.
2138	TOO MANY INITIALIZERS	The number of initial values specified for a **struct** 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. S: Uses only the initial values corresponding to the number of **struct** members, array elements, or the first members of **union**. The rest are ignored. P: Specify the correct number of initial values.
2139	NO PARAMETER TYPE	A type is not specified in a function parameter declaration. S: Assumes **int** as the parameter declaration type. P: Specify the correct type for the parameter declaration.
2140	ILLEGAL BIT FIELD	A bit field is used in a **union**. S: Ignores the bit field. P: Use the bit field in a **struct.**
2141	ILLEGAL BIT FIELD	An unnamed bit field is used as the first member of a **struct**. S: Ignores the bit field. P: Specify the name of the bit field.

Error No.	Message	Explanation
2142	ILLEGAL VOID TYPE	**void** is used illegally. S: Assumes that **void** is **int**. P: **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	ILLEGAL STATIC FUNCTION	A **static** storage class function has no definition in the source program. S: Ignores the function declaration. P: Either delete the function declaration or define the function.
2200	INDEX NOT INTEGER	An array index expression type is not an integer. S: Assumes that the type is **int**. P: Specify an integer expression for the array index.
2201	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. S: Assumes that the correct parameter type is specified and continues processing. P: Specify an expression whose type corresponds to the one specified in the prototype declaration.
2202	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. S: Assumes that the number of parameters for the function call is equal to the number of parameters specified in the prototype declaration, and continues processing. P: Specify the correct number of parameters.

Error No.	Message	Explanation
2203	ILLEGAL MEMBER REFERENCE	The expression to the left of the (.) operator is not a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Use a **struct** or **union** expression to the left of the (.) operator.
2204	ILLEGAL MEMBER REFERENCE	The expression to the left of the –> operator is not a pointer to a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Use an expression which deals with pointer to **struct** or **union** to the left of the –> operator according to the member.
2205	UNDEFINED MEMBER NAME	An undeclared member name is used to reference a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Specify the correct member name.
2206	MODIFIABLE LVALUE REQUIRED	The operand for a unary prefix or suffix operator ++ or – – has an lvalue that cannot be assigned (an lvalue whose type is not array or **const**). S: Assumes that the expression with an lvalue that can be assigned is specified as an operand and continues processing. P: Specify an expression, whose lvalue can be assigned, as an operand.
2207	SCALAR REQUIRED	The unary operator ! is used on an expression that is not scalar. S: Assumes **int** as the type of the result and continues processing. P: Use a scalar expression as the operand.
2208	POINTER REQUIRED	The operand for the unary operator * is an expression of pointer to **void** or is not an expression of pointer. S: Ignores *. P: Use an operand that is an expression other than pointer to **void**.

Error No.	Message	Explanation
2209	ARITHMETIC TYPE REQUIRED	The unary operator + or – is used on a non-arithmetic expression. S: Assumes that the operand type is **int** and continues processing. P: Change the expression to an arithmetic expression.
2210	INTEGER REQUIRED	The unary operator ~ is used on a non-integral expression. S: Assumes that the result type is **int** and continues processing. P: Change the expression to an integral expression.
2211	ILLEGAL SIZEOF	A **sizeof** operator is used for a bit field member, function, **void**, or array with an undefined size. S: Assumes **int** as the operand type and continues processing. P: A **sizeof** operator cannot be used to obtain the size of a bit field, function, **void**, or array with an undefined size. Use an appropriate operand.
2212	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. S: Assumes that the result is **int** and continues processing. P: Cast operation can only be performed on scalar data items. Use appropriate operands.
2213	ARITHMETIC TYPE REQUIRED	The binary operator *, /, *=, or /= is used in an expression that is not arithmetic. S: Assumes **int** as the result and continues processing. P: Specify arithmetic expressions as the operands.

Error No.	Message	Explanation
2214	INTEGER REQUIRED	The binary operator <<, >>, &, |, ^, %, <<=, >>=, &=, | =, ^=, or %= is used in an expression that is not an integer expression. S: Assumes **int** as the result type and continues processing. P: Specify integer expressions as the operands.
2215	ILLEGAL TYPE FOR +	The combination of operand types used with the binary operator + is illegal. S: Assumes the result type is **int** and continues processing. P: Specify a correct type of operands. Only the following type combinations are allowed for the binary operator +: — Two arithmetic operands — Pointer and integer
2216	ILLEGAL TYPE REQUIRED FOR PARAMETER	**void** is specified for a function call parameter type. S: Ignores the parameter type and continues processing. P: Specify a function call parameter type so that a value can be passed to the function.
2217	ILLEGAL TYPE FOR –	The combination of operand types used with the binary operator – is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following three combinations are allowed for the binary operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) The first operand is a pointer and the second operand is an integer.

Error No.	Message	Explanation
2218	SCALAR REQUIRED	The first operand of the conditional operator ?: is not a scalar. S: Assumes that the first operand is a scalar and continues processing. P: Specify a scalar expression as the first operand.
2219	TYPE NOT COMPATIBLE	The types of the second and third operands of the conditional operator ?: do not match with each other. S: Assumes that the result type is i**nt** and continues processing. P: Specify a correct type combination of operands. Only one of the following six combinations is allowed for the second and third operands when using the ?: operator: (1) Two arithmetic operands (2) Two **void** operands (3) Two pointers assigned to the same data type (4) A pointer and an integer constant whose value is zero or another pointer that is assigned to **void** that was converted from an integer constant whose value is zero (5) A pointer and another pointer assigned to **void** (6) Two **struct** or **union** variables with the same data type
2220	MODIFIABLE LVALUE REQUIRED	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 | =. S: Assumes that the left expression whose left value can be assigned is used and continues processing. P: Specify a left expression whose left value can be assigned.

Error No.	Message	Explanation
2221	ILLEGAL TYPE FOR POSTINCREMENT OR POSTDECREMENT	The operand of the unary suffix operator ++ or – – is a function type, a pointer assigned to **void**, or not a scalar type. S: Assumes that the result type is **int** and continues processing. P: Use a scalar type that is not a function or a pointer assigned to **void** as the operand.
2222	TYPE NOT COMPATIBLE	The operand types for the assignment operator = do not match. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following five types of combinations are allowed for the operands of the = assignment operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) The left operand is a pointer and the right operand is an integer constant whose value is zero or another pointer that is assigned to **void** that was converted from an integer constant whose value is zero. (4) A pointer and another pointer assigned to **void** (5) Two **struct** or **union** variables with the same data type
2223	INCOMPLETE TAG USED IN EXPRESSION	An incomplete tag name is used for a **struct** or **union** in an expression. S: Assumes that the incomplete tag name is **int** and continues processing. P: Declare the tag name.

Error No.	Message	Explanation
2224	ILLEGAL TYPE FOR ASSIGN	The operand types of the assignment operator += or –= are illegal. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following two types of combinations are allowed as operands for the assignment operator += or –=: (1) Two arithmetic operands (2) The left operand is a pointer and the right operand is an integer.
2225	UNDECLARED NAME: **"name"**	An undeclared name is used in an expression. S: Assumes that the name is declared as an **int** external identifier and continues processing. P: Either declare the name or modify it so that it corresponds with one of the declared names.
2226	SCALAR REQUIRED	The binary operator && or || is used in a non-scalar expression. S: Assumes that the result type is **int** and continues processing. P: Use scalar expressions as operands.
2227	ILLEGAL TYPE FOR EQUALITY	The combination of operand types for the equality operator == or != is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following three combinations of operand types for the equality operator == or != are allowed: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) A pointer and an integer constant whose value is zero or another pointer assigned to **void**

Error No.	Message	Explanation
2228	ILLEGAL TYPE FOR COMPARISON	The combination of operand types for the relational operator >, <, >=, or <= is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following two combinations of operand types are allowed for a relational operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type
2230	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. S: Ignores the actual parameter list and the parentheses which indicate this list. P: Correctly specify a function type expression or pointer assigned to a function type.
2231	ADDRESS OF BIT FIELD	The unary operator & is used on a bit field. S: Ignores the bit field, assumes that the unary operator & is correctly specified, and continues processing. P: Correct the expression. A bit field address cannot be used.
2232	ILLEGAL TYPE FOR PREINCREMENT OR PREDECREMENT	A type that is not a scalar, or that is a pointer assigned to a function or **void** is specified as the operand for the prefix operator ++ or – –. S: Assumes **int** as the result type and continues processing. P: Use an operand that is a scalar other than a pointer assigned to a function or **void**.
2233	ILLEGAL ARRAY REFERENCE	An expression used as an array is not one of the following types. — Array — Pointer assigned to a data type other than a function or **void** S: Ignores the square brackets ([ ]) and the array subscript enclosed. P: When an array subscript is required, specify the correct expression.

Error No.	Message	Explanation
2234	ILLEGAL TYPEDEF NAME REFERENCE	A **typedef** name is used as a variable in an expression. S: Ignores the expression. P: Use **typedef** name correctly.
2235	ILLEGAL CAST	An attempt is made to cast a pointer with a floating-point type. S: Ignores the attempt. P: Cast the pointer with an integer type, then with a floating-point type.
2236	ILLEGAL CAST IN CONSTANT	If the CPU/operating mode is 300, 300hn, 2600n, or 2000n, an attempt is made to cast a pointer with a **char** or **long** in a constant expression. If the CPU/operating mode is 300ha, 2600a, or 2000a, an attempt is made to cast a pointer with a **char, short**, or **int** in a constant expression. S: Ignores the cast operation. P: Use an expression other than a constant expression.
2237	ILLEGAL CONSTANT EXPRESSION	In a constant expression, a pointer constant cast with an integer is specified as an operand. S: Assumes that the conversion is not specified and continues processing. P: Use an expression other than a constant expression.
2238	LVALUE OR FUNCTION TYPE REQUIRED	The unary operator & is used on the left value or is used in an expression other than function type. S: Assumes that an expression with a left value is specified as the operand and continues processing. P: Specify an expression that has a left value or a function type expression as the operand.

Error No.	Message	Explanation
2300	CASE NOT IN SWITCH	A **case** label is specified outside a **switch** statement. S: Ignores the **case** label. P: Specify the **case** label in the **switch** statement.
2301	DEFAULT NOT IN SWITCH	A **default** label is specified outside a **switch** statement. S: Ignores the **default** label. P: Specify the **default** label in the **switch** statement.
2302	MULTIPLE LABELS	A label is defined more than once in a function. S: Ignores redundant label definitions. P: Keep one label name and delete or modify the other.
2303	ILLEGAL CONTINUE	A **continue** statement is specified outside a **while, for,** or **do** statement. S: Ignores the **continue** statement. P: Only use the **continue** statement in the **while, for,** or **do** statement.
2304	ILLEGAL BREAK	A **break** statement is specified outside a **while, for, do,** or **switch** statement. S: Ignores the **break** statement. P: Only use the **break** statement in the **while, for, do,** or **switch** statement.
2305	VOID FUNCTION RETURNS VALUE	A **return** statement specifies a return value for a function with a **void** return type. S: Ignores the **return** statement expression. P: For a function with a **void** return type, do not specify an expression in the **return** statement or do not use the **return** statement.
2306	CASE LABEL NOT CONSTANT	A **case** label expression is not an integer constant expression. S: Ignores the **case** label. P: Use an integer constant expression for the **case** label.

Error No.	Message	Explanation
2307	MULTIPLE CASE LABELS	Two or more **case** labels with the same value are used in one **switch** statement. S: Ignores redundant **case** labels. P: Modify the **switch** statement so that each **case** label has a unique value.
2308	MULTIPLE DEFAULT LABELS	Two or more **default** labels are specified for one **switch** statement. S: Ignores redundant **default** labels. P: Modify the **switch** statement so that it has only one **default** label.
2309	NO LABEL FOR GOTO	There is no label corresponding to the destination specified by a **goto** statement. S: Continues processing. P: Specify the correct label in the **goto** statement.
2310	SCALAR REQUIRED	The control expression (that determines statement execution) for a **while, for**, or **do** statement is not a scalar. S: Assumes that an **int** control expression is specified and continues processing. P: Use a scalar expression as the control expression for the **while, for,** or **do** statement.
2311	INTEGER REQUIRED	The control expression (that determines statement execution) for a **switch** statement is not an integer. S: Assumes that an **int** control expression is specified and continues processing. P: Use an integer expression as the control expression for the **switch** statement.
2312	MISSING (	The control expression (that determines statement execution) does not follow a left parenthesis “(” for an **if, while, for, do,** or **switch** statement. S: Assumes that the control expression follows a left parenthesis and continues processing. P: Specify the control expression for the **if, while, for, do,** or **switch** statement and enclose it in parentheses.

Error No.	Message	Explanation
2313	MISSING ;	A **do** statement is ended without a semicolon (;). S: Assumes that the **do** statement ends with a semicolon (;) and continues processing. P: Place a semicolon at the end of the **do** statement.
2314	SCALAR REQUIRED	A control expression (which determines statement execution) for an **if** statement is not a scalar. S: Assumes that an **int** control expression is specified and continues processing. P: Use a scalar expression as the control expression for the **if** statement.
2316	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. S: Assumes that the expression in the **return** statement is the type expected to be returned by the function and continues processing. P: Convert the expression in the **return** statement so that it matches the type of value expected.
2330	ILLEGAL INTERRUPT FUNCTION DECLARATION	The interrupt function declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma interrupt** directive using the correct format.
2331	ILLEGAL INTERRUPT FUNCTION CALL	A function with an interrupt function declaration is called or referenced. S: Assumes the function was not declared to be an interrupt function and continues processing. P: A function declared to be an interrupt function in the same file, cannot be called or referenced during processing. Delete the interrupt function declaration or the statement which calls or references the function.

Error No.	Message	Explanation
2332	INTERRUPT FUNCTION ALREADY DECLARED	The function specified by interrupt function declaration **#pragma interrupt** has been declared as a normal function. S: Ignores the interrupt function declaration. P: Add the interrupt function declaration before the corresponding function declaration.
2333	MUTIPLE INTERRUPT FOR ONE FUNCTION	Interrupt function declaration **#pragma interrupt** has been declared more than once against the same function. S: Ignores the interrupt function declaration. P: Delete the second and subsequent declarations.
2334	ILLEGAL PARAMETER IN INTERRUPT FUNCTION	Parameter type used for interrupt function is illegal. S: Ignores the interrupt function declaration. P: Correct the description of the interrupt function. Interrupt functions cannot pass parameters. Only **void** can be used for the parameter.
2335	MISSING PARAMETER DECLARATION IN INTERRUPT FUNCTION	An undeclared variable or function is used in stack switching specification (sp) for interrupt function declaration **#pragma interrupt** or interrupt function end specification (sy). S: Ignores the interrupt function declaration. P: Provide a variable or function declaration before the interrupt function declaration **#pragma interrupt**.
2336	PARAMETER OUT OF RANGE IN INTERRUPT FUNCTION	The value of parameter **tn** for interrupt function declaration **#pragma interrupt** exceeds 3. S: Ignores parameter tn. P: Modify parameter tn to 3 or less.
2340	IILLEGAL ABS8 DECLARATION	Short absolute address variable declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma abs8** directive using the correct format.

