Hitachi Microcomputer Support Software

SH Series C Compiler

USER'S MANUAL

HITACHI

ADE-702-095
HS0700CLCU1SE

Preface

This manual explains the facilities and operating procedures for the SH series C compiler (Ver. 2.0). The C compiler translates source programs written in C into relocatable object programs or assembly programs for Hitachi SH7000 series RISC microcomputers.

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

(1) PART I OVERVIEW AND OPERATIONS
The overview sections cover the following:
V C compiler functions
W Developing procedures
The operation sections cover the following:
X How to invoke the C compiler
Y Optional functions
Z 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.

This manual corresponds to operating systems that function on UNIX, MS-DOS, or IBM-PC systems. In this manual, operating systems functioning on MS-DOS or IBM-PC systems are referred to as PC systems.

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

Symbols Used in This Manual

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

UNIX is an operating system administrated by the UNIX System Laboratories (United States).
MS-DOS is an operating system administrated by the Microsoft Corporation (United States).
IBM-PC is an personal computer system administrated by IBM (United States).

Contents

Figures

Part I

Part II

Part III

Part IV

Appendix

Tables

PART I

OVERVIEW AND OPERATIONS

Section 1 Overview

The SH series C compiler inputs source programs written in C and outputs relocatable object programs or assembly source programs.

The C compiler supports the SH7000-series microcomputers (referred to as SH).

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.

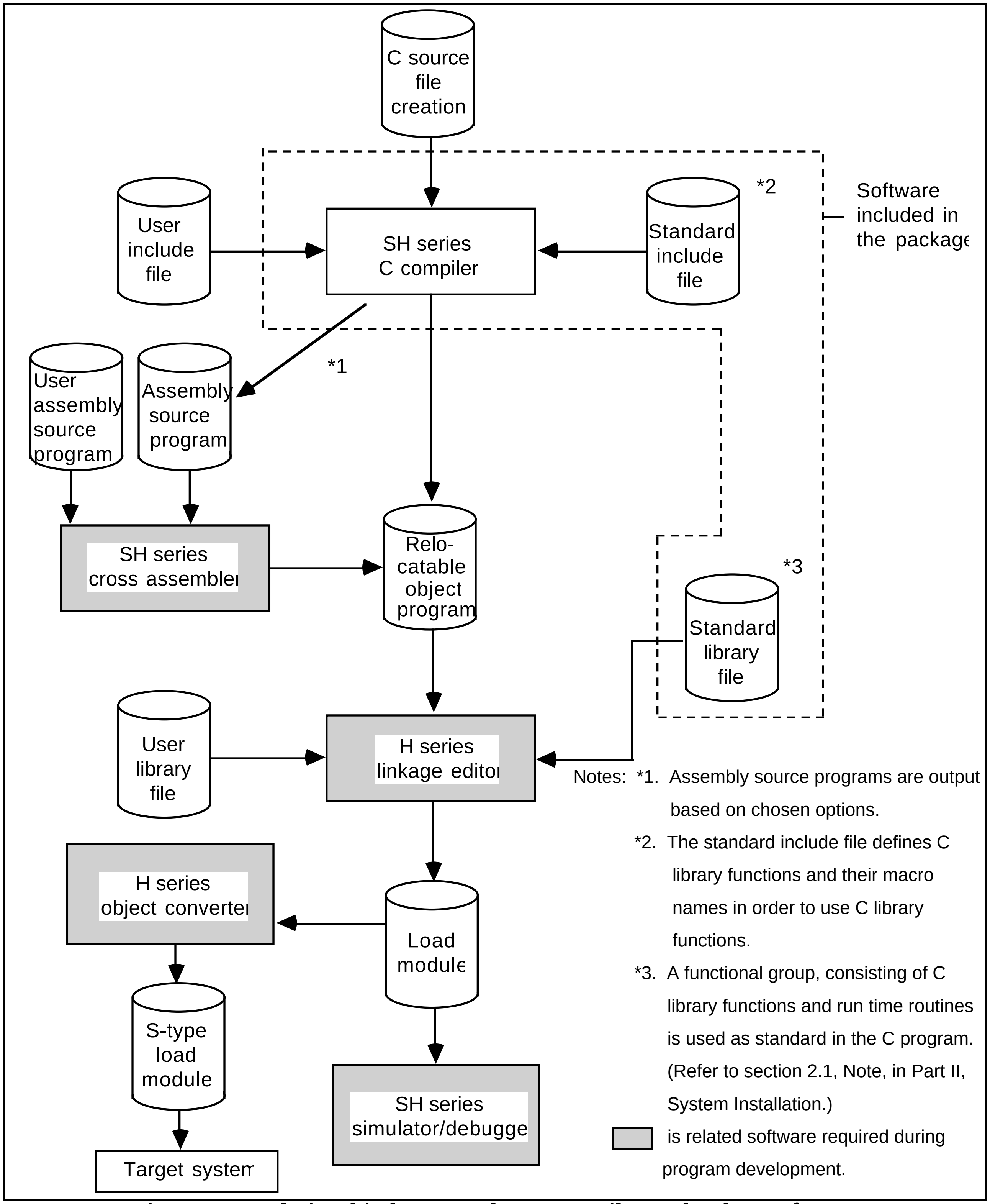


Figure 2-1 Relationship between the C Compiler and Other Software

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.

UNIX systems:

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

PC systems:

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

The general operations of the C compiler are described below.

Compiling Programs:

```
shcΔtest.c (RET)
```

The C source program test.c is compiled.

C Compiler Options (UNIX):

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

Insert minus (-) before options (**debug**, **listfile**, and **show**). When multiple options are specified, separate them with a space (Δ). Also when multiple suboptions are specified, separate them with a comma (,).

C Compiler Options (PC):

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

Insert a slash (/) before the options (**debug**, **listfile**, and **show)**. When multiple options are specified, separate them with a space (Δ). Also when multiple suboptions are specified, separate them with a comma (,) and enclose them in parentheses.

Compiling Multiple Programs:

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

Example 1: Specifying multiple programs

```
shcΔtest1.cΔtest2.c(RET)
```

Example 2: Specifying options for all C source programs

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

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

Example 3: Specifying options for particular C source programs

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

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

Option List:

```
shc (RET)
```

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

3.2 Naming Files

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 3-1.

Table 3-1 Standard File Extensions Used by the C Compiler

File Extensioner	Description
c	Source program file written in C
h	Include file
lst, lis	Listing file*1
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: *1. The listing file extension is lis on UNIX systems and 1st on PC systems.

The general conventions for naming files depend on the host machine. Refer to the manual of the host machine in use.

3.3 Compiler Options

Table 3-2 shows C compiler option formats, abbreviations, and defaults. Characters underlined indicate the minimum valid abbreviation. Bold characters indicate default assumption.