Error No.	Message	Explanation
2341	ABS8 ALREADY DECLARED	The variable specified by short absolute address variable declaration **#pragma abs8** has been declared as a variable. S: Ignores the short absolute address variable declaration. P: Add the short absolute address variable declaration before the corresponding variable declaration.
2342	ILLEGAL ABS8 TYPE	The variable specified by short absolute address variable declaration **#pragma abs8** has been declared as a type other than a variable name. S: Ignores the short absolute address variable declaration. P: Declare and define the name specified by the short absolute address variable as a variable name.
2345	IILLEGAL ABS16 DECLARATION	Short absolute address variable declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma abs16** directive using the correct format.
2346	ABS16 ALREADY DECLARED	The variable specified by short absolute address variable declaration **#pragma abs16** has been declared as a variable. S: Ignores the short absolute address variable declaration. P: Add the short absolute address variable declaration before the corresponding variable declaration.
2347	ILLEGAL ABS16 TYPE	The variable specified by short absolute address variable declaration **#pragma abs16** has been declared as a type other than a variable name. S: Ignores the short absolute address variable declaration. P: Declare and define the name specified by the short absolute address variable as a variable name.

Error No.	Message	Explanation
2350	IILLEGAL SECTION NAME DECLARATION	The **#pragma section** specification is illegal. S: Ignores the section name specification. P: Correct the **#pragma section** specification.
2352	SECTION NAME TABLE OVERFLOW	The number of normal sections, 8/16-bit absolute address sections, and indirect memory call sections exceed 64, or the total number of sections exceeds 256. S: Ignores section name specification. P: Reduce the specified sections or divide the file.
2360	IILLEGAL INDIRECT FUNCTION DECLARATION	Indirect memory function declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma indirect** directive using the correct format.
2361	INDIRECT FUNCTION ALREADY DECLARED	The function specified by indirect memory function declaration **#pragma indirect** has been declared as a function. S: Ignores the indirect memory function declaration. P: Add the indirect memory function declaration before the corresponding function declaration.
2362	ILLEGAL INDIRECT TYPE	The function specified by indirect memory function declaration **#pragma indirect** has been declared and defined as a type other than a function. S: Ignores the **#pragma indirect** specification. P: Declare, refer, and define as a function type the name specified by **#pragma indirect**.
2363	TOO MANY INDIRECT IDENTIFIERS	The number of names that can be specified in a file of the indirect memory function exceeds the limit. The limit is 128 for a CPU/operating mode of 2600n, 2000n, 300hn, or 300, and 64 for a CPU/operating mode of 2600a, 2000a, or 300ha. S: Ignores the indirect memory function. P: Reduce the indirect memory function specifications or divide the file.

Error No.	Message	Explanation
2370	IILLEGAL REGSAVE/NOREGSAVE DECLARATION	The **#pragma regsave** or **#pragma noregsave** declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma regsave** or **#pragma noregsave** directive using the correct format.
2371	REGSAVE/NOREGSAVE FUNCTION ALREADY DECLARED	The function specified by **#pragma regsave** or **#pragma noregsave** has been declared as a function. S: Ignores the **#pragma regsave** or **#pragma noregsave** declaration. P: Add the **#pragma regsave** or **#pragma noregsave** declaration before the corresponding function declaration.
2372	ILLEGAL REGSAVE/NOREGSAVE TYPE	The function specified by **#pragma regsave** or **#pragma noregsave** has been declared and defined as a type other than a function. S: Ignores the **#pragma regsave** or **#pragma noregsave** specification. P: Declare, refer, and define as a function type the name specified by **#pragma regsave** or **#pragma noregsave**.
2380	IILLEGAL IN-LINE DECLARATION	The **#pragma inline** declaration is illegal. S: Ignores the illegal and subsequent declarations. P: Declare the **#pragma inline** directive using the correct format.
2381	IN-LINE FUNCTION ALREADY DECLARED	The function specified by **#pragma inline** has been declared as a function. S: Ignores the **#pragma inline** declaration. P: Add the **#pragma inline** declaration before the corresponding function declaration.

Error No.	Message	Explanation
2382	ILLEGAL IN-LINE TYPE	The function specified by **#pragma inline** has been declared and defined as a type other than a function. S: Ignores the **#pragma inline** specification. P: Declare, refer, and define as a function type the name specified by **#pragma inline**.
2400	ILLEGAL CHARACTER: **"character"**	An illegal character is detected. S: Assumes that the character is a blank character and continues processing. P: Delete the character.
2401	INCOMPLETE CHARACTER CONSTANT	An end of a line is detected in the middle of a character constant. S: Assumes that a quotation mark (') is placed before the end of line indicator and continues processing. P: Correct the character constant.
2402	INCOMPLETE STRING	An end of line is detected in the middle of a string literal. S: Assumes that a double quotation mark (") is placed before the end of line indicator and continues processing. P: Correct the string literal.
2403	EOF IN COMMENT	An end of file is detected in the middle of a comment. S: Assumes that the program ends when the end of file indicator is reached and continues processing. P: End the comment with */.
2404	ILLEGAL CHARACTER CODE: **"character code"**	An illegal character code is detected. S: Assumes that the character code is a blank character and continues processing. P: Delete the illegal character code.
2405	NULL CHARACTER CONSTANT	There are no characters in a character constant (i.e., no characters are specified between two quotation marks). S: Assumes that '\0' is specified and continues processing. P: Correct the character constant.

Error No.	Message	Explanation
2406	OUT OF FLOAT	The number of significant digits in a floating-point constant exceeds 17. S: Depending on the sign, the system assumes +∞ or –∞. P: Ensure that the number of significant digits in a floating-point constant is less than or equal to 17.
2407	INCOMPLETE LOGICAL LINE	A backslash (\) or a backslash followed by an end of the line indicator (\ (RET)) is specified as the last character in a non-empty source file. S: Ignores the last logical line. P: Delete the backslash or continue the physical line.
2500	ILLEGAL TOKEN: **"token"**	An illegal token sequence is used. S: Ignores data up to a semicolon (;), left brace ({), right brace (}), comma (,), or keyword (**if, while, for, switch, do, case, default, return, break,** or **continue**). P: Correct the token sequence.
2501	DIVISION BY ZERO	An integer is divided by zero in a constant expression. S: Assumes a result value of zero and continues processing. P: Modify the constant expression so that an integer is not divided by zero.
2600	#ERROR DIAGNOSTIC MESSAGE: **"character string"**	An error message specified by string literal **#error** is output to the list file if **nolist** option is not specified. S: Continues processing.
2801	ILLEGAL PARAMETER TYPE IN IN-LINE FUNCTION	The parameter type of a intrinsic function is illegal. S: Disables the intrinsic function. P: Set the correct parameter.
2802	PARAMETER OUT OF RANGE IN IN-LINE FUNCTION	The parameter value of a intrinsic function is out of range. S: Disables the intrinsic function. P: Set the allowable parameter value.

(4) Fatal-Level Messages

Error No.	Message	Explanation
3000	STATEMENT NEST TOO DEEP	The nesting level of **if, while, for, do,** and **switch** statements exceeds 32. S: Terminates processing. P: Modify the program so that the nesting level never exceeds 32.
3001	BLOCK NEST TOO DEEP	The nesting level of compound statements exceeds 32. S: Terminates processing. P: Modify the program so that the nesting level never exceeds 32.
3002	#IF NEST TOO DEEP	The conditional compilation (**#if, #ifdef, #ifndef, #elif,** and **#else**) nesting level exceeds 32. S: Terminates processing. P: Modify the program so that the nesting level never exceeds 32.
3003	TOO MANY EXTERNAL IDENTIFIERS	The number of external identifiers exceeds the limit. S: Terminates processing. P: Expand the limit using the **limits** option, or divide the program so that the number of external identifiers is less than or equal to the limit.
3004	TOO MANY LOCAL IDENTIFIERS	The number of effective identifiers (internal identifiers) in one function exceeds the limit. S: Terminates processing. P: Expand the limit using the **limits** option, or divide the compound statements so that the number of identifiers declared in one compound statement is less than or equal to the limit.
3005	TOO MANY MACRO IDENTIFIERS	The number of macro names defined in the **#define** directive exceeds the limit. S: Terminates processing. P: Expand the limit using the **limits** option, or divide the program so that the number of macro names is less than or equal to the limit.

Error No.	Message	Explanation
3006	TOO MANY PARAMETERS	The number of parameters in either a function declaration or a function call exceeds 63. S: Terminates processing. P: Modify the program so that the number of function parameters is less than or equal to 63.
3007	TOO MANY MACRO PARAMETERS	The number of parameters in a macro definition or a macro call exceeds 63. S: Terminates processing. P: Modify the program so that the number of macro parameters is less than or equal to 63.
3008	LINE TOO LONG	After a macro expansion, the length of a line exceeds 4096 characters. S: Terminates processing. P: Divide the line so that its length does not exceed 4096 characters after macro expansion.
3009	STRING LITERAL TOO LONG	The length of the character string exceeds 512 characters. The length of the string literal is the byte number generated after the specified string is connected continuously. The length of the string literal in the source program is not the length of the source program, in the string data. This byte number is located in the string literal data with the expansion sign counted as one character. S: Terminates processing. P: Modify the program so that the total length of the string literal does not exceeds 512 bytes.
3010	#INCLUDE NEST TOO DEEP	The nesting level of the **#include** directive exceeds the limit of 30. S: Terminates processing. P: Ensure that the file inclusion nesting level does not exceed the limit of 30.

Error No.	Message	Explanation
3011	MACRO EXPANSION NEST TOO DEEP	The nesting level of macro expansion performed by the **#define** directive exceeds 32. S: Terminates processing. P: Modify the program so that the nesting level of macro expansion never exceeds 32. Note that a macro may be defined recursively.
3012	TOO MANY FUNCTION DEFINITIONS	The number of function definitions is greater than 512. S: Terminates processing. P: Divide the program so that the number of function definitions is less than or equal to 512 in one compile unit.
3013	TOO MANY SWITCHES	The number of **switch** statements exceeds 256. S: Terminates processing. P: Divide the program so that the number of **switch** statements is less than or equal to 256 in one compile unit.
3014	FOR NEST TOO DEEP	The nesting level of **for** statement exceeds 16. S: Terminates processing. P: Modify the program so that the nesting level of **for** statement never exceeds 16.
3015	SYMBOL TABLE OVERFLOW	The number of symbols to be generated by the C compiler exceeds the limit. S: Terminates processing. P: Expand the limit using the **limits=symbol** option, or divide the file so that the number of symbols does not exceed the limit.
3016	INTERNAL LABEL OVERFLOW	The number of internal labels to be generated by the C compiler exceeds the limit. S: Terminates processing. P: Expand the limit using the **limits=symbol** option, or divide the file so that the number of internal labels does not exceed the limit.
3017	TOO MANY CASE LABELS	The number of **case** labels in one **switch** statement exceeds 511. S: Terminates processing. P: Ensure that the number of **case** labels does not exceed 511.

Error No.	Message	Explanation
3018	TOO MANY GOTO LABELS	The number of **goto** labels defined in one function exceeds 511. S: Terminates processing. P: Ensure that the number of **goto** labels defined in a function does not exceed 511.
3019	CANNOT OPEN SOURCE FILE	A source file cannot be opened. S: Terminates processing. P: Specify the correct file name.
3020	SOURCE FILE INPUT ERROR	A source or include file cannot be read. S: Terminates processing. P: Check that the file is not read protected.
3021	MEMORY OVERFLOW	The C compiler cannot allocate sufficient memory to compile the program. S: Terminates processing. P: Divide the file so that less memory is needed for compilation.
3022	SWITCH NEST TOO DEEP	The nesting level of **switch** statement exceeds 16. S: Terminates processing. P: Modify the program so that the nesting level of **switch** statement never exceeds 16.
3023	TYPE NEST TOO DEEP	The number of types (pointer, array, and function) that qualify the basic type exceeds 16. S: Terminates processing. P: Ensure that the number of types is less than or equal to 16.
3024	ARRAY DIMENSION TOO DEEP	An array has more than six dimensions. S: Terminates processing. P: Ensure that arrays have no more than six dimensions.
3025	SOURCE FILE NOT FOUND	A source file name is not specified in the command line. S: Terminates processing. P: Specify a source file name.
3026	EXPRESSION TOO COMPLEX	An expression is too complex. S: Terminates processing. P: Divide the expression into smaller units.

Error No.	Message	Explanation
3027	SOURCE FILE TOO COMPLEX	The nesting level of statements in the program is too deep or an expression is too complex. S: Terminates processing. P: Reduce the nesting level of statements or divide the expression.
3028	SOURCE LINE NUMBER OVERFLOW	The last source line number is greater than 65535. S: Terminates processing. P: Modify both the line count specified in the **#line** directive and the source program so that the last source line number is less than or equal to 65535.
3029	PHYSICAL LINE OVERFLOW	The number of physical lines (including the include files) exceeds 65535. S: Terminates processing. P: Divide the file so that the number of physical lines does not exceed 65535
3031	DATA SIZE OVERFLOW	The size of an array or a structure exceeds the limit. The limit is 65535 for a CPU/operating mode of 2600n, 2000n, 300hn, or 300, 1048575 for a CPU/operating mode of 2600a:20, 2000a:20, or 300ha:20, 16777215 for a CPU/operating mode of 2600a:24, 2000a:24, or 300ha:24, 268435455 for a CPU/operating mode of 2600a:28 or 2000a:28, and 4294967295 for a CPU/operating mode of 2600a:32 or 2000a:32. S: Terminates processing. P: Reduce the size of the array or the structure until it is less than or equal to the limit.
3033	SYMBOL TABLE OVERFLOW	The number of symbols used for debug information exceeds 65535. S: Terminates processing. P: Divide the file so that the number of symbols does not exceed 65535.

Error No.	Message	Explanation
3200	OBJECT SIZE OVERFLOW	The size of the object program exceeds the memory limit of 64 kbytes for a CPU/operating mode of 2600n, 2000n, 300hn, or 300, 1 Mbytes for a CPU/operating mode of 2600a:20, 2000a:20, or 300ha:20, 16 Mbytes for a CPU/operating mode of 2600a:24, 2000a:24, or 300ha:24, 256 Mbytes for a CPU/operating mode of 2600a:28 or 2000a:28, and 4 Gbytes for a CPU/operating mode of 2600a:32 or 2000a:32. S: Terminates processing. P: Divide the program so that the size of the object program does not exceed the limit.
3202	ILLEGAL STACK ACCESS	The local variable and temporary area, and the register save area are placed at an address that exceeds the limit value for the stack pointer (SP) or frame pointer (FP), or the parameter area is placed at an address that exceeds the limit value for the SP or FP. The offset limit from an SP or FP is 32 kbytes for a CPU/operating mode of 2600n, 2000n, 300hn, or 300, 512 kbytes for a CPU/operating mode of 2600a:20, 2000a:20, or 300ha:20, 8 Mbytes for a CPU/operating mode of 2600a:24, 2000a:24, or 300ha:24, 128 Mbytes for a CPU/operating mode of 2600a:28 or 2000a:28, and 2 Gbytes for a CPU/operating mode of 2600a:32 or 2000a:32. S: Terminates processing. P: Use the size of parameters and the size of local variables that are within the effective scope.
3204	TOO MANY SOURCE LINES FOR DEBUG	The number of execution statements used for debugging exceeds the limit. S: Terminates processing. P: Divide the file so that the number of execution statements does not exceed the limit.
3300	CANNOT OPEN INTERNAL FILE	An intermediate file internally generated by the C compiler cannot be opened. S: Terminates processing. P: Check that the intermediate file generated by the C compiler is not being used.

Error No.	Message	Explanation
3301	CANNOT CLOSE INTERNAL FILE	An intermediate file internally generated by the C compiler cannot be closed. S: Terminates processing. P: Check that the intermediate file generated by the C compiler is not being used.
3302	CANNOT INPUT INTERNAL FILE	An intermediate file internally generated by the C compiler cannot be read. S: Terminates processing. P: Check that the intermediate file generated by the C compiler is not being used.
3303	CANNOT OUTPUT INTERNAL FILE	An intermediate file internally generated by the C compiler cannot be written. S: Terminates processing. P: Increase the disk size.
3304	CANNOT DELETE INTERNAL FILE	An intermediate file internally generated by the C compiler cannot be deleted. S: Terminates processing. P: Check that the intermediate file generated by the C compiler is not being used.
3305	INVALID COMMAND PARAMETER **"option name"**	An invalid compiler option is specified. S: Terminates processing. P: Specify the correct option.
3306	INTERRUPT IN COMPILATION	An interrupt generated by a (CNTL) C command (from a standard input terminal) is detected during compilation. S: Terminates processing. P: Input the compile command again.
3307	COMPILER VERSION MISMATCH **"file name"**	The file version specified by the "file name" in the C compiler does not match other file versions. S: Terminates processing. P: Refer to the Install Guide for the installation procedure, and reinstall the C compiler.

Error No.	Message	Explanation
3320	COMMAND PARAMETER BUFFER OVERFLOW	The number of characters of a command line exceeds 256. S: Terminates processing. P: Ensure that the number of characters of the command line does not exceed 256.
3321	ILLEGAL ENVIRONMENT VARIABLE	The file name specification is illegal in setting environment variable CH38, or the number of characters of a pass exceeds 118. S: Terminates processing. P: Correctly set environment variable CH38.
3322	LACKING CPU SPECIFICATION	A CPU/operating mode is not specified. S: Terminates processing. P: Specify the CPU/operating mode using the **cpu** option or environment variable H38CPU.
3323	ILLEGAL ENVIRONMENT SPECIFIED	The specification of environment variable CH38TMP or H38CPU used by the C compiler is illegal. S: Terminates processing. P: Correctly set environment variable CH38TMP or H38CPU.
4000 to 4999	INTERNAL ERROR	An internal error occurs during compilation. S: Terminates processing. P: Report the error occurrence to your local Hitachi dealer.

Section 2 Error Messages Output for C Library Functions

Some library functions set error numbers to macro **errno** defined by the header file **<stddef.h>** in the C library function when an error occurs during the library function execution. Error messages corresponding to error numbers have already been defined and can be output. The following shows an example of a program which causes an error message output.

Example:

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

main()
{

    FILE *fp;