Table 3-2 C Compiler Options

<table>
<tr><th>Item</th><th>Format</th><th>Suboption</th><th>Specification</th></tr>
<tr><td>Optimization level</td><td>optimize =</td><td>0 |</td><td>Object without optimization is output.</td></tr>
<tr><td></td><td></td><td>1</td><td>Object with optimization is output.</td></tr>
<tr><td>Listings and formats[*1]</td><td>show =</td><td>source |nosource |</td><td>Source list yes/no</td></tr>
<tr><td></td><td></td><td>object |noobject |</td><td>Object list yes/no</td></tr>
<tr><td></td><td></td><td>statistics |nostatistics |</td><td>Statistics information yes/no</td></tr>
<tr><td></td><td></td><td>include |noinclude |</td><td>List after include expansion yes/no</td></tr>
<tr><td></td><td></td><td>expansion |noexpansion |</td><td>List after macro expansion yes/no</td></tr>
<tr><td></td><td>[*2]</td><td>width = <numeric value> |</td><td>Maximum characters per line: 0, 80–132</td></tr>
<tr><td></td><td>[*2]</td><td>length = <numeric value></td><td>Maximum lines per page: 0, 40–255</td></tr>
<tr><td></td><td></td><td>Default: w = 132, l = 66</td><td></td></tr>
<tr><td>Listing file</td><td colspan="2">listfile [= <listing file name>] [*3]</td><td>Output</td></tr>
<tr><td></td><td colspan="2">nolistfile</td><td>No output</td></tr>
<tr><td>Object file</td><td colspan="2">objectfile = <object file name></td><td>Output</td></tr>
<tr><td>Object progam format</td><td>code =</td><td>machine code |</td><td>Program in machine language is output.</td></tr>
<tr><td></td><td></td><td>asmcode</td><td>Assembly source progam is output.</td></tr>
<tr><td>Debug information</td><td>debug</td><td></td><td>Output</td></tr>
<tr><td></td><td>nodebug</td><td></td><td>No output</td></tr>
<tr><td>Macro name</td><td>define =</td><td><macro name> = <name> |</td><td><name> is defined as <macro name></td></tr>
<tr><td></td><td></td><td><macro name> = <constant> |</td><td><constant> is defined as <macro name></td></tr>
<tr><td></td><td></td><td><macro name> [*4]</td><td><macro name> is assumed to be defined.</td></tr>
<tr><td>Include file</td><td>include =</td><td><path name> [*5]</td><td>Include file destination path name is specified. (Multi-specification is possible.) [*4]</td></tr>
<tr><td>Section name</td><td>section = [*5]</td><td>program = <section name> |</td><td>Program area section name is specified.</td></tr>
<tr><td></td><td></td><td>const = <section name> |</td><td>Constant area section name is specified.</td></tr>
<tr><td></td><td></td><td>data = <section name |</td><td>Initialized data area section name is specified.</td></tr>
<tr><td></td><td></td><td>bss =<section name></td><td>Non-initialized data area section name is specified.</td></tr>
<tr><td></td><td></td><td>Default: p = P, c = C, d = D,b = B</td><td></td></tr>
<tr><td>Help message</td><td>help[*6]</td><td></td><td>Output</td></tr>
</table>

Notes: *1. **show** option is invalid when **listfile** is specified.

*2. The assignments of **show = width = 0** or **show = length = 0** are interpreted as below.
show = width = 0: No line feed is performed until line feed code is output.
show = length = 0: Maximum line number is not specified, and page feed is not performed.

*3. If file name is not specified, standard file extension is added to the source file name.

*4. Macro names specified by options are shown in table 3-3.

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

Item	Explanation
Macro name	A character string beginning with an alphabetic letter or an underscore followed by zero or more alphabetic letters, underscores, and numbers (0 to 9).
Name	A character string beginning with a letter or an underscore followed by zero or more alphabetic letters, underscores, and numbers.
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.

*5. Refer to descriptions in Preprocessor Specifications,in Appendix A.1 for details on how to retrieve the include file.

*6. When the **help** option is specified, all other options are invalid.

3.4 Option Combinations

If a pair of conflicting options or suboptions are specified for a file, only one of them is considered valid. Table 3-4 shows such option combinations.

Table 3-4 Option Combinations

Option Combinations

Valid Option	Invalid Options
nolistfile	**show**
code = asmcode	**debug**, and **show = object**
help	All other options

3.5 C Compiler Listings

This section describes C compiler listings and their formats.

Structure of C Compiler Listings: Table 3-5 shows the structure and contents of C compiler listings.

Table 3-5 Structure and Contents of C Compiler Listings

List Structure	Contents	Option Specification Method *1	Default
Source listing	Listing consists of source programs	**show = source** **show = nosource**	Output
	Source program listing after include file and macro expansion	**(show = include)** *2 **(show = expansion)** **(show = noinclude)** **(show = noexpansion)**	No output
Object listing	Machine language generated by the C compiler	**show = object** **show = noobject**	Output
Statistics	Total number of errors, the number of source program lines, length of each section (byte), and the number of symbols	**show = statistics** **show = nostatistics**	Output
command line specification	File names and options specified on the **command** line	—	Output

Notes: *1. All options are valid when **listfile** is specified.

*2. The option enclosed in parentheses is only valid when **show = source** is specified.

Source Listing: The source listing can be output in two ways. When **show = noinclude** and **show = noexpansion** is specified, the unpreprocessed source program is output. When **show = include**

or **show =expansion** is specified, the preprocessed source program is output. Figures 3-1 and 3-2 show these output formats, respectively. Bold characters in figure 3-2 show the differences.

Figure 3-1 Source Listing Output for show = noinclude and noexpansion

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

FILE NAME: m0260.c

  Seq    File      Line     0----+----1----+----2----+----3----+----4----+----5--
    1 m0260.c         1      #include "header.h"
    4 m0260.c         2
    5 m0260.c         3      int sum2(void)
    6 m0260.c         4      {   int j;
    7 m0260.c         5
    8 m0260.c         6      #ifdef SMALL
    9 m0260.c         7          j=SML_INT;
   10 m0260.c         8      #else
   11 m0260.c         9          j=LRG_INT;
   12 m0260.c        10      #endif
   13 m0260.c        11
   14 m0260.c        12          return j; /* continue12345678901234567890123456
   V   W            X     +234567890123456789 0  */
```

Figure 3-2 Source Listing Output for show = include and expansion

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

FILE NAME: m0260.c

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

Object Listing: Figure 3-3 shows an example of an object listing.

Description

V Listing line number

W Source program file name or include file name

X Line number in source program or include file

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

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

[Lines containing a macro expansion due to **#define** directives are marked with an E when **show=expansion** is specified.

\ 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.

Figure 3-3 Object Listing

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

FILE NAME: m0251.c

  Seq File          Line     0----+----1----+----2----+----3----+----4----+----5
    1 m0251.c          1      extern int sum(int);
    2 m0251.c          2
    3 m0251.c          3      int
    4 m0251.c          4      sum(int x)
    5 m0251.c          5      {
    6 m0251.c          6              int i;
    7 m0251.c          7              int j;
    8 m0251.c          8
    9 m0251.c          9              j=0;
   10 m0251.c         10              for(i=0; i<=x; i++) {
   11 m0251.c         11                      j+=i;
   12 m0251.c         12              }
   13 m0251.c         13              return j;
   14 m0251.c         14      }

************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT    OFFSET   CODE          C LABEL     INSTRUCTION OPERAND      COMMENT
V        W         X                                           Y
Z
P                               ; File m0251.c    ,  Line 4     ; block
     00000000                   _sum:                  [       ; function: sum
                                                                ; frame size=8 \
     00000000   7FF8                    ADD       #-8,R15
                                ; File m0251.c    , Line 5      ; block
                                ; File m0251.c    , Line 9      ; expression statement
     00000002   E300                    MOV       #0,R3
     00000004   2F32                    MOV.L     R3,@R15
                                ; File m0251.c    , Line 10     ; for
     00000006   E300                    MOV       #0,R3
     00000008   1F31                    MOV.L     R3,@(4,R15)
     0000000A   A009                    BRA       L104
     0000000C   0009                    NOP
```

Description

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

W The offset indicates the offset address relative to the beginning of each section.