    fp=fopen("file","w");
    fp=NULL;
    fclose(fp);                        /* error occurred */-----①
    printf("%s\n",strerror(errno)) ; /*print error message */--②
}
```

Explanation:

① An error occurs because the file pointer value **NULL** is passed to the **fclose** function as an actual parameter. In this case, an error number is set in **errno**.

② If the error number is passed to the **strerror** function as an actual parameter, a pointer to the corresponding error message is returned. Specifying the character string to be output in the **printf** function outputs the error message.

C Library Function Error Messages

Error No.	Message and Explanation	Functions to Set Error Numbers
ERANGE	DATA OUT OF RANGE [An overflow occurs.]	**frexp, ldexp, modf, ceil, floor, fmod, strtol, atoi, atol, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf**
EDIV	DIVISION BY ZERO [Division by zero was performed.]	**div, ldiv**
ESTRN	TOO LONG STRING [The length of the character string exceeds 512 characters.]	**strtol, strtod, atoi, atol, atof**
PTRERR	INVALID FILE POINTER [The **NULL** pointer constant is specified as 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**
ECBASE	INVALID RADIX [An invalid radix was specified.]	**strtol, atoi, atol**
ETLN	NUMBER TOO LONG [The specified number exceeds 17 digits.]	**strtod, fscanf, scanf, sscanf, atof**
EEXP	EXPONENT TOO LARGE [The specified exponent exceeds three digits.]	**strtod, fscanf, scanf, sscanf, atof**
EEXPN	NORMALIZED EXPONENT TOO LARGE [The exponent exceeds three digits when the character string is normalized to the IEEE standard decimal format.]	**strtod, fscanf, scanf, sscanf, atof**
ENUM	NOT A NUMBER [Not a number was specified as an actual parameter.]	**frexp, ldexp, modf, ceil, fabs, floor, fmod**
EFLOATO	OVERFLOW OUT OF FLOAT [The **float** data followed by **f** suffix is out of range (overflows).]	**strtod, fscanf, scanf, sscanf, atof**
EFLOATU	UNDERFLOW OUT OF FLOAT [The **float** data followed by **f** suffix is out of range (underflows).]	**strtod, fscanf, scanf, sscanf, atof**

Error No.	Message and Explanation	Functions to Set Error Numbers
EOVER	FLOATING POINT OVERFLOW [The constant is out of **double** data range (overflow).]	**strtod, fscanf, scanf, sscanf, atof**
EUNDER	FLOATING POINT UNDERFLOW [The constant is out of **double** data range (underflow).]	**strtod, fscanf, scanf, sscanf, atof**
NOTOPN	FILE NOT OPEN [The file is not open.]	**fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen**
EBADF	BAD FILE NUMBER [An output function was issued for an input file or output function is issued for input file.]	**fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite**
ECSPEC	ERROR IN FORMAT [An erroneous format was specified for an in input/output function using format.]	**fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror**

Appendix

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

This section shows the implementation dependent specifications of the C compiler that are not included in the C language specifications (in ANSI standard for the programming).

A.1 C Compiler Language Specifications

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 parameter 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 internal-identifier characters not used for external linkage	The first 31 characters are valid
Number of valid external-identifier characters used for external linkage	The first 31 characters are valid
Lowercase and uppercase character distinction in external identifiers used for external linkage	Lowercase characters are distinguished from uppercase characters

Note: Two different identifiers in which the first 31 characters are the same are considered to be identical.

Example:
(a) longnameabcdefghijklmnopqrstuvwx;
(b) longnameabcdefghijklmnopqrstuvwy;
Identifiers (a) and (b) are indistinguishable because the first 31 characters are the same.

A.1.4 Characters

Table A-4 Character Specifications

Item	C Compiler Specification
Elements of a source character set and an execution environment character set	ASCII characters are used for both sets.
Shift state used for encoding multiple-byte characters	The 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 the source character set that appears either in a character constant or a string literal and the execution environment character set	ASCII characters are used for both sets.
Integer character constants including characters and extended representation that are not specified by the language	Characters and extended representations that are not specified by the language are not supported.
Character constant of two or more characters or a wide character constant of two or more multiple-byte characters	The upper two characters of the character constant are valid, and the most significant character of the wide character constant is valid. If a wide character constant of more than one character is specified, a warning error message is output.
locale specifications used to convert multiple-byte characters to wide character	**locale** is not supported.
Simple **char** having the same value range as **signed char** or **unsigned char**	The same range as **signed char**

A.1.5 Integer

Table A-5 Integer Specifications

Item	C Compiler Specification
Integer type representation and its values	Listed in Table A-6.
Values when a smaller signed integer type or an unsigned integer type is converted into a signed integer type of the same size (when an integer is too large to be converted into a signed integer)	The lower one or two bytes of the integer is used as the conversion result.
The result of bit-wise 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 a negative signed integer type	The sign bit is unchanged by the shift operation.

Table A-6 Integer Types and Corresponding Data Ranges

Type	Value Range	Data Size
char	–128 to 127	1 byte
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	–32768 to 32767	2 bytes
unsigned int	0 to 65535	2 bytes
long	–2147483648 to 2147483647	4 bytes
unsigned long	0 to 4294967295	4 bytes

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 its values	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 r representation, conversion specifications, and operation specifications). Table A-8 shows the limits for representing floating-point numbers.
Data converted from **double** or **long double** to **float**	
Internal representation of floating-point data	

Table A-8 Limits on Floating-Point Numbers

	Limit	
Item	**Decimal ***	**Internal Representation**
Maximum **float**	3.4028235677973364e+38f (3.4028234663852886e+38f)	`7f7fffff`
Positive minimum **float**	7.0064923216240862e–46f (1.4012984643248171e–45f)	`00000001`
Maximum **double** or **long double**	1.7976931348623158e+308 (1.7976931348623157e+308)	`7fefffffffffffff`
Positive minimum **double** or **long double**	4.9406564584124655e–324 (4.9406564584124654e–324)	`0000000000000001`

Note: The decimal limit is either a non-zero minimum value or a maximum value that is not infinite. Values within ( ) indicate theoretical values.

A.1.7 Arrays and Pointers

Table A-9 Array and Pointer Specifications

Item	C Compiler Specification
Integer type required for maximum array size **(size_t)**	**unsigned int** — H8S/2600 in normal mode — H8S/2000 in normal mode — H8/300H in normal mode — H8/300
	unsigned long — H8S/2600 in advanced mode — H8S/2000 in advanced mode — H8/300H in advanced mode
Conversion from a pointer type to an integer type (Pointer type size ≥ Integer type size)	The lower byte of a pointer type is used.
Conversion from a pointer type to an integer type (Pointer type size < Integer type size)	Expanded with zeros
Conversion from an integer type to a pointer type (Integer type size ≥ Pointer type size)	The lower byte of an integer type is used.
Conversion from an integer type to a pointer type (Integer type size < Pointer type size)	Expanded with zeros
Integer type required for holding pointer differences between members in the same array **(ptrdiff_t)**	**int** — H8S/2600 in normal mode — H8S/2000 in normal mode — H8/300H in normal mode — H8/300
	long — H8S/2600 in advanced mode — H8S/2000 in advanced mode — H8/300H in advanced mode

A.1.8 Register

Table A-10 Register Specifications

Item	C Compiler Specification
Registers to which register variables can be allocated	H8S/2600, H8S/2000, H8/300H with optimization: (ER3), ER4, ER5, ER6 * H8/300H without optimization: (ER3), ER4, ER5 *
	H8/300 with optimization: (R3), R4, R5, R6 * H8/300 without optimization: (R3), R4, R5 *
Type of register variables that can be allocated to registers (H8S/2600, H8S/2000, H8/300H)	**char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long,** and **float** types, and pointers
Type of register variables that can be allocated to registers (H8/300)	**char, unsigned char, short, unsigned short, int,** and **unsigned int** types, and pointers

Note: Register variables are not allocated when the **noregexpansion** option is specified for the register in parentheses.

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 setting a union member and referencing a union member using another member whose data type is different	Reference is possible but the referred value is not guaranteed.
Structure member alignment	Structures consisting of **char** members are aligned in 1-byte units. Structures consisting of any other member are aligned in 2-byte units. See section 2.2, Aggregate Type Data, in part II.
Sign of an **int** bit field	Assumed to be a **signed int**.
Allocation order of bit fields in **int** type	Beginning from the high order bit to low order bit
Result when a bit field has been allocated in an **int** type and the next bit field to be allocated is larger than the remaining **int** type	The next bit field is allocated to the next **int** area.
Type specifier allowed for bit field	**char, unsigned char, short, unsigned short, int,** and **unsigned int**
Integer type describing enumeration	**int**

Note: See section 2.2, Bit Fields, in part II for details about allocating bit fields.

A.1.10 Modifier

Table A-12 Modifier Specifications

Item	C Compiler Specification
volatile data access type	Not specified.

A.1.11 Declarations

Table A-13 Declaration Specifications

Item	C Compiler Specification
Number of types that can qualify basic types (arithmetics, structure, and union)	Up to 16 types can be specified.

(a) Example of counting the number of types that qualify the basic types

Examples:

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

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

A.1.12 Statement

Table A-14 Statement Specifications

Item	C Compiler Specification
The number of **case** labels specified by a **switch** statement	Up to 511 labels

A.1.13 Preprocessor

Table A-15 Preprocessor Specifications

Item	C Compiler Specification
Correspondence between a single character constant and an execution environment character set in the conditional compilation	The character constant in the preprocessor directive matches the execution environment character set.
Reading an include file	The file within < > is read from a path specified by the **include** option. (Default: The path specified by environment variable CH38.)
Supporting an include file with its name enclosed in a pair of double quotation marks	The C compiler supports include files with their names 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 a path specified by the **include** option.
Source file character string correspondence (blank character in a character string after macro expansion)	Strings of blanks are expanded as one blank character.
#pragma directive operation	**#pragma abs8, #pragma abs16, #pragma asm, #pragma endasm, #pragma indirect, #pragma inline, #pragma interrupt, #pragma section, #pragma abs8 section, #pragma abs16 section, #pragma indirect section, #pragma regsave,** and **#pragma noregsave** are supported.
Value of _ _DATE_ _, _ _TIME_ _	Dependent on the host-machine timer when the compilation starts.

A.2 C Library Function Specifications

This section explains the specifications for C library functions declared in standard include files. Refer to the include file for the actual macro names defined in a standard include file.

A.2.1 stddef.h

Table A-16 stddef.h Specifications

Item	C Compiler Specification
Value of macro **NULL**	The pointer value 0 is set to **void**.
Contents of macro **ptrdiff_t**	**int** type — H8S/2600 in normal mode — H8S/2000 in normal mode — H8/300H in normal mode — H8/300
	long type — H8S/2600 in advanced mode — H8S/2000 in advanced mode — H8/300H in advanced mode

A.2.2 assert.h

Table A-17 assert.h Specifications

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

(a) The following message is output when the expression is 0 for **assert** (expression):
ASSERTION FAILED: <expression> FILE <file name>, LINE <line number>

A.2.3 ctype.h

Table A-18 ctype.h Specifications

Item	C Compiler Specification
The character set is inspected by the **isalnum, isalpha, iscntrl, islower, isprint,** and **isupper** functions	Character set represented by the **unsigned char** type. Table A-19 shows the character set that results in a true return value.

Table A-19 Set of Characters that Return True Values

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	The C compiler does not support a mathematical function whose input parameter is out of the range.
Is **errno** set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function?	The C compiler does not support a mathematical function in which an underflow error occurs.
Does a range error occur if the second parameter in the **fmod** function is 0	A range error occurs.

Note: **math.h** defines macro name **EDOM, ERANGE** which indicates a C library function error number.

A.2.5 stdio.h

Table A-21 stdio.h Specifications

Item	C Compiler Specification
Does the last line of the input text require a line feed character indicating end?	Not specified. Depends on the low-level interface routine specifications.
Is the blank character immediately before the carriage return character read?	
Number of **NULL** characters added to data written in the binary file	
Initial value of file position specifier in the addition mode	
Is a file data lost following text file output?	
File bufferring 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
Input data representation of the %p format conversion in the **fscanf** function, the meaning of conversion character "–" in the **fscanf** function	Hexadecimal representation If "–" is not the first or last character or "–" does not follow "^", the C compiler indicates the previous character and following characters.
Value of **errno** specified by the **fgetpos** or **ftell** function	The **fgetpos** function is not supported. The **ftell** function does not specify the errno value. The errno value is determined depending on the low-level interface routine.
Output format of messages generated by the **perror** function	See (a) below for the output message format.
calloc, malloc, or **realloc** function operation when the size is 0	The 0-byte area is allocated.

(a) Messages generated by the **perror** function follow this format:
<character string> : <error message corresponding to the error number indicated by **error**>

(b) Table A-22 shows the format used to indicate infinity and not-a-number for floating-point numbers when using the **printf** or **fprintf** function.

Table A-22 Infinity and Not-a-Number

Value	Format
Positive infinity	+ + + + + +
Negative infinity	– – – – – –
Not a number	* * * * * *

A.2.6 string.h

Table A-23 string.h Specifications

Item	C Compiler Specification
Error message returned by the **strerror** function	See part IV, section 2, Standard Library Error Messages.

A.2.7 Unsupported Library

Table A-24 lists libraries in the C language specifications not supported by the C compiler.

Table A-24 Libraries not Supported by the C Compiler

Header File	Library Name
math.h	acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, log, log10, pow, sqrt
signal.h	signal.h, signal, raise
stdio.h	remove, rename, tmpfile, tmpnam
stdlib.h	abort, exit, getenv, onexit, system
time.h	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 standard IEEE format. This section explains this standard.

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

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

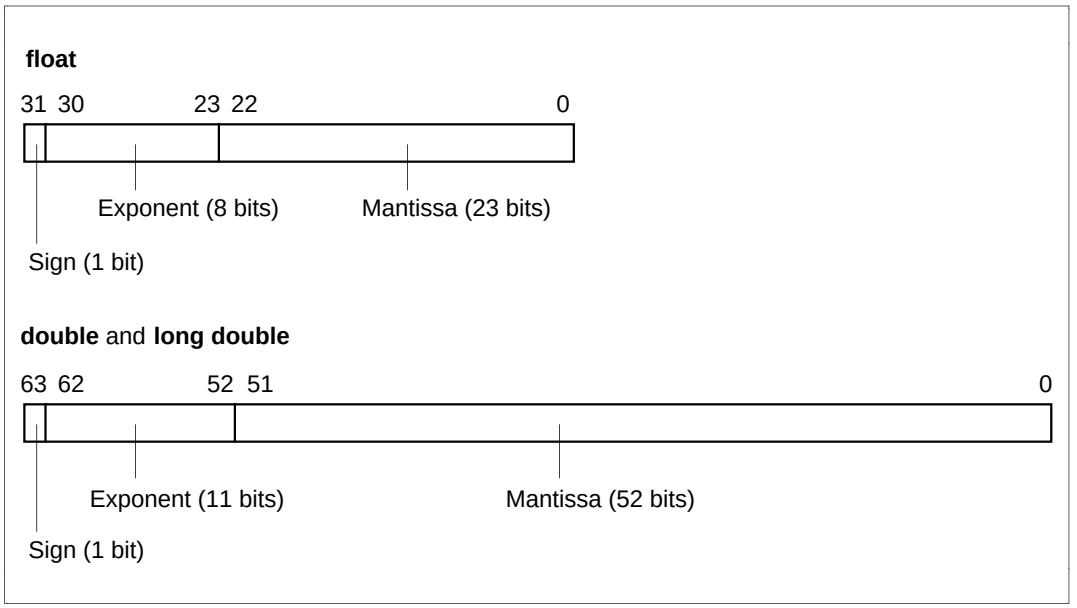


Figure A-1 Structure for the Internal Representation of Floating-Point Numbers

The elements of the structure have the following meanings.

(i) Sign
This indicates the sign of a floating-point number. Positive and negative are represented by 0 and 1, respectively.

(ii) Exponent
This indicates the exponent of a floating-point number as a power of two.

(iii) Mantissa
This determines the significant digits of a floating-point number.

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

(i) Normalized Number
The exponent is not 0 or the maximum. A normalized number represents a general real number.

(ii) 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.

(iii) Zero
The exponent and mantissa are both 0. Zero represents the value 0.0.

(iv) Infinity
The exponent is the maximum and mantissa is 0.

(v) 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, ∞/∞, and ∞ – ∞).

Table A-25 shows the conditions used to determine values represented by floating-point numbers.

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-25 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

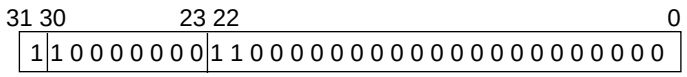
A.3.2 float

float is internally represented as a 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 this value 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 $2^{23} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{23}) is 1 and a decimal point follows it.

Value represented by a normalized number:
$(-1)^{<sign>} \times 2^{<exponent> - 127} \times (1 + <mantissa> \times 2^{-23})$

Example:

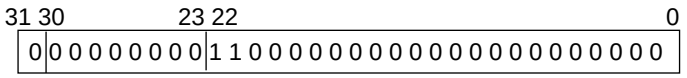


Sign: –
Exponent: $10000000_{(2)} - 127 = 1$ ($_{(2)}$ indicates binary data throughout this manual.)
Mantissa: $1.11_{(2)} = 1.75$
Value: $-1.75 \times 2^1 = -3.5$

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 $2^{23} - 1$. For an actual mantissa, it is assumed that a highest order bit (2^{23}) is 0 and a decimal point follows it.

Value represented by a denormalized number:
$(-1)^{<sign>} \times 2^{-126} \times (<mantissa> \times 2^{-23})$

Example:



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

Zero: The sign bit is either 0 (positive) or 1 (negative), (i.e., there are two distinct zero values, +0.0 and –0.0). The exponent and mantissa are 0. Both +0.0 and –0.0 represent 0.0. See appendix A.3.4, Floating-Point Operation Specification, for differences in each operation dependent on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) (i.e., $+\infty$ and $-\infty$ can be represented). 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: The sign of a not-a-number is arbitrary and the value of the mantissa is not limited (except that it may not be equal to 0).

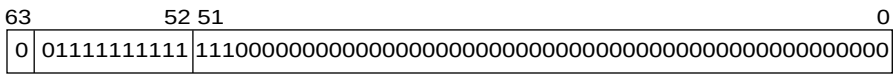
A.3.3 **double** and **long double**

A double or **long double** is represented as a 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 this value 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 $2^{52} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{52}) is 1 and a decimal point follows it.

Value represented by a normalized number:
$(-1)^{<sign>} \times 2^{<exponent> - 1023} \times (1 + <mantissa> \times 2^{-52})$

Example:

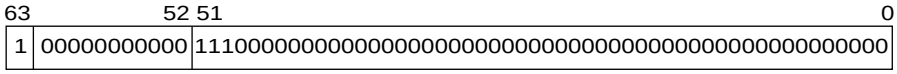


Sign: +
Exponent: $1111111111_{(2)} - 1023 = 0$
Mantissa: $1.111_{(2)} = 1.875$
Value: $1.875 \times 2^0 = 1.875$

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 $2^{52} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{52}) is 0 and a decimal point follows it.