X Contents of the offset address of each section

Y Assembly code corresponding to machine language

Z Comments indicating the C program structure (only output when not optimized; however, labels are always output)

[Line information corresponding to the C program (only output when not optimized)

\ Stack frame size in bytes (always output)

Statistics Information: Figure 3-4 shows an example of statistics information.

Figure 3-4 Statistics Information

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

********** ERROR INFORMATION ***********
                                                        } V
NUMBER OF ERRORS:             0
NUMBER OF WARNINGS:           0

******* SOURCE LINE INFORMATION ********
                                                        } W
COMPILED SOURCE LINE:         13

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

PROGRAM  SECTION(P): 0x00004A Byte(s)                   } X
CONSTANT SECTION(C): 0x000000 Byte(s)
DATA     SECTION(D): 0x000000 Byte(s)
BSS      SECTION(B): 0x000000 Byte(s)

 TOTAL PROGRAM SIZE: 0x00004A Byte(s)


********** LABEL INFORMATION ***********
                                                        } Y
NUMBER OF EXTERNAL REFERENCE  SYMBOLS:        0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:        1
```

Description

V Total number of messages by the level

W Number of compiled lines from the source file

X Size of each section and total size of sections

Y Number of external reference symbols, number of external definition symbols, and total number of internal and external labels

Note: Section size information (X) and label information (Y) are not output if an error-level error or a fatal-level error has occurred when option **noobject** is specified. In addition, section size information (X) is not output when option **code = asmcode** is specified.

command Line Specification: The file names and options specified on the **command** line when the compiler is invoked are displayed. Figure 3-5 shows an example of **command** line specification information.

Figure 3-5 command Line Specification

```
*** COMMAND PARAMETER ***

-listfile test.c
```

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. To edit and compile efficiently, it is recommended to split the source program into smaller programs (approximately 2 ksteps) and compile them separately.

Table 1-1 Limitation of the C Compiler

		Limit	
Classification	**Item**	**UNIX**	**PC**
Invoking the C compiler	Number of source programs that can be compiled at one time	16	1
	Total number of macro names that can be specified using the **define** option	16	16
	Length of file name (characters)	128	128
Source programs	Length of one line (characters)	4096	512
	Number of source program lines	32767	16383
Preprocessing	Nesting level of files in an **#include** directive	8	5
	Total number of macro names that can be specified in a **#define** directive *1	4096	1024
	Number of arguments that can be specified using a macro definition or a macro call operation	63	31
	Depth of the recursive expansion of a macro name	32	16
	Nesting level of **#if**, **#ifdef**, **#ifndef**, **#else**, or **#elif** directives	32	6
	Total number of operators and operands that can be specified in an **#if** or **#elif** directive	512	210
Declarations	Number of function definitions	512	256
	Number of external identifiers used for external linkage*2	4096	511
	Number of internal identifiers that can be used in one function	4096	512
	Number of internal labels*3	16384	2048
	Number of symbol table entries*4	8192	1024
	Total number of pointers, arrays, and functions that qualify the basic type	16	16
	Array dimensions	6	6

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

Classification	Item	Limit UNIX	Limit PC
Statements	Nesting levels of compound statements	32	15
	Levels of statement nesting in a combination of repeat (**while**, **do**, and **for**) and select (**if** and **switch**) statements	32	15
	Number of **goto** labels that can be specified in one function	511	256
	Number of **switch** statements	256	128
	Nesting levels of **switch** statements	16	15
	Number of **case** labels	511	255
	Nesting levels of **for** statements	16	15
Expressions	Number of arguments that can be specified using a function definition or a function call operation	63	31
	Total number of operators and operands that can be specified in one expression	About 500	About 200
C library functions	Number of files that can be opened simultaneously by the **open** function	20	20

Notes: *1. As the C compiler itself defines five macro names (__ LINE __, __ FILE __, __ DATE __, __ TIME __, and __ STDC __), the user can define a maximum of 4091 macro names in UNIX systems and a maximum of 1019 macro names in PC systems.

*2. As the C compiler itself defines two symbols, the user can define a maximum of 4094 external identifiers in UNIX systems and a maximum of 509 external identifiers in PC systems.

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

*4. The number of symbol table entries is determined by adding the following numbers:
Number of external identifiers
Number of internal identifiers for each function
Number of string literals
Number of initial values for structures and arrays in compound statements
Number of compound statements
Number of **case** labels
Number of **goto** labels

Section 2 Executing a C Program

This section covers object programs which are generated by the C compiler. In particular, this section explains what items are required to link C programs with assembly programs and how to install programs on the SH 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 shared among 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 arguments 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 SH 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.

V 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**.

W Write Operation
Indicates whether write operations are enabled at program execution.

X Initial Value
Shows whether there is an initial value when program execution starts.

Y Alignment
Restricts addresses to which data is allocated.

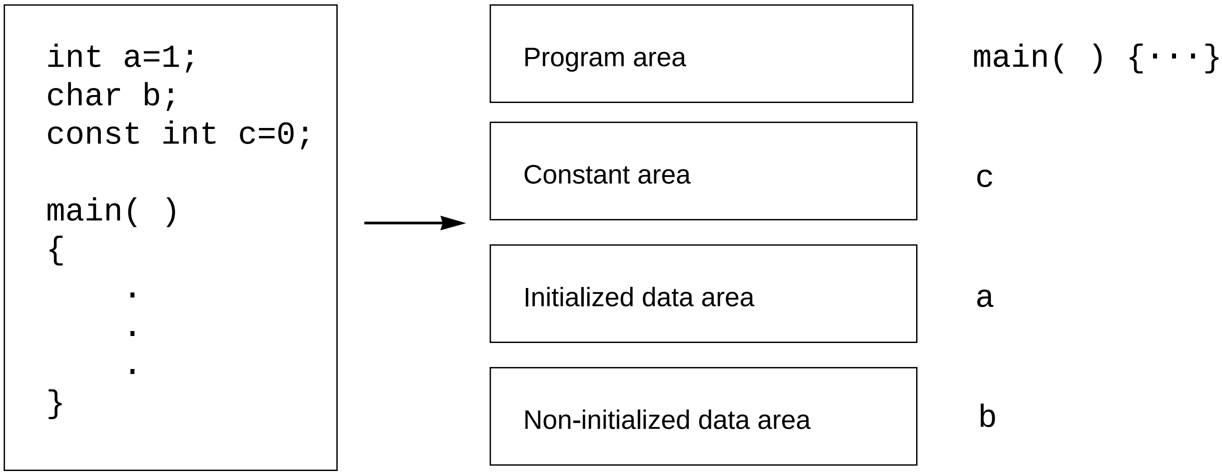
Table 2-1 shows the types and characteristics of those memory areas.

Table 2-1 Memory Area Types and Characteristics

Memory Area Name	Section Name *	Section Type	Write Operation	Initial Value	Alignment	Contents
Program area	P	code	Disabled	Yes	4 bytes	This area stores machine codes.
Constant area	C	data	Disabled	Yes	4 bytes	This area stores **const** data.
Initialized data area	D	data	Enabled	Yes	4 bytes	This area stores data whose initial values are specified.
Non-initialized data area	B	data	Enabled	No	4 bytes	This area stores data whose initial values are not specified.
Stack area	—	—	Enabled	No	4 bytes	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, Allocation to Dynamic Area, 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**.

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



C program Section generated by the 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:

V Size
  Shows the amount of memory needed to store the data.

W Alignment
  Restricts the addresses to which data is allocated. There are three types of alignment, 1-byte alignment in which data can be allocated to any address, 2-byte alignment in which data is allocated to an even byte address, and 4-byte alignment in which data is allocated to an address indivisible by four.

X Data range
  Shows the range of scalar-type data.

Y 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	Data Range Maximum Value
char	1	1	Used	-2^{7} (–128)	$2^{7}-1$ (127)
signed char	1	1	Used	-2^{7} (–128)	$2^{7}-1$ (127)
unsigned char	1	1	Unused	0	$2^{8}-1$ (255)
short	2	2	Used	-2^{15} (–32768)	$2^{15}-1$ (32767)
unsigned short	2	2	Unused	0	$2^{16}-1$ (65535)
int	4	4	Used	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
unsigned int	4	4	Unused	0	$2^{32}-1$ (4294967295)
long	4	4	Used	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
unsigned long	4	4	Unused	0	$2^{32}-1$ (4294967295)
enum	4	4	Used	-2^{31} (–2147483648)	$2^{31}-1$ (2147483647)
float	4	4	Used	$-\infty$	$+\infty$
double **long double**	8	4	Used	$-\infty$	$+\infty$
Pointer	4	4	Unused	0	$2^{32}-1$ (4294967295)

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

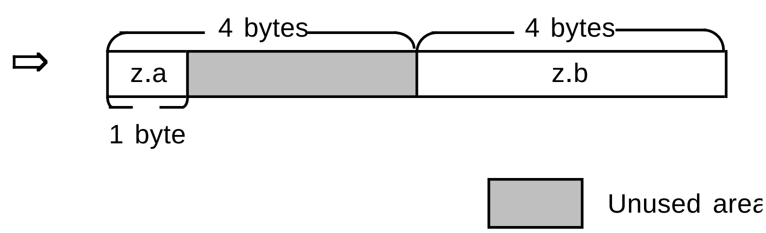
Data Type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array type	Array element alignment	(Number of array elements) x (Element size)	`int a[10];` Alignment: 4 bytes Size: 40 bytes

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

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

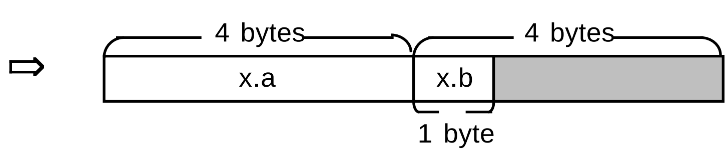
Notes: *1. When structure members are allocated, unused area may be generated between structure members to align data types.

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



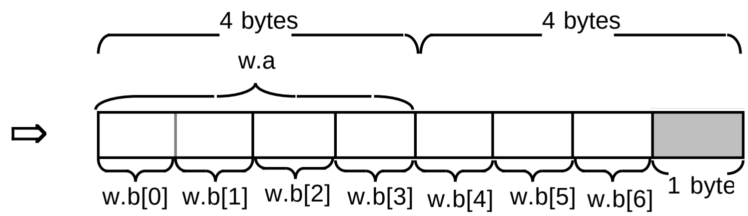
If a structure has 4-byte alignment and the last member ends at an address indivisible by four, the remaining bytes are included in this structure.

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



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