Value represented by a denormalized number:
$(-1)^{<sign>} \times 2^{-1022} \times (<mantissa> \times 2^{-52})$

Example:



Sign: –
Exponent: –1022
Mantissa: $0.111_{(2)} = 0.875$
Value: 0.875×2^{-1022}

Zero: The sign bit is either 0 (positive) or 1 (negative) (i.e., there are two distinct zero values + 0.0 and –0.0). 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) (i.e., $+\infty$ and $-\infty$ can be represented). 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: The sign of a not-a-number is arbitrary and the value of the mantissa is not limited (except that it may not be equal to 0).

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 either the internal representation of floating-point numbers generated during C compiler or standard library function processing.

Arithmetic Operation Specifications:

(i) 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:

① The result is rounded to the nearest internally representable floating-point number.

② 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.

(ii) Handling of Overflow and Underflow

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

① For an overflow, positive or negative infinity is used depending on the sign of the result.

② For an underflow, positive or negative zero is used depending on the sign of the result.

③ An invalid operation is assumed when: (i) infinity is added to infinity and each has a different sign; (ii) infinity is subtracted from infinity and each 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 a not-a-number.

④ In any case, the variable errno is set to the error number corresponding to the error. See part IV, Error Messages, section 2, C Library Error Messages, for the error number.

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.

(iii) Special Value Operations

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

① The result is positive zero if positive zero and negative zero are added.

② If zero is subtracted from zero and both zeros have the same sign, the result is a positive zero.

③ The operation result is always a not-a-number if one or both operands are not-a-numbers.

④ A positive zero is equal to a negative zero in relational operations.

⑤ If one or both operands are not-a-numbers in a relational 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 an internal representation. The conversion between decimal representation and internal representation of ASCII string literal floating-point numbers using library functions is also explained.

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

① For normalized **float**
$0 \le M \le 10^9 - 1$
$0 \le N \le 99$

② For normalized **double** and **long double**
$0 \le M \le 10^{17} - 1$
$0 \le N \le 999$

An overflow or underflow occurs if a floating-point number in a decimal representation cannot be normalized. If a floating-point number in a 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 compile time and the variable errno is set equal to the corresponding error number at run time.

To convert a floating-point number from a decimal representation to a normalized decimal representation, the length of the original ASCII string literal 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 an internal representation to a normalized decimal representation. According to a specified format, the result is then converted to an ASCII string literal.

(ii) Conversion between Normalized Decimal Representation and Internal Representation

If the exponent of a floating-point number to be converted between a decimal representation and an 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:

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

② For **double** and **long double:** $0 \le M \le 10^{17} - 1,\ 0 \le N \le 27$

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

b) Conversion and Rounding Error

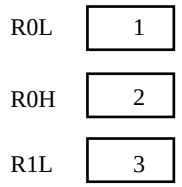
The difference between, (i) the error occurring when the exponent outside the proper range is converted, and (ii) the error occurring when the value is precisely rounded, does not exceed the result of multiplying the lowest significant digit by 0.47. If an exponent outside the proper range is converted,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

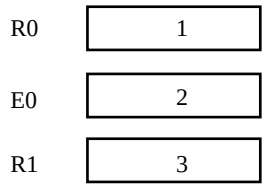
Appendix B Parameter Allocation Examples

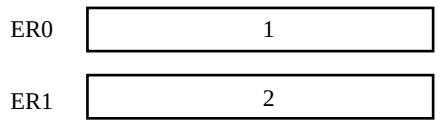
B.1 H8S/2600, H8S/2000, and H8/300H Register Parameters

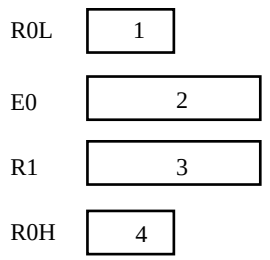
(cpu = 2600a, cpu = 2600n, cpu = 2000a, cpu = 2000n, cpu = 300ha, cpu = 300hn)

Example 1: Register parameters are allocated to registers ER0 and ER1 depending on the order of declaration.

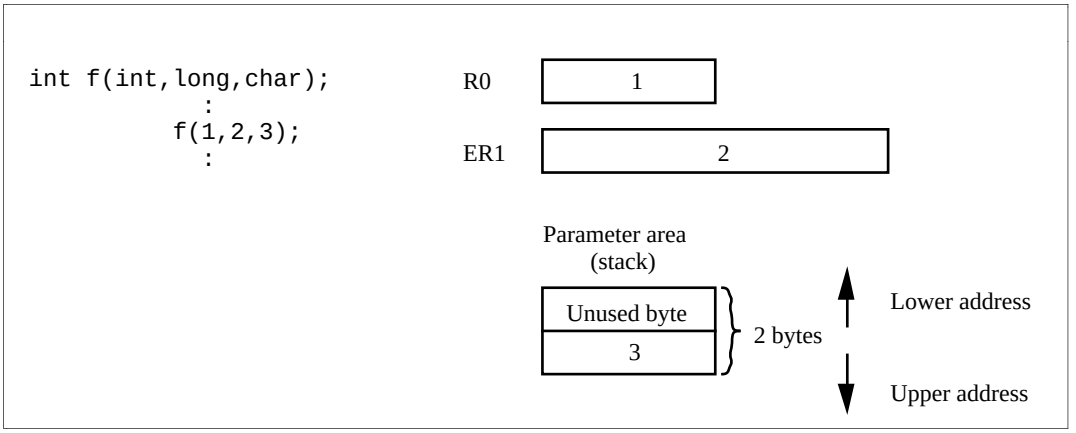
①
```
int f(char,char,char);
          :
        f(1,2,3);
          :
```


②
```
int f(int,int,int);
          :
        f(1,2,3);
          :
```


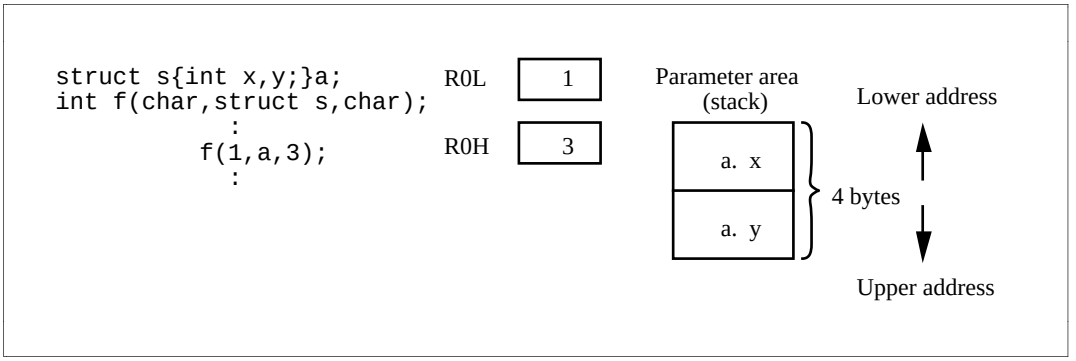
③
```
int f(long,long);
          :
        f(1,2);
          :
```


④
```
int f(char,int,int,char);
          :
        f(1,2,3,4);
          :
```


Example 2: Parameters that could not be allocated to registers ER0 and ER1 are allocated to the stack area as shown below. If **char**-type parameter is allocated to a parameter area on the stack, an unused byte is allocated to the lower address.

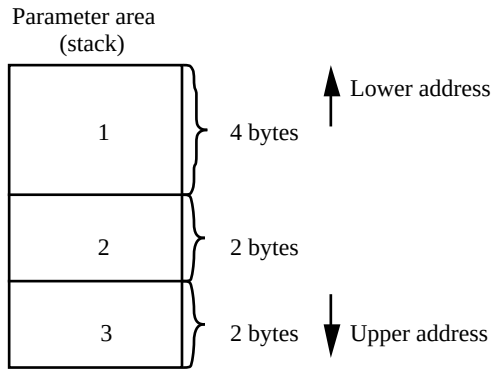


Example 3: Parameters that have a type which cannot be allocated to registers ER0 and ER1 are allocated to the stack area.

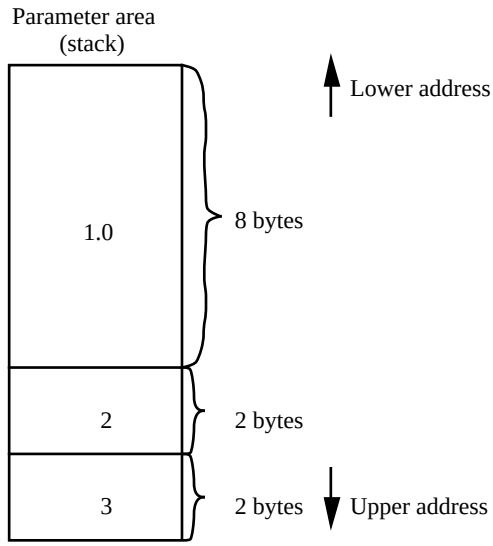


Example 4: If a function with a variable number of parameters is specified by prototype declaration, parameters which do not have a corresponding type in the declaration and in the immediately preceding parameter are allocated to a stack.

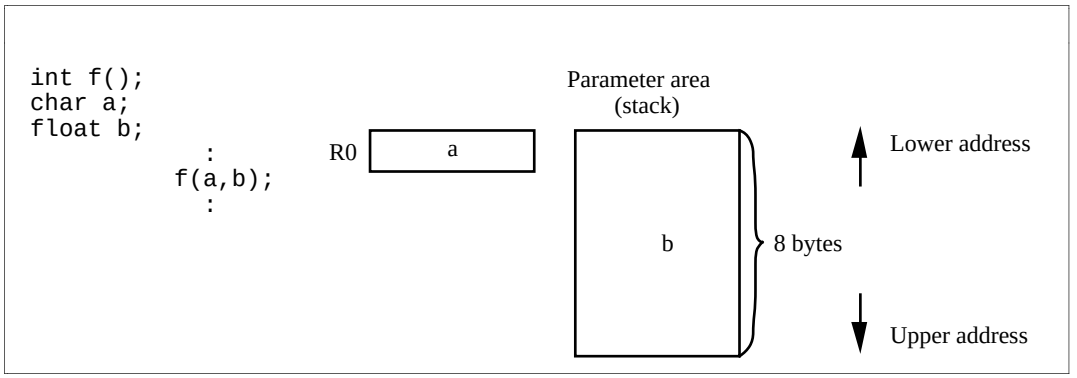
①
```
int f(long,...);
      :
f(1,2,3);
      :
```



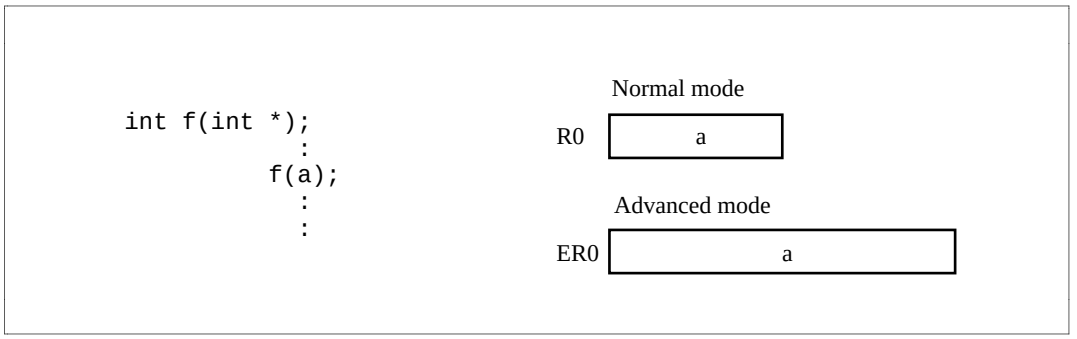
②
```
int f(double,int,...);
      :
f(1.0,2,3);
      :
```



Example 5: If no prototype is declared, **char** and **float** types are extended to **int** and **double** types, respectively.

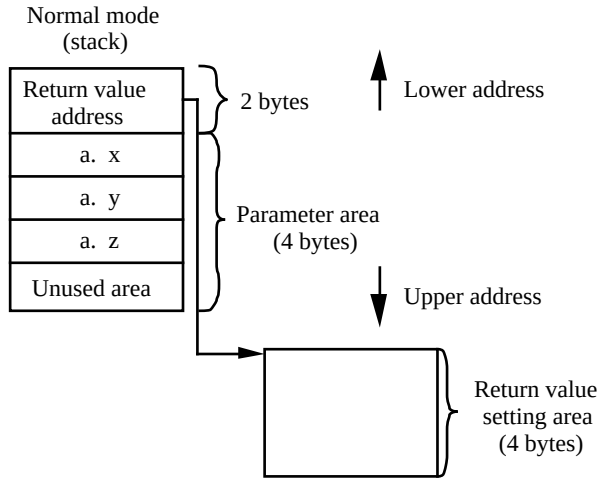


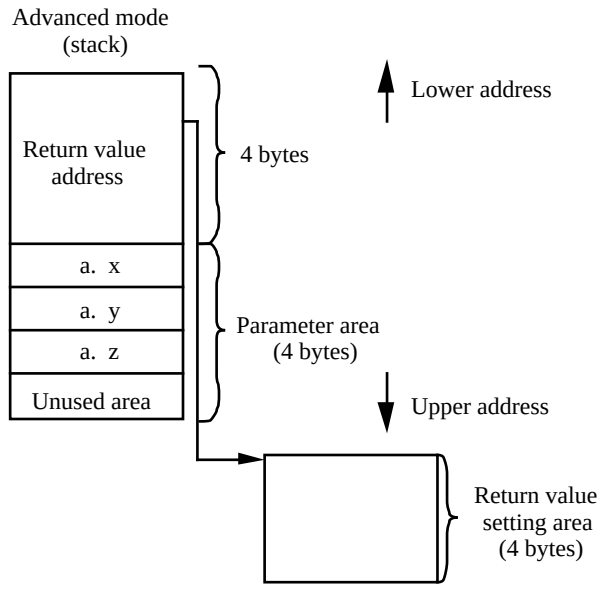
Example 6: Pointer-type parameters are allocated to the 2-byte area in normal mode and the 4-byte area in advanced mode.



Example 7: If a parameter returned by a function is a structure or exceeds 4 bytes, a return value address is specified just before the parameter area. If the structure size is an odd number of bytes, an unused area byte is inserted in the parameter area.

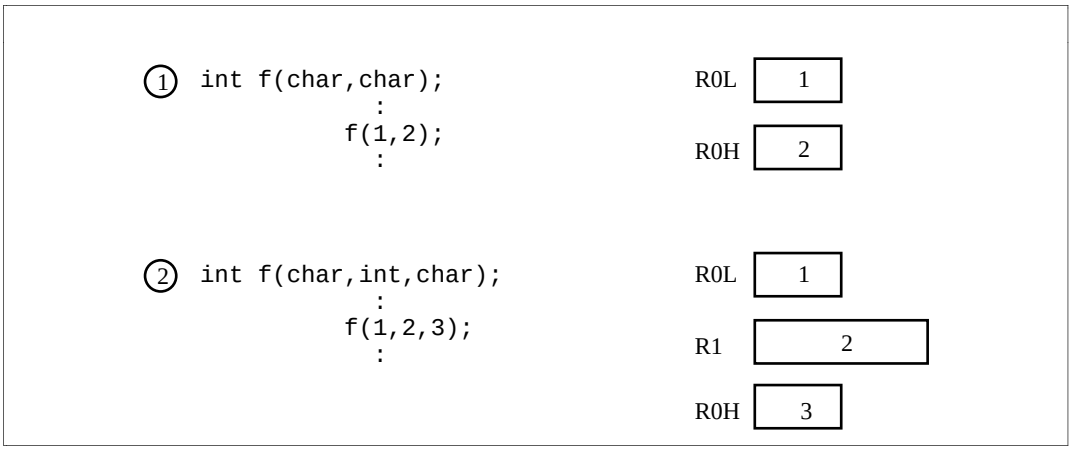
```
struct s{char x,y,z;}a,b;
float f(struct s);
          :
        f(a);
          :
          :
```



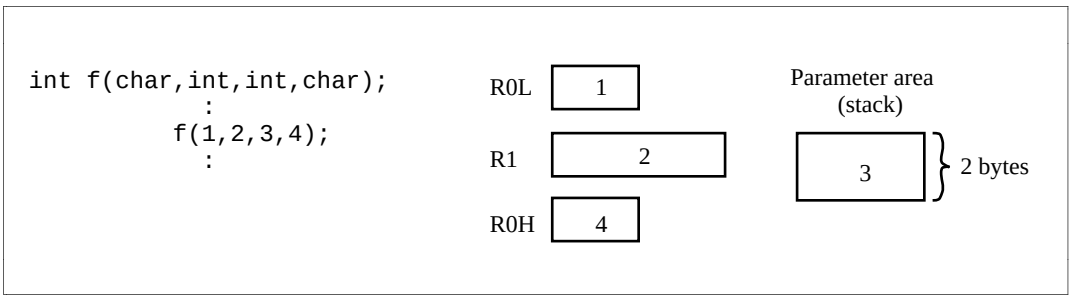


B.2 H8/300 Register Parameter (cpu = 300)

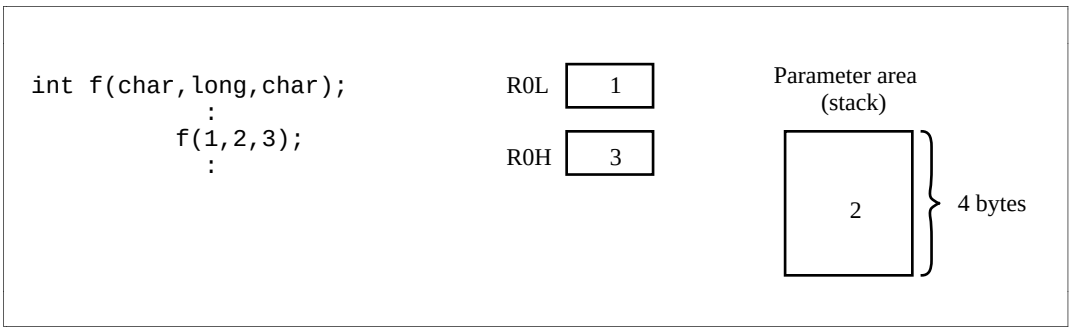
Example 1: Register parameters are allocated to registers R0 and R1 depending on the order of declaration.



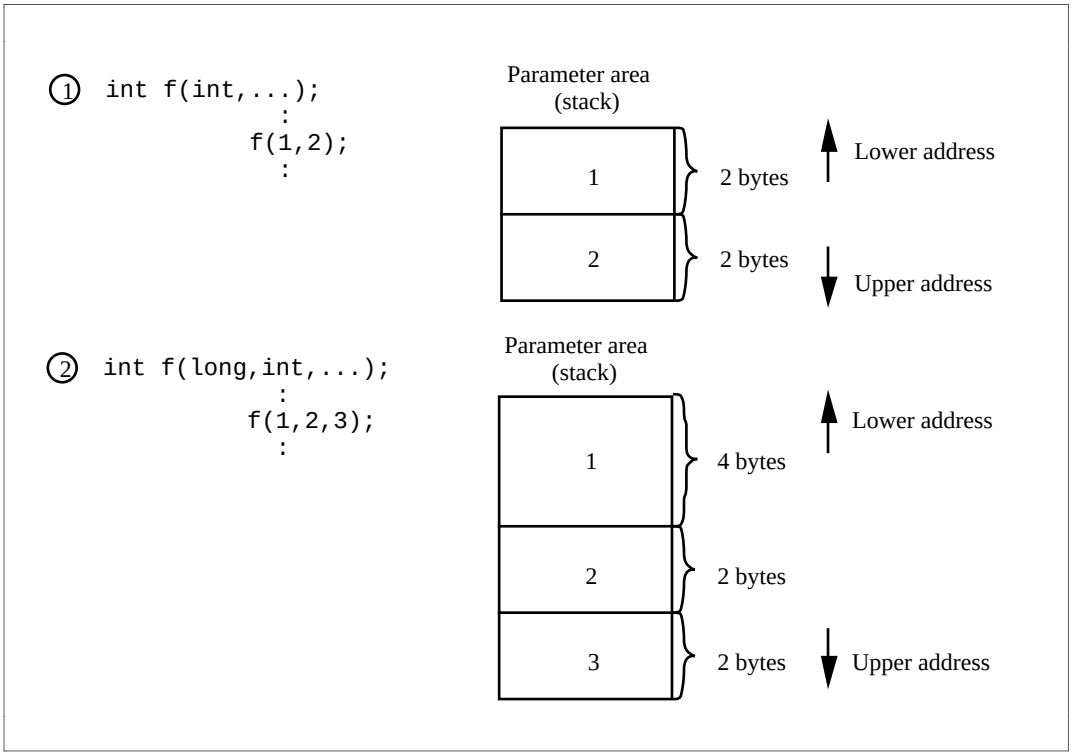
Example 2: Parameters that could not be allocated to registers R0 and R1 are allocated to the stack area as shown below.



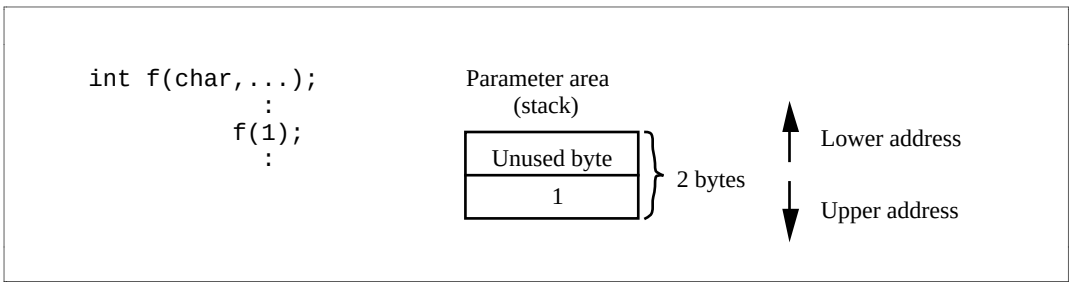
Example 3: Parameters that have a type which cannot be allocated to registers R0 and R1 are allocated to the stack area.



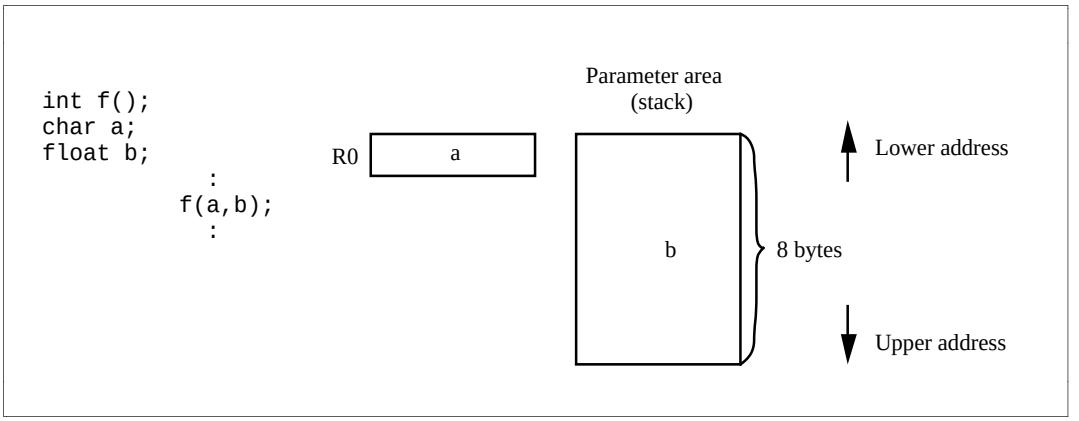
Example 4: If a function whose number of parameters changes is specified by prototype declaration, the parameters which do not have a corresponding type in the declaration and in the immediately preceding parameters are allocated to a stack.



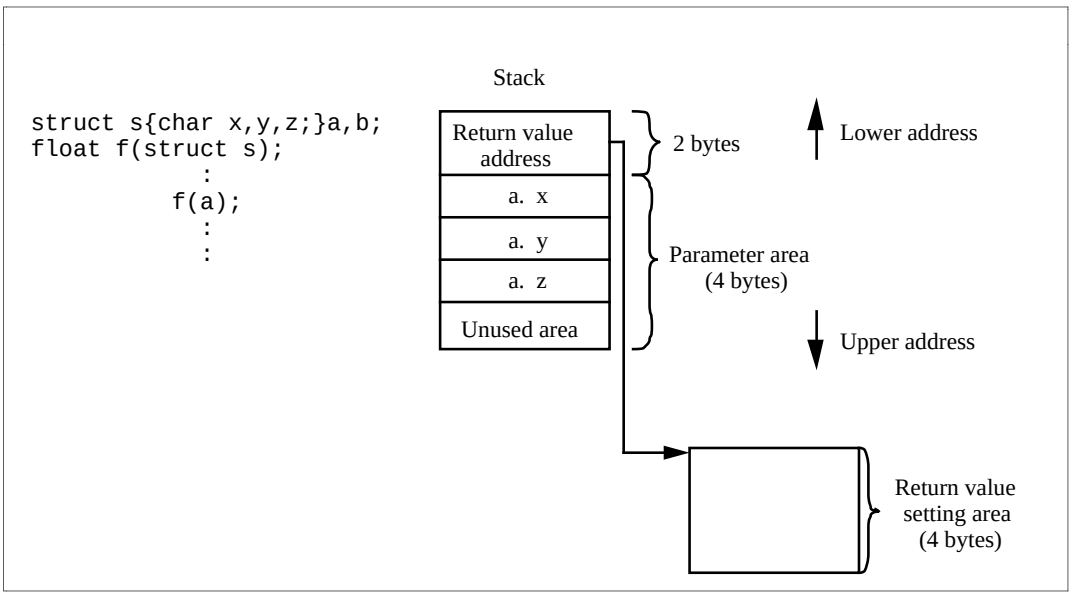
Example 5: If **char**-type parameter is allocated to a parameter area on a stack, an unused byte is inserted in a lower address of the parameter area.



Example 6: If no prototype is declared, **char** and **float** types are extended to **int** and **double** types, respectively.



Example 7: If a value returned by a function exceeds 2 bytes, a return value address is specified just before the parameter area. If the structure size is an odd number of bytes, an unused area byte is inserted in the parameter area.



Appendix C Usage of Registers and Stack Area

C.1 H8S/2600, H8S/2000, and H8/300H in Advanced Mode

(cpu = 2600a, cpu = 2000a, cpu = 300ha)

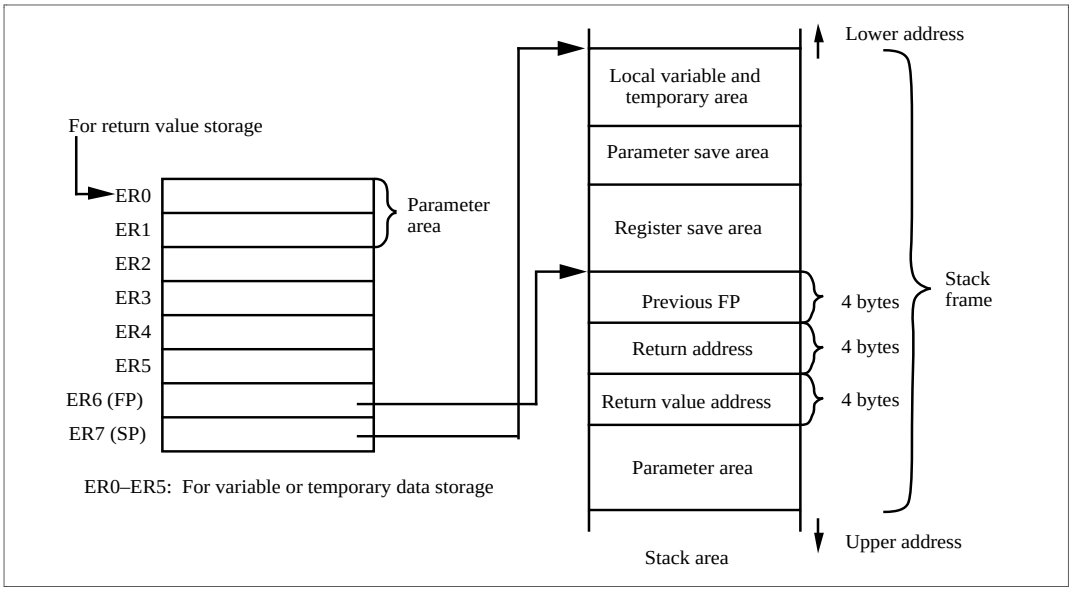


Figure C-1 Usage of Registers and Stack Area at Non-Optimization (H8S/2600, H8S/2000, and H8/300H in Advanced Mode)

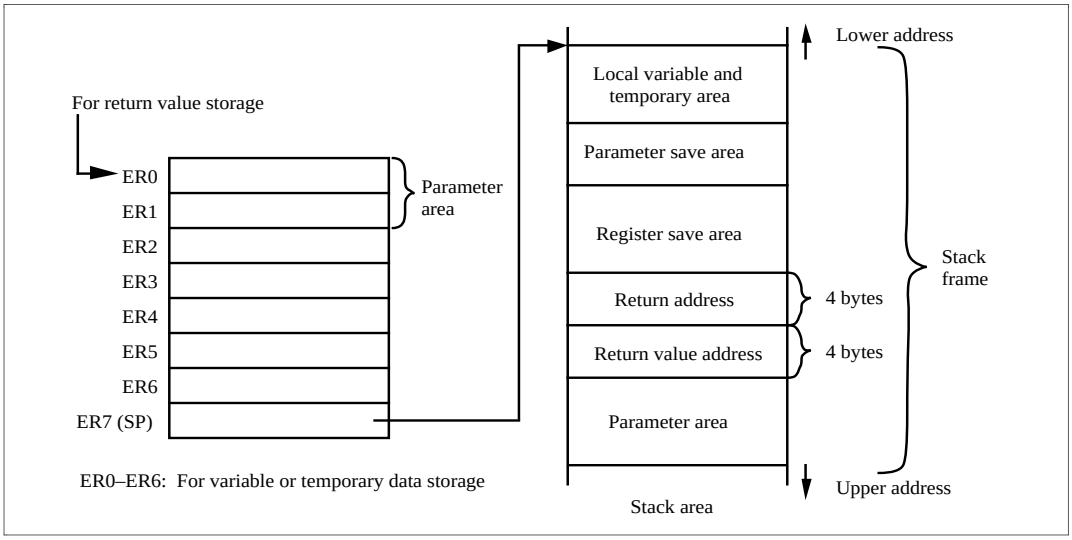


Figure C-2 Usage of Registers and Stack Area at Optimization (H8S/2600, H8S/2000, and H8/300H in Advanced Mode)

C.2 H8S/2600, H8S/2000, and H8/300H in Normal Mode

(cpu = 2600n, cpu = 2000n, cpu = 300hn)

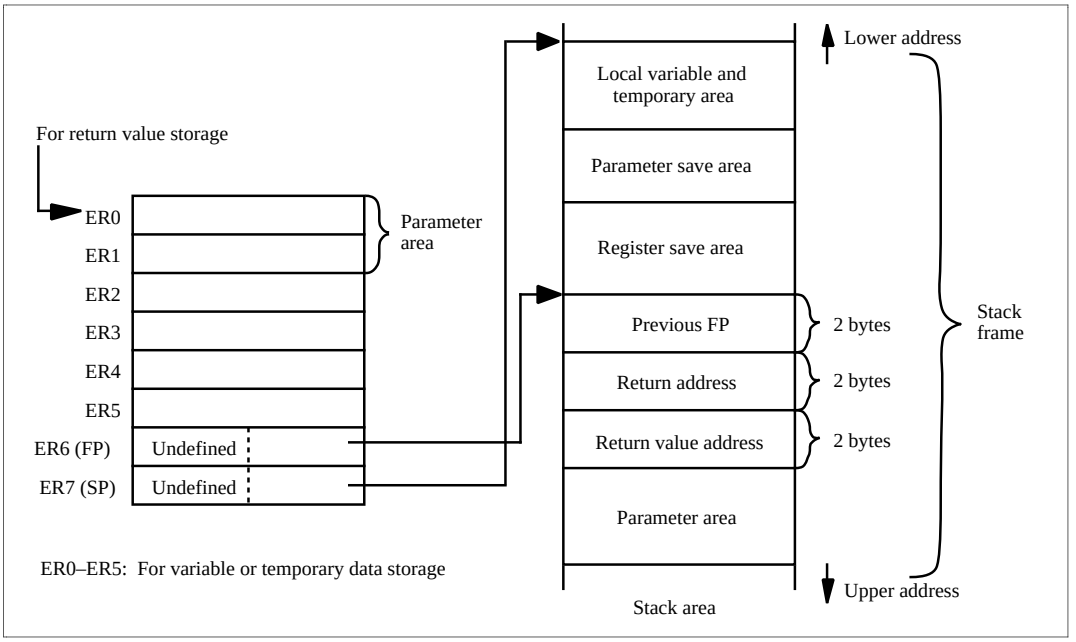


Figure C-3 Usage of Registers and Stack Area at Non-Optimization (H8S/2600, H8S/2000, and H8/300H in Normal Mode)

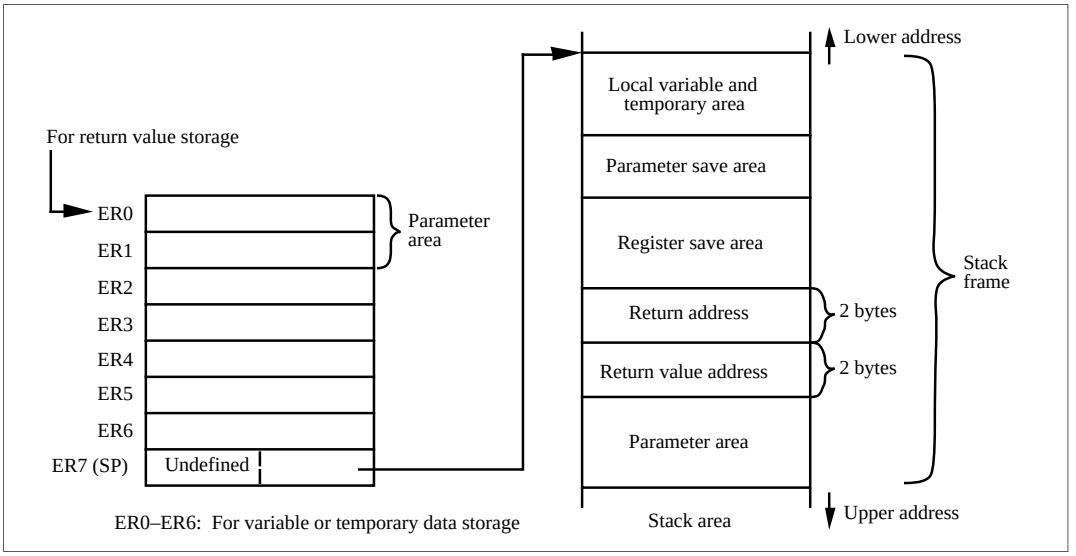


Figure C-4 Usage of Registers and Stack Area at Optimization (H8S/2600, H8S/2000, and H8/300H in Normal Mode)

C.3 H8/300 (cpu = 300)

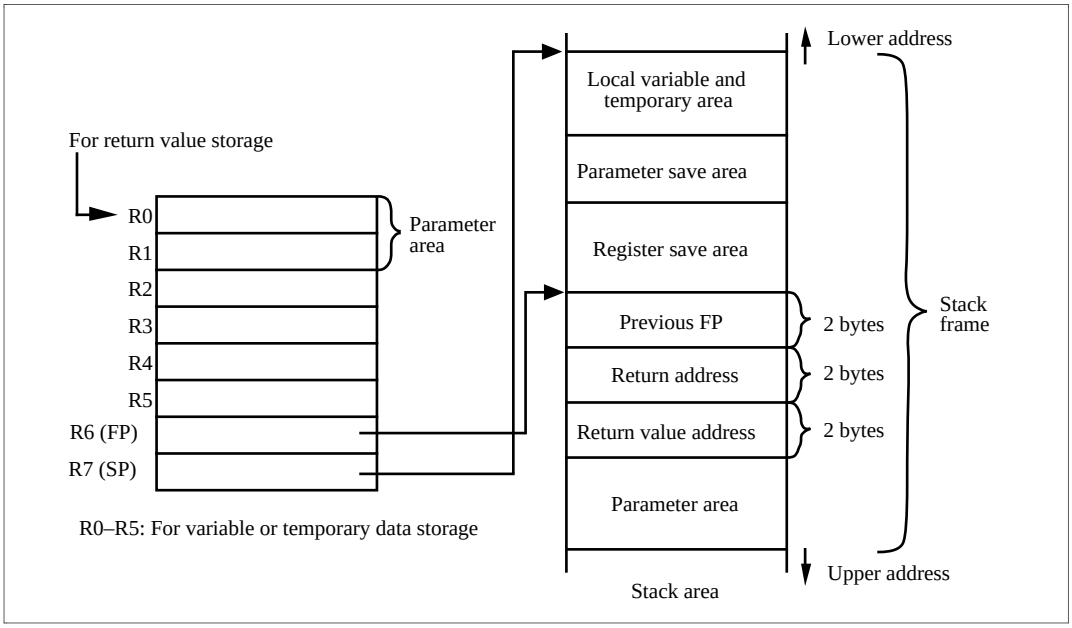


Figure C-5 Usage of Registers and Stack Area at Non-Optimization (H8/300)

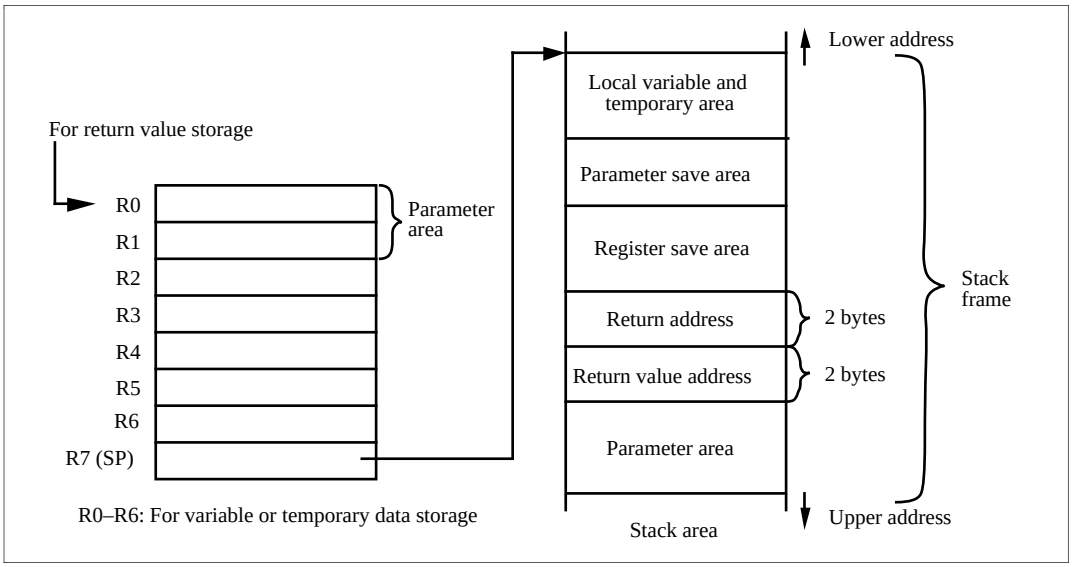


Figure C-6 Usage of Registers and Stack Area at Optimization (H8/300)

Appendix D Creating Termination Function

D.1 Creating a Library "onexit" Function

This section describes how to create a library **onexit** function that defines termination routines. The **onexit** function defines a function address, 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 more than one, NULL is returned. Otherwise, a value other than NULL is returned. In the following example, an address in which a function is defined is returned. An example of the **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++)       /* Checks if the same function
                                               has been defined */
      if(_onexit_buf[i]==f)
            return NULL ;
     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 */
          return f;
     }
}
```

D.2 Creating an "exit" Function

This section describes how to create an **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. This is because terminating 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.

Example:

```
#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(void);
extern void exit(int);

void exit(code)
int code ;
{

      int i;

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

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

      longjmp(_init_env, 1) ;            /* Returns to the environment saved by
                                            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 using 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 an "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 an abort routine in which an error message is output to the standard output device, closes all files, enters an endless loop, and waits for reset.

Example:

```
#include <stdio.h>
extern void abort(void);
extern void _CLOSEALL(void);
void abort()
{

      printf("program is abort !!\n");       /* Outputs message      */

      _CLOSEALL();                           /* Closes all files     */

      while(1);                              /* Enters endless loop  */

}
```

Appendix E Examples of a Low-Level Interface Routine

```
/************************************************************************/
/*                              lowsrc.c:                               */
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*        H8S and H8/300-series simulator debugger interface routine    */
/*     - Only standard I/O files (stdin, stdout, stderr) are supported -*/
/************************************************************************/
#include <string.h>

/* file number */

#define STDIN  0                    /* Standard input (console)          */
#define STDOUT 1                    /* Standard output (console)         */
#define STDERR 2                    /* Standard error output (console)   */
#define FLMIN  0                    /* Minimum file number               */
#define FLMAX  3                    /* Maximum number of files           */

/* file flag */

#define O_RDONLY 0x0001             /* Read only                         */
#define O_WROMLY 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 */

#if __CPU__==3| __CPU_ _==5| __CPU__==7 /* __CPU__==3:300ha, 5:2600a, 7:2000a  */
#define HEAPSIZE 2064
#else
#define HEAPSIZE 2056
#endif

/************************************************************************/
/* Declaration of reference function                                    */
/* Reference to assembly program in which the simulator debugger inputs or    */
/* outputs characters to the console                                    */
/************************************************************************/
extern void charput(char);          /* One character input               */
extern char charput(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 {
              short  dummy ;        /* Dummy for 2-byte boundary         */
              char heap[HEAPSIZE]; /* Declaration of the area managed by sbrk */
             } heap_area ;
static char *brk=(char *)&heap_area; /* End address of area assigned by sbrk  */
```

```
/*************************************************************************/
/*    open: file open                                                    */
/*          Return value: File number (Pass)                             */
/*                        -1          (Failure)                          */
/*************************************************************************/
open(char *name,                          /* File name                   */
     int mode,                            /* File mode                   */
     int flg)                             /* Unused                      */
{
    /* Check mode depending on 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)                                   */
/*************************************************************************/
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)               */
/*************************************************************************/
read(int  fileno,                          /* File number               */
     char *buf,                            /* Destination buffer address*/
     int  count)                           /* Number of read characters */
{
       int i;

    /* Check mode according to file name and stores each character in buffer */

       if(flmod[fileno]&O_RDONLY||flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   *buf=charget();
                   if(*buf==CR)          /* Line feed character replacement */
                         *buf=LF;
                   buf++;
             }
             return count;
       }
       else
             return -1;
}
```

```
/*****************************************************************************/
/* write: Data write                                                         */
/*        Return value: Number of write characters (Pass)                    */
/*                      -1                          (Failure)                */
/*****************************************************************************/
write(int  fileno,                          /* File number                   */
      char *buf,                            /* Destination buffer address    */
      int  count)                           /* Number of write characters    */
{
       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 potision        */
           int  base)                      /* Origin of offset           */
{
     return -1L;
}


/*************************************************************************/
/*     sbrk: Data write                                                  */
/*           Return value: Start address of the assigned area (Pass)     */
/*                         -1                                 (Failure)  */
/*************************************************************************/
char  *sbrk(int size)                      /* Assigned area size         */
{
       char  *p ;

       if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size           */
            return (char *)-1 ;

       p=brk ;                             /* Area assignment            */
       brk += size ;                       /* End address update         */
       return p ;
}
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                              lowlvl.nor                                       |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       H8S and H8/300-series simulator debugger interface routine              |
;                     -Input/output one character-                              |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    H8S/2600, H8S/2000, H8/300H in normal mode (cpu=2600n, cpu=2000n,           |
;    cpu=300hn)                                                                 |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .CPU         2600N            ; , 2000N, or 300HN
          .EXPORT      _charput
          .EXPORT      _charget

SIM_IO:   .EQU         H'00FE           ; Defines TRAP_ADDRESS

          .SECTION     P,CODE,ALIGN=2



;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: one character output                                               |
;          C program interface: charput(char)                                   |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
           MOV.B       R0L,@IO_BUF      ; Specifies parameter in buffer
           MOV.W       #H'0102,R0       ; Specifies parameter and function code
           MOV.W       #LWORD IO_BUF,R1
           MOV.W       R1,@PARM         ; Specifies I/O buffer address
           MOV.W       #LWORD PARM,R1   ; Specifies parameter block address
           JSR         @SIM_IO
           RTS
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charget: one character input                                              |
;          C program interface:char charget(void)                             |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
           MOV.W       #H'0101,R0       ; Specifies parameter and function code
           MOV.W       #LWORD IO_BUF,R1
           MOV.W       R1,@PARM         ; Specifies I/O buffer address
           MOV.W       #LWORD PARM,R1   ; Specifies parameter block address
           JSR         @SIM_IO
           MOV.B       @IO_BUF,R0L
           RTS

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  I/O buffer definition                                                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION      B,DATA,ALIGN=2

PARM:     .RES.W        1              ; Parameter block area
IO_BUF:   .RES.B        1              ; I/O buffer area

          .END
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                              lowlvl.adv                                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       H8S and H8/300-series simulator debugger interface routine             |
;                    -Input/output one character-                              |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  H8S/2600, H8S/2000, and H8/300H in advanced mode (cpu=2600a, cpu=2000a,     |
;  cpu=300ha)                                                                  |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          .CPU          2600A          ; , 2000A, or 300HA
          .EXPORT       _charput
          .EXPORT       _charget

SIM_IO:   .EQU          H'01FE         ; Defines TRAP_ADDRESS

          .SECTION      P,CODE,ALIGN=2

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: one character output                                              |
;          C program interface: charput(char)                                  |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
           MOV.B        R0L,@IO_BUF    ; Specifies parameter in buffer
           MOV.W        #H'0112,R0     ; Specifies parameter and function code
           MOV.L        #IO_BUF,ER1
           MOV.L        ER1,@PARM      ; Specifies I/O buffer address
           MOV.L        #PARM,ER1      ; Specifies parameter block address
           JSR          @SIM_IO
           RTS
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charget: one character input                                              |
;          C program interface: char charget(void)                            |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
           MOV.W       #H'0111,R0     ; Specifies parameter and function code
           MOV.L       #IO_BUF,ER1
           MOV.L       ER1,@PARM      ; Specifies I/O buffer address
           MOV.L       #PARM,ER1      ; Specifies parameter block address
           JSR         @SIM_IO
           MOV.B       @IO_BUF,R0L
           RTS

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  I/O buffer definition                                                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION       B,DATA,ALIGN=2

PARM:     .RES.L         1              ; Parameter block area
IO_BUF:   .RES.B         1              ; I/O buffer area

          .END
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                              lowlvl.reg                                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       H8S and H8/300-series simulator debugger interface routine             |
;                      -Input/output one character-                            |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                             H8/300 (cpu=300)                                 |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .CPU        300
          .EXPORT     _charput
          .EXPORT     _charget

SIM_IO:   .EQU        H'00FE            ; Defines TRAP_ADDRESS

          .SECTION    P,CODE,ALIGN=2

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: one character output                                              |
;          C program interface: charput(char)                                  |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
           MOV.B       R0L,@IO_BUF      ; Specifies parameter in  buffer
           MOV.W       #H'0102,R0       ; Specifies parameter and function code
           MOV.W       #IO_BUF,R1
           MOV.W       R1,@PARM         ; Specifies I/O buffer address
           MOV.W       #PARM,R1         ; Specifies parameter block address
           JSR         @SIM_IO
           RTS
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charget: one character input                                              |
;          C program interface: char charget(void)                            |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
           MOV.W       #H'0101,R0       ; Specifies parameter and function code
           MOV.W       #IO_BUF,R1
           MOV.W       R1,@PARM         ; Specifies I/O buffer address
           MOV.W       #PARM,R1         ; Specifies parameter block address
           JSR         @SIM_IO
           MOV.B       @IO_BUF,R0L
           RTS

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  I/O buffer definition                                                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION     B,DATA,ALIGN=2

PARM:     .RES.W       1                ; Parameter block area
IO_BUF:   .RES.B       1                ; I/O buffer area