```
union {
    int a;
    char b[7];}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**, **unsigned int**, **long**, and **unsigned long**
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 *2

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 all higher-order bits generated during data 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**.

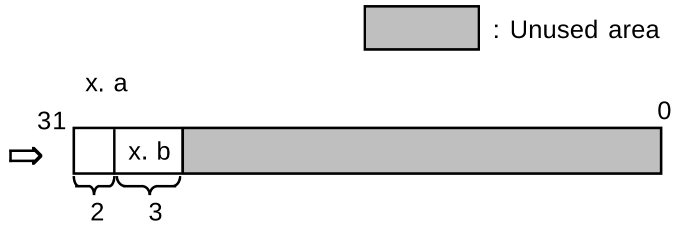
- Bit field allocation

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

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

Example:

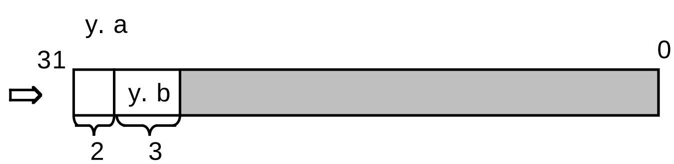
```
struct b1{
        int a:2;
        int b:3;
    }x;
```



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

Example:

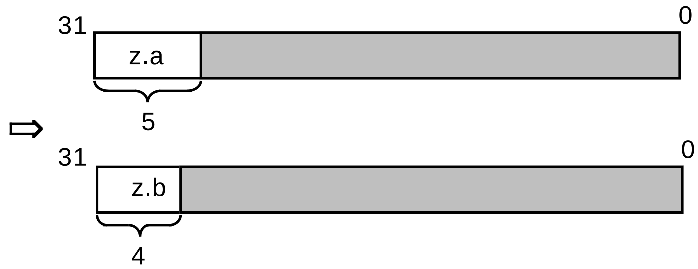
```
struct b1{
        long         a:2;
        unsigned int b:3;
    }y;
```



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

Example:

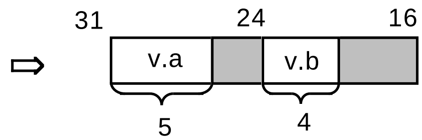
```
struct b1{
        int  a:5;
        char b:4;
    }z;
```



Y 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:

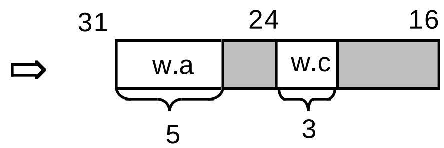
```
struct b2{
      char a:5;
      char b:4;
  }v;
```



Z 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 to describe almost all processes in microcomputer application systems. In particular, the SH-series C compiler supports operations, such as access to the SH microcomputer registers as intrinsic functions. Refer to section 3.2, Intrinsic Functions, in Part II, Programming, for details on intrinsic functions.

Processes which cannot be written in C, for example, calculations like multiplication and addition performed by the MAC instruction, must be written in assembly language, and then linked with the C program.