          .END
```

Appendix F Access Range of Short Absolute Addresses

Table F-1 shows the access range of 8-bit absolute addresses and 16-bit absolute addresses in CPU/operating mode.

Table F-1 Access Range of Short Absolute Addresses

CPU/Operating Mode	Access Range of 8-Bit Absolute Addresses (@aa:8)	Access Range of 16-Bit Absolute Addresses (@aa:16)
2600a[:32], 2000a[:32]	0xFFFFFF00 to 0xFFFFFFFF	0x0 to 0x7FFF, 0xFFFF8000 to 0xFFFFFFFF
2600a:28, 2000a:28	0xFFFFF00 to 0xFFFFFFF	0x0 to 0x7FFF, 0xFFF8000 to 0xFFFFFFF
2600a:24, 2000a:24, 300ha[:24]	0xFFFF00 to 0xFFFFFF	0x0 to 0x7FFF, 0xFF8000 to 0xFFFFFF
2600a:20, 2000a:20, 300ha:20	0xFFF00 to 0xFFFFF	0x0 to 0x7FFF, 0xF8000 to 0xFFFFF
2600n, 2000n, 300hn, 300	0xFF00 to 0xFFFF	—

Appendix G Difference from the Old Version

This section shows the difference between the new version (H8S and H8/300-series C compiler Ver. 1.0) and the old version (H8/300-series C compiler Ver. 2.0).

G.1 Additional Functions and Improved Features

G.1.1 Extension of CPU/Operating Mode

The H8S/2600 and H8S/2000-series microcomputers are newly added to the lineup which includes the H8/300H and H8/300 series.

The H8/300-series stack object program creation function, previously supported by the old version, is deleted.

The CPU/operating mode can be selected using the **cpu** option or H38CPU environment variable.

The H38CPU environment variable values can be commonly referred to using the cross assembler and simulator debugger.

When the CPU/operating mode is selected simultaneously with the **cpu** option and H38CPU environment variable, the selection with the **cpu** option has priority.

Note:
The CPU/operating mode selection is omitted, an error occurs for the new version although the H8/300 register object has been created for the old version. Therefore, select the **cpu** option or H38CPU environment variable.

The bit width of an address space can also be selected in H8S/2600, H8S/2000, and H8/300H-series advanced mode.

The bit width of an address space is 20, 24, 28, or 32 in H8S/2600 and H8S/2000-series advanced mode, meaning 1-Mbyte, 16-Mbyte, 256-Mbyte, or 4-Gbyte address space, respectively. When the bit width of an address space is omitted, 32 bits are assumed to be selected.

The bit width of an address space is 20 or 24 in the H8/300H-series advanced mode, meaning 1-Mbyte or 16-Mbyte address space, respectively. When the bit width of an address space is ignored, 24 bits are assumed to be selected.

G.1.2 Optimization Function Improvement

The optimization function is improved to reduce the object code size, including constant allocation to registers, external variable optimization, and decrease in the same character string area.

G.1.3 Addition of the "speed" Option

The new option is added to improve the following speed functions.

- Register save/recovery processing of function entry/exit
- Expansion code of shift operation, a loop statement, and a **switch** statement
- In-line expansion of function calls
- Assignment code of a structure and **double**

G.1.4 CPU-Unique Function Support

The following functions are supported in the H8S and H8/300-series CPU architecture.

- Use of a short absolute addressing mode (**abs8**, **abs16** option, **#pragma abs8**, **#pragma abs16**):
  Data that is allocated to the static area is accessed using the short absolute addressing mode (**@aa:8**, **@aa:16**).

- Expansion interpretation of operation size (**cpuexpand** option):
  Effective CPU instruction codes are created for data multiplication and division.

- Indirect memory function call (**indirect** option, **#pragma indirect**):
  A function is called in indirect memory addressing mode (**@aa:8**).

- Using the block transfer instruction (**eepmov** option):
  The structure assignment expression is expanded by the **EEPMOV** instruction.

G.1.5 Extension Function Support

ANSI extension functions (**#pragma**) are added and modified.

- Function in-line expansion (**#pragma inline**):
  A called function is expanded in the line of function call.

- Interrupt function creation (**#pragma interrupt**):
  Stack switching, trap-instruction return, and interrupt function end specifications are added.

- Section switching (**#pragma section, #pragma abs8 section, #pragma abs16 section, #pragma indirect section**):
  Sections can be switched in C program.

- Control of a register save/recovery code (**#pragma regsave, #pragma noregsave**):
  All register save/recovery codes other than ER0 and ER1 (R0 and R1 for H8/300) can be unconditionally output or inhibited at the function entry/exit.

G.1.6 Intrinsic Function Support

The following functions, which cannot be described in C language, are supported as intrinsic functions.

- Setting and reference of the condition code register (CCR)
- Setting and reference of the extend register (EXR)
- Special instructions (**TRAPA, SLEEP, MOVFPE, MOVTPE, EEPMOV, MAC**)
- Rotation operation
- Condition code reflection operation
- Decimal operation

G.1.7 Debugging Function Improvement

Debugging information is output for C-source level debugging during optimization option specification. In addition, when an assembly source program is specified for an object type, the **.LINE** control instruction is output, enabling C-source level debugging.

G.1.8 Addition of Other Options

- One-byte **enum** support (**byteenum** option)
- Comment nest (**comment** option)
- **switch** statement code (**case=ifthen**, **table** options)
- Limit value extension (**limits** option)
- Information message output specification (**message** option)
- Register extension of register variable assignment (**regexpansion** option)
- Option specification using file (**subcommand** option)
- External variable optimization (**volatile** option)

G.2 Language Specification Expansion

G.2.1 Changing Character String Specifications of the "error" Statement

Double quotation marks are not required for character strings specified by the **#error** statement.

Example:

Old version

```
#error "character-string"
```

New version

```
#error character-string
```

G.2.2 Identifier Class Specifications of "typedef" and Structure Member Name

An identifier having the same structure-member name as the identifier of **typedef** can be used.

Example:

Old version

```
typedef int INT;
main()
{
    struct S{int INT;}; → Error
}
```

New version

```
typedef int INT;
main()
{
    struct S{int INT;}; → Normal
}
```

G.2.3 errno.h Support

Declaration **errno** indicating a library error state can be declared in include file **errno.h**.

G.2.4 Changing Macro Expansion Timing

The expansion timing of a function-type macro is changed.

Example:

Old version

```
#define A(a) a++
int b;
main()
{
    int A; → Error
    A(b);
}
```

New version

```
#define A(a) a++
int b;
main()
{
    int A; → Normal
    A(b);
}
```

G.2.5 Changing Initialization Specifications of the "void" Pointer

Types other than **void** can be used as the initialization specifier of the **void** pointer.

Example:

Old version

```
int a[10];
void *p=a; → Error
```

New version

```
int a[10];
void *p=a; → Normal
```

G.2.6 Changing Initialization Specifications of the "char" Array

Character strings can be specified by enclosing them in "{" and "}" as the initialization specifier of the **char** array.

Example:

Old version

```
main()
{
    char a[]={"abc"}; → Error
}
```

New version

```
main()
{
    char a[]={"abc"}; → Normal
}
```

G.2.7 "memmove" Function Support

The **memmove** function is supported as a standard library function.

"memmove" Function

- Function

  The specified storage area data is copied to the destination storage area. Even if the source storage area partially overlaps with the destination storage area, the overlapping parts of the source storage area are copied before being overwritten, preventing incorrect copying.

- Calling procedure

```
#include <string.h>
void *ret, *s1;
const void *s2;
size_t      n;
      ret=memmove(s1,s2,n)
```

- Parameter

Name	Type	Function
s1	Pointer indicating **void**	Points to the storage area of a copied destination
s2	Pointer indicating **const void**	Points to the source storage area
n	**size_t**	Number of copied characters

- Return values

  Normal type: Pointer to **void** is the **s1** value
  Abnormal type: —

G.3 Object Program Compatibility

The C compiler creates an object program that has a different format from the old version. When the object program is linked to an object program created by the old-version C compiler, the object program format must be modified using the SYSROF file converter attached to the C compiler.

When the old-version object program calls a routine (function having symbol name "xxx2") during execution, the standard library format must also be modified and linked.

Object program compatibility is shown below.

G.3.1 New Version Format Link

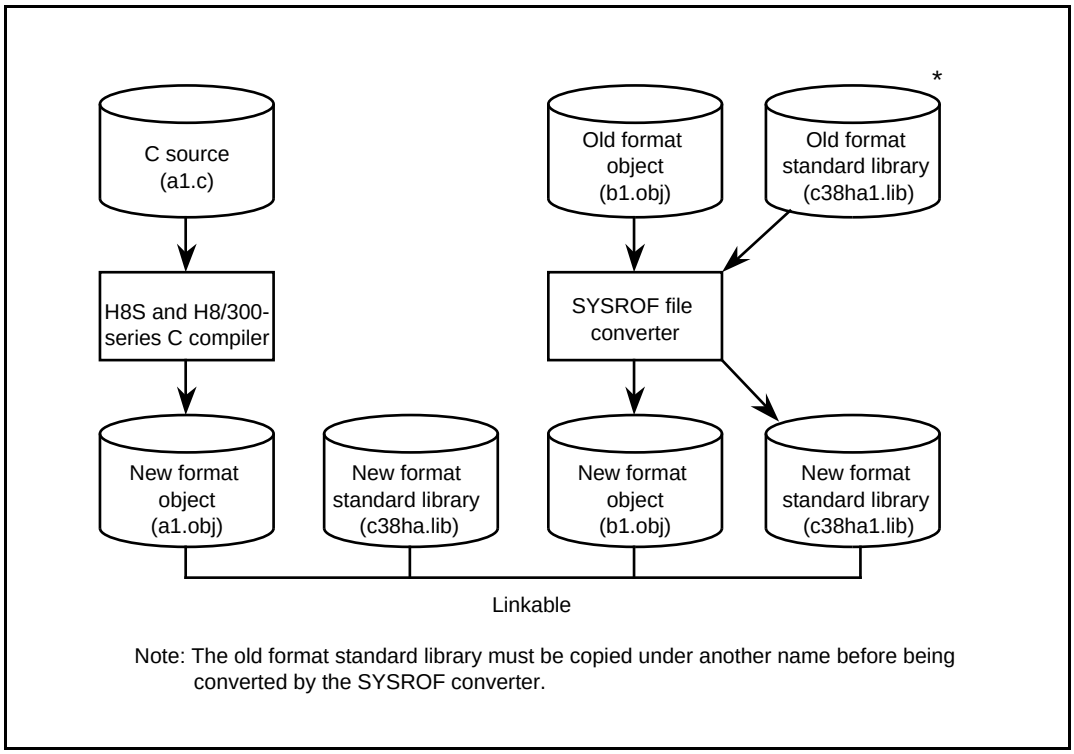


Figure G-1 New-Version Link

Usage Example:

```
① $ch38 -cpu=300ha a1.c
② $fcnv b1.obj -v2
③ $fcnv c38ha1.lib -v2
④ $lnk a1.obj,b1.obj -lib=c38ha,c38ha1
```

<Explanation>

① Compiles the C source and creates a new-format object program.

② Converts the old object program format to the new format.

③ Converts the old standard library format to the new format.

④ Links the new-format object program to the standard library.

G.3.2 Old Version Format Link

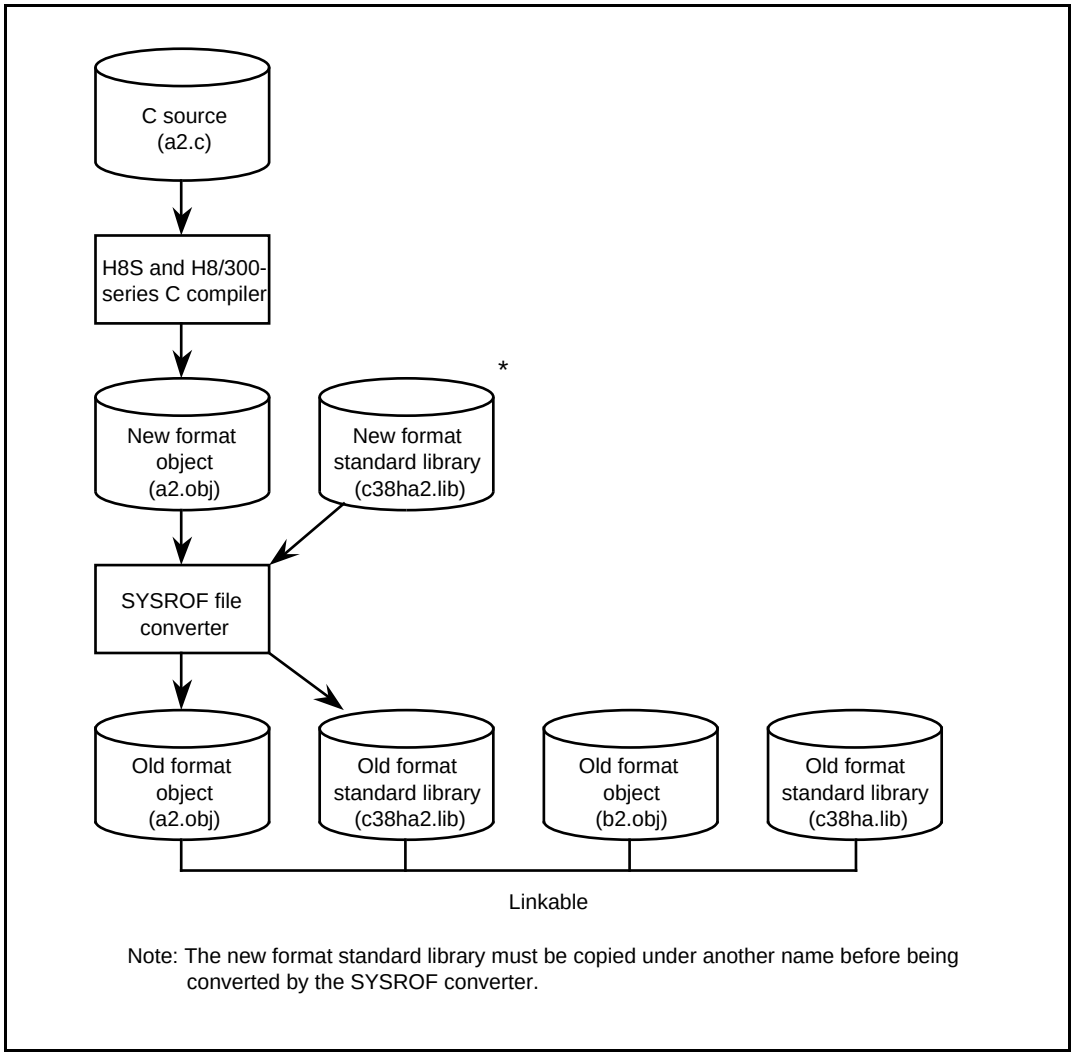


Figure G-2 Old-Version Link

Usage Example:

```
① $ch38 -cpu=300ha a2.c
② $fcnv a2.obj
③ $fcnv c38ha2.lib
④ $lnk a2.obj,b2.obj -lib=c38ha2,c38ha
```

<Explanation>

① Compiles the C source and creates a new-format object program.

② Converts the new object program format to the old format.

③ Converts the new standard library format to the old format.

④ Links the old-format object program to the standard library.

See the "H8S and H8/300-Series SYSROF File Converter Instruction Manual," which is attached to the converter product, for more information on the SYSROF file converter.

Appendix H ASCII Codes

Table H-1 ASCII Codes

Upper 4 Bits / Lower 4 Bits	0	1	2	3	4	5	6	7	
0	NUL	DLE	SP	0	@	P	`	p	
1	SOH	DC1	!	1	A	Q	a	q	
2	STX	DC2	"	2	B	R	b	r	
3	ETX	DC3	#	3	C	S	c	s	
4	EOT	DC4	$	4	D	T	d	t	
5	ENQ	NAK	%	5	E	U	e	u	
6	ACK	SYN	&	6	F	V	f	v	
7	BEL	ETB	'	7	G	W	g	w	
8	BS	CAN	(	8	H	X	h	x	
9	HT	EM	)	9	I	Y	i	y	
A	LF	SUB	*	:	J	Z	j	z	
B	VT	ESC	+	;	K	[	k	{	
C	FF	FS	,	<	L	\	l	|	
D	CR	GS	-	=	M	]	m	}	
E	SO	RS	.	>	N	^	n	~	
F	SI	US	/	?	O		o	DEL	