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

- External identifier reference
- Function call interface

2.3.1 External Identifier Reference

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

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

When variable 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=4
_a:  .DATA.L  1
_b:  .DATA.L  1
     .END
```

C program (reference)

```
extern int a,b;


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

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

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

C program (definition)

```
int a;
```

Assembly program (reference)

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

2.3.2 Function Call Interface

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

(1) Stack Pointer
(2) Allocating and Deallocating Stack Frames
(3) Registers
(4) Setting and Referencing Parameters and Return Values

Stack Pointer: Valid data must not be stored in a stack area with an address lower than the stack pointer, since the data may be destroyed by an interrupt process.

Allocating and Deallocating Stack Frames: In a function call (right after the JSR or the BSR instruction has been executed), the stack pointer indicates the lowest address of the stack used by the calling function. Allocating and setting data at addresses greater than this one is a role of the calling function. After the called function deallocates the area it has set with data, control returns to the calling function usually with the RTS instruction. The calling side then deallocates the area having an address higher than the return value address and the parameter area.

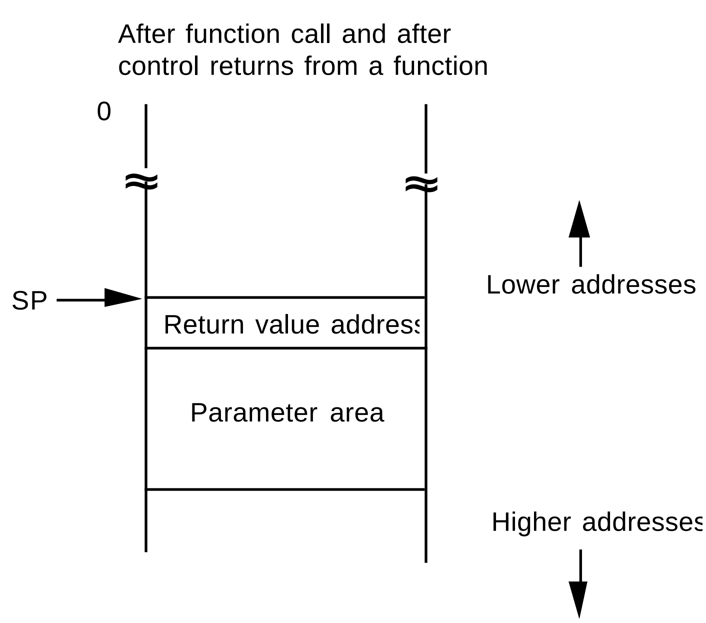


Figure 2-1 Allocation and Deallocation of a Stack Frame

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

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

Item	Registers Used in a Function	Notes on Programming
Guaranteed registers	R0 – R7	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.
Non-guaranteed	R8 – R15, MACH, MACL, and PR	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.

(a) A subroutine in an assembly program is called by a C program

Assembly program (called program)

```
        .EXPORT   _sub
        .SECTION  P,CODE,ALIGN=2
_sub:   MOV.L     R14,@-R15
        MOV.L     R13,@-R15
        ADD       #-8,R15
              .
              .
              .
        ADD       #8,R15
        MOV.L     @R15+,R13
        RTS
        MOV.L     @R15+,R14
```

- MOV.L R14,@-R15 / MOV.L R13,@-R15: Data in those registers needed by the called function is pushed onto the stack.
- ADD #-8,R15 … ADD #8,R15: Function processing (Since data in registers R0 to R7 is pushed onto a stack by the calling C program, the assembly program can use them freely without having to save them first.)
- MOV.L @R15+,R13 … MOV.L @R15+,R14: Register data is popped from the stack.

C program (calling program)

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

(b) 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
              .
              .
              .
        STS.L    PR,@-R15

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

The called function is declared by the .IMPORT control instruction with an underscore character (_) at the beginning.

Store the PR register (return address storage register) when calling the function.
If registers R0 and R7 contain valid data, the data is pushed onto the stack or stored in unused registers.

The sub function is called.

The PR register is restored.

Address data of the sub function

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.

(a) General rules concerning parameters and return values

(i) Passing parameters

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

(ii) 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.

— Type Conversion of Parameters Whose Types are Declared

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

— Type conversion of parameters for which types are not declared

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

- Parameters whose types are **char**, **unsigned char**, **short**, or **unsigned short** are converted to **int**.
- Parameters whose types are **float** are converted to **double**.
- Other parameters are not converted.

— Return value type conversion

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

Example:

```
v  long f( );
    long f( )
    {   float x;
        return x;
    }
w  void p ( int,... );
    f( )
    {   char c;
        P ( 1.0, c );
    }
```

The return value is converted to long.

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

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.

(b) Parameter area allocation

Parameters are allocated to registers, or when this is impossible, to a stack parameter area. Figure 2-2 shows the parameter area allocation. Table 2-6 lists the general parameter area allocation rules.

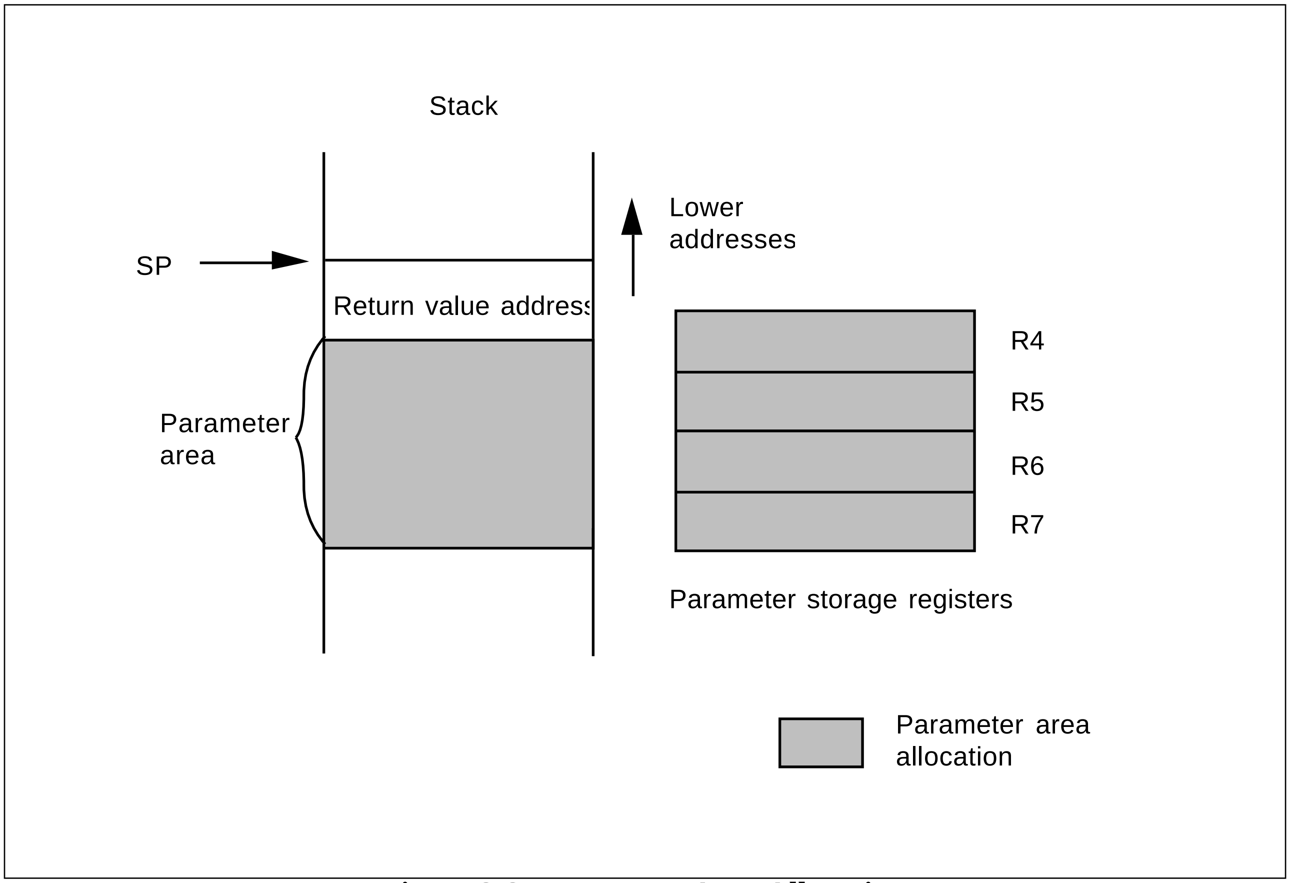


Figure 2-2 Parameter Area Allocation

Table 2-6 General Rules on Parameter Area Allocation

Allocation Rules

Parameters Allocated to Registers		Parameters Allocated to a Stack
Parameter Storage Registers	**Target Type**	
R4 – R7	**char**, **unsigned char**, **short**, **unsigned short**,**int**, **unsigned int**, **long**, **unsigned long**, **float**, and pointer	V Parameters whose types are other than target types for register passing W Parameters of a function which has been declared by a prototype declaration to have variable-number parameters* X Other parameters are already allocated to R4 – R7.

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,int, int,...);
      :
      f2(a,b,c,x,y,z);
      :
```

→ x, y, and z are allocated to a stack.

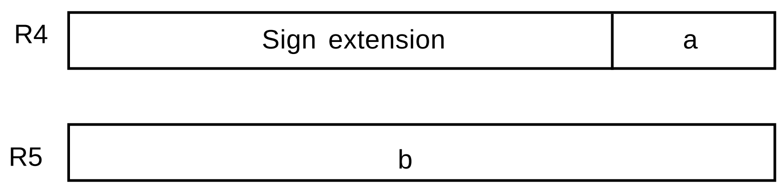
(c) Parameter allocation

(i) Allocation to parameter storage registers

Following the order of their declaration in the source program, parameters are allocated to the parameter storage registers starting with the smallest numbered register. Figure 2-3 shows an example of parameter allocation to registers.

Figure 2-3 Example of Allocation to Parameter Registers

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



(ii) 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 4-byte alignment. Also, the area size for each parameter must be a multiple of four bytes. This is because the SH stack pointer is incremented or decremented in 4-byte units.

Refer to appendix B, Parameter Allocation Examples, for examples of parameter allocation.

(d) Return value location

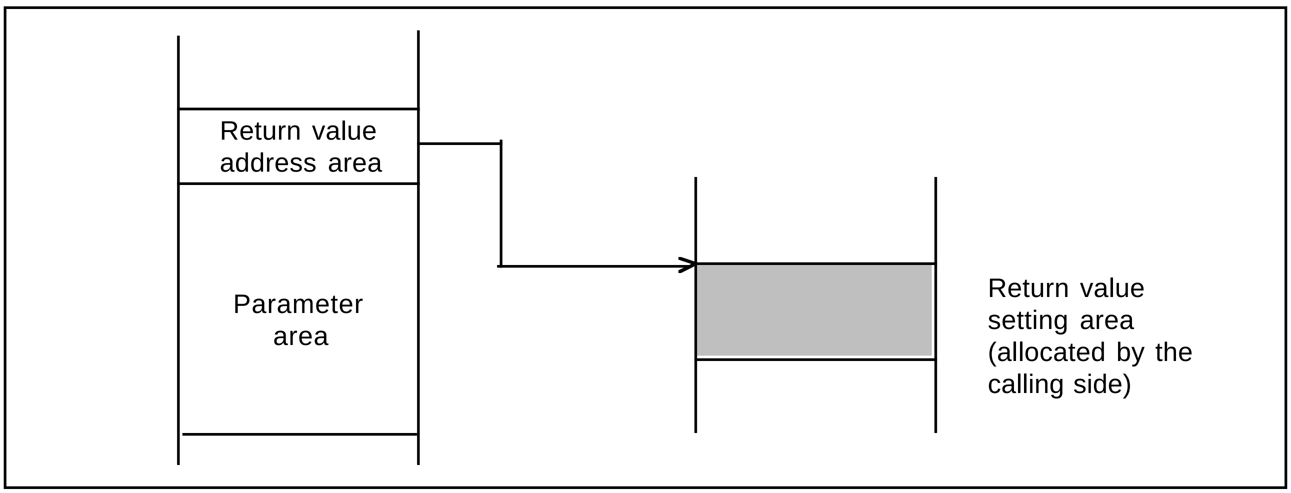
The return value is written to either a register or memory depending on its type. Refer to table 2-7 for the relationship between the return value type and location.

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 Type	Return Value Location
char, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, and Pointer	R0: 32 bits (If the return value type is **char** or **short**, perform sign extension before setting the return value in R0. If the return value type is **unsigned char** or **unsigned short**, perform zero extension before setting it in R0.)
double, **long double**, structure, and union	Return value setting area (memory)

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



Section 3 Extended Specifications

This section describes two C compiler extended specifications: interrupt functions and intrinsic functions.

3.1 Interrupt Functions

A preprocessor directive (**#pragma**) specifies an external (hardware) interrupt function. The following section describes how to create an interrupt function.

Description:

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

Table 3-1 lists interrupt specifications.

Table 3-1 Interrupt Specifications

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

Explanation: #pragma interrupt declares an interrupt function. An interrupt function 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 is specified, the TRAPA instruction is executed at the end of the function. An interrupt function with no specifications is processed in the usual procedure. The stack switching specification and the trap-instruction return specification can be specified together.

Example:

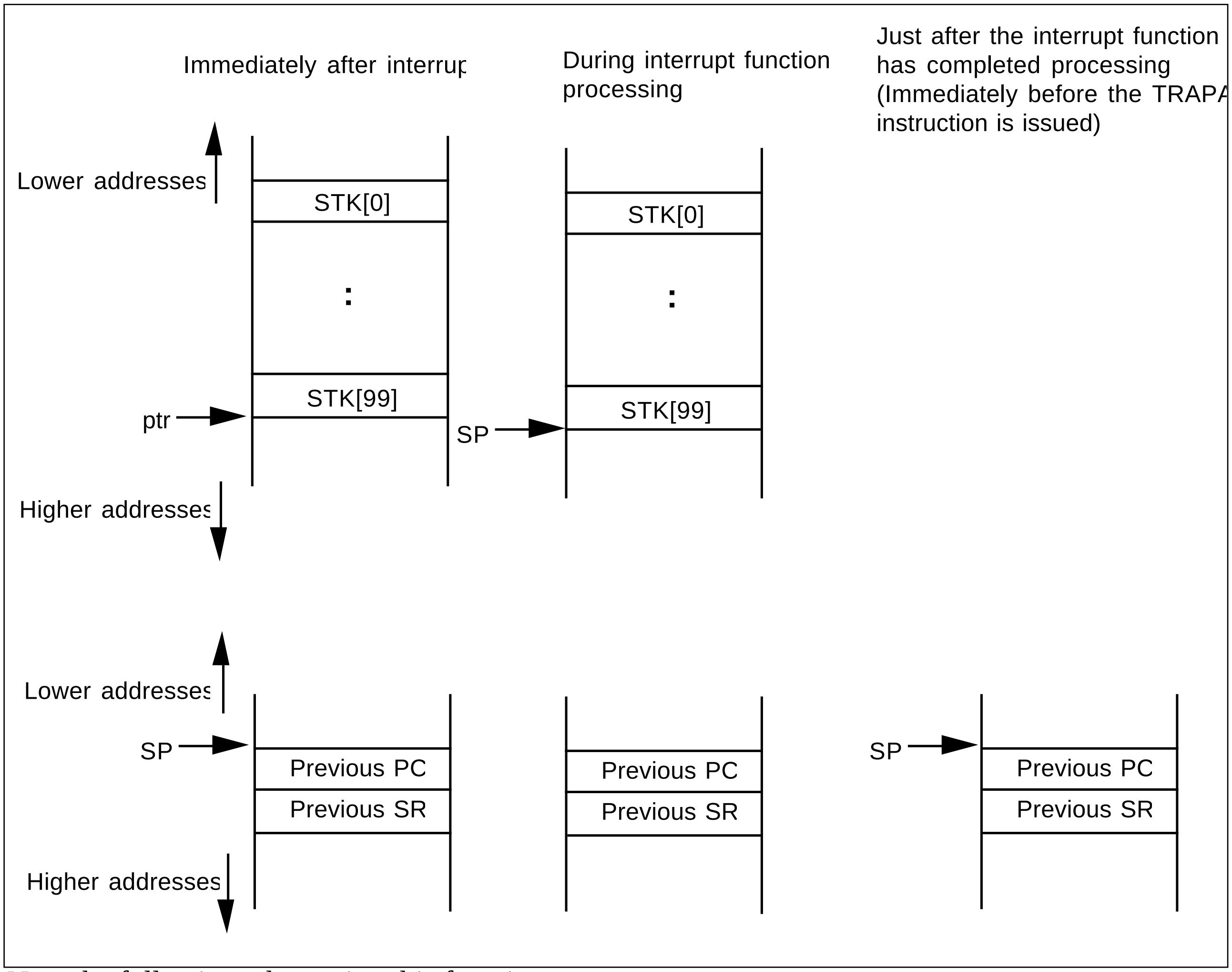
```
extern int  STK[100];

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

Explanation:

V Stack switching specification: ptr is set as the stack pointer used by interrupt function f.

W Trap-instruction return specification: After the interrupt function has completed its processing, TRAPA #10 is executed. The SP at the beginning of trap exception processing shown in the figure below. After the previous PC and SR (status register) are popped from the stack by the RTE instruction in the trap routine, control is returned from the interrupt function.

Figure 3-1 Stack Processing by an Interrupt Function



Note the following when using this function.

Table 3-2 Intrinsic Functions (cont)

Warnings:

V 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(SP=100){···} ················ (a)
int f2(){···}  ····················· (b)
```

Description:

(a) is declared correctly.

(b) returns data that is not **void**, thus (b) is declared incorrectly. An error is output.

W 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)
```

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

Example (An interrupt function is not declared):

```
int f2(){ f1();} ·············· (b)
```

Description: Because function f1 is not declared as an interrupt function, an object for extern int f1(); is generated. If function f1 is declared as an interrupt function not to be compiled in the same file as f2, correct program execution is not guaranteed.

x 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)
}
```

Description: 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

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

3.2 Intrinsic Functions

In this C compiler, system control instructions of the SH microcomputer can be written in C as intrinsic functions. The following describes the intrinsic functions provided.

Intrinsic Functions: The following functions can be specified by intrinsic functions.

V Setting and referencing the status register

W Setting and referencing the vector base register

X I/O functions using the global base register

Y System instructions which do not compete with register sources in C

Description: #include <machine.h> must be specified when using intrinsic functions.

Intrinsic Function Specifications: Table 3-2 lists intrinsic functions.

Table 3-2 Intrinsic Functions

Item	Function	Specification	Description
Status register	Setting the status register	`void set_cr(int cr)`	Sets cr (32 bits) in the status register
	Referencing the status register	`int get_cr(void)`	References the status register
	Setting the interrupt mask	`void set_imask(int mask)`	Sets mask (4 bits) in the interrupt mask (4 bits)
	Referencing the interrupt mask	`int get_imask(void)`	References the interrupt mask (4 bits)
Vector base register (VBR)	Setting the vector base register	`void set_vbr(void **base)`	Sets **base (32 bits) in VBR
	Referencing the vector base register	`int **get_vbr(void)`	References VBR
Global base register (GBR)	Setting GBR	`void set_gbr(void *base)`	Sets *base (32 bits) in GBR
	Referencing GBR	`void *get_gbr(void)`	References GBR
	Referencing GBR-based byte	`unsigned char gbr_read_byte(int offset)`	References byte data (8 bits) at the address indicated by adding GBR and the offset specified
	Referencing GBR-based word	`unsigned word gbr_read_word(int offset)`	References word data (16 bits) at the address indicated by adding GBR and the offset specified

Item	Function	Specification	Description
Global base register (GBR) (cont)	Referencing GBR-based long word	`unsigned long gbr_read_long(int offset)`	References long word data (32 bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based byte	`void gbr_write_byte( int offset, unsigned char data)`	Sets data (8bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based word	`void gbr_write_word( int offset, unsigned short data)`	Sets data (16 bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based long word	`void gbr_write_word( int offset, unsigned long data)`	Sets data (32 bits) at the address indicated by adding GBR and the offset specified
	AND of GBR base	`void gbr_and_byte( int offset, unsigned char mask)`	ANDs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	OR of GBR base	`void gbr_or_byte( int offset, unsigned char mask)`	ORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	XOR of GBR base	`void gbr_xor_byte( int offset, unsigned char mask)`	XORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	TEST of GBR base	`void gbr_tst_byte( int offset, unsigned char mask)`	Checks if the byte data at the offset from GBR is 0 or not, and sets the result in the T bit
Special instruc-tions	SLEEP instruction	`void sleep(void)`	Executes the SLEEP instruction
	TAS instruction	`void tas(char *addr)`	Executes TAS.B @addr
	TRAPA instruction	`int trapa(int trap_no)`	Executes TRAPA #trap_no

Warnings: The offsets and masks shown in table 3-2, Intrinsic Functions, must be constants. Also, the specification range for offsets is +255 bytes when the access size is shown in bytes, +510 bytes when the access size is shown as a word, and +1020 bytes when the access size is shown as a long word. Masks which can be specified for performing logical operations (AND, OR, XOR, or TEST) on a location relative to GBR (global base register) must be within the range of 0 to +255. As GBR is a control register whose contents are not preserved by all functions in this C compiler, take care when changing GBR settings.

Example:

```
#include <machine.h>

#define CDATA1 0
#define CDATA2 1
#define CDATA3 2
#define SDATA1 4
#define IDATA1 8
#define IDATA2 12

struct{
        char  cdata1;          /* offset  0*/
        char  cdata2;          /* offset  1*/
        char  cdata3;          /* offset  2*/
        char  sdata1;          /* offset  4*/
        char  idata1;          /* offset  8*/
        char  idata2;          /* offset 12*/
}table;

void f()
{
    set_gbr( &table);                 /* Set the start address of table to GBR */
          :
    gbr_write_byte( CDATA2, 10);      /* Set 10 to table.cdata2.               */
    gbr_write_long( IDATA2, 100);     /* Set 100 to table.idata2.              */
          :
    if(gbr_read_byte( CDATA2) != 10)  /* Reference table.cdata2.               */
          gbr_and_byte( CDATA2, 10);  /* AND 10 and table.cdata2, and set it   */
          :                           /* to table.cdata2.                      */
    gbr_or_byte( CDATA2, 0x0F);       /* OR H'0F and table.cdata2, and set it  */
          :                           /* to table.cdata2.                      */
    sleep();                          /* Expanded to the sleep instruction     */
}
```

Effective use of intrinsic functions:

V Set the start address of a structure which is allocated to memory and frequently accessed in GBR and access its members by gbr_read_byte, gbr_write_byte, etc.

W In the case of V, byte data frequently used in logical operations should be declared within 128 bytes from the start address of the structure.

Section 4 Notes on Programming

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

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); ·············V

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

void
f(float x)
{

}
```

Since function f has a **float** parameter, a prototype must be declared as shown at `V`.

Program Whose Evaluation Order is Not Regulated: The effect of the execution is not guaranteed in a program whose execution results differ depending on the evaluation order.

Example:

`a[i]=a[++i];` ···· The value of i on the left side differs depending on whether the right side of the assignment expression is evaluated first.

`sub(++i, i);` ···· The value of i for the second parameter differs depending on whether the first function parameter is evaluated first.

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. */

     }
```

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:

```
V  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.                             */
```

W 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 Programming Development

Table 4-1 shows troubleshootings for developing programs at compilation or when debugging.

Table 4-1 Troubleshooting

Trouble	Check Points	Solution	References
Error 314, **cannot found section**, is output at linkage	The section name which is output by the C compiler must be specified in capitals in **start** option of linkage editor.	Specify the correct section name.	Part II, Programming, 2.1
Error 105, **undefined external symbol**, is output at linkage	If identifiers are mutually referenced by a C program and an assembly program, an underscore must be attached to the symbol in the assembly program.	Reference parameters with the correct parameters.	Part II, Programming, 2.3.1
	Check if the C program uses a library function.	Specify a standard library as the input library at linkage.	Standard library specification: Part II, Programming, 4.2.1 (3)
	An undefined reference symbol identifier must not start with a _ _ (A run time routine in a standard library must be used.)		Execution routine in a standard library: Part III, System Installation, 2.1 (2)
	Check if a standard I/O library function is used in the C program.	Create low level interface routines for linking.	Part III, System Installation, 4. (6)
Debugging at the C source level cannot be performed	**debug** option must be specified at both compilation and linkage.	Specify **debug** option at both compilation and linkage.	Part I, Overview and Operation, 3.3
	A linkage editor of Ver.5.0 or higher must be used.	Use a linkage editor of Ver.5.0 or higher.	

PART III

SYSTEM INSTALLATION

Section 1 Overview

Part III describes how to install object programs generated by the C compiler on a SH 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 SH system.

- Execution environment setting for a 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 are used, library must be initialized according to the execution environment specification. Specifically, if I/O function (**stdio.h**) and memory allocation function (**stdlib.h**) are used, the user must create low-level I/O routines and memory allocation routines appropriate to the user system.

Section 2 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, the size of each memory area must be determined, then the areas must be appropriately 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.

This section describes how the size of each area is determined and how to allocate an area in memory.

2.1 Static Area Allocation

2.1.1 Data to be Allocated in Static Area

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): 0x00004A Byte(s)
CONSTANT  SECTION(C): 0x000018 Byte(s)
DATA      SECTION(D): 0x000004 Byte(s)
BSS       SECTION(B): 0x000004 Byte(s)

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

Figure 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 sections to the size shown in section size information. However, if the standard library is used, the memory area used by the library functions must be added to the the memory area size of each section. The standard library includes C library functions based on C language specifications and arithmetic routines required for C program execution. Accordingly, the standard library must be linked even if library functions are not used in the C source program.

For details on memory area size used by the standard library functions, refer to the attached Standard Library Memory Stack Size Listing. The following example shows how to calculate static area size based on the section size information shown in figure 2-1.

Calculation Example

<**ctype.h**>

Function Name	Low-Level Routine	Library [*1]	Memory Size (Bytes)				Stack Size (Bytes)
			Section P	Section B	Section C	Section D	
isalnum	None	isalnum, _ctype	32	0	256	0	16
isalpha	None	isalpha, _ctype	32	0	256	0	16

Note: *1. Library functions required for linkage. The library functions include those used by the C program and the library function itself.

1. isalnum function of <**ctype.h**> is used

   Add 32 bytes to section P and 256 bytes to section C.

Section Name	Size (Bytes)		
	C Program	Library	Total
P	74	32	106
B	24	0	24
C	4	256	260
D	4	0	4

2. isalnum and isalpha functions of <**ctype.h**> are used

   When a library function is used by multiple functions, memory size required for the library need not to be duplicated. The following table shows memory size example, when library function **_ctype** is used by multiple functions.

<Library common routine>

Section Name	Memory Size (Bytes)			
	Section P	Section B	Section C	Section D
_ctype	0	0	256	0

Each section size is calculated by the following formula:

Note: *1. Section size = C program + Library 1 + Library 2 – Duplicated library

	Size (Byte)				
Section Name	**C Program**	**Library 1**	**Library 2**	**Duplicated Library**	**Total** *1
P	74	32	32	0	138
B	24	0	0	0	24
C	4	256	256	256	260
D	4	0	0	0	4
		(**isalnum**)	(**isalpha**)	(**_ctype**)	

Note: The standard library supplied by the C compiler includes C library functions (based on C language specification), and arithmetic routines (required for C program execution). The size required for run time routines must also be added to the memory area size in the same way as C library functions.

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

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

C program

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

Assembly program output by the C compiler

```
           .IMPORT   __divls     ; An external reference definition for the run time routine
           .EXPORT    _f
           .SECTION   P,CODE,ALIGN=4
_f:
           MOV        R5, R1
           MOV.L      A_divls, R2
           JSR        @R2
           MOV        R4, R0
           RTS
           NOP
A_divls:   DATA.L     __divls
           .END
```

An external reference definition (**.IMPORT**) beginning with __ indicates a run time routine. In the above example, **__divls** is a run time routine used in the C program.

2.1.3 ROM and RAM Allocation

When allocating a program to memory, 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 addressed 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.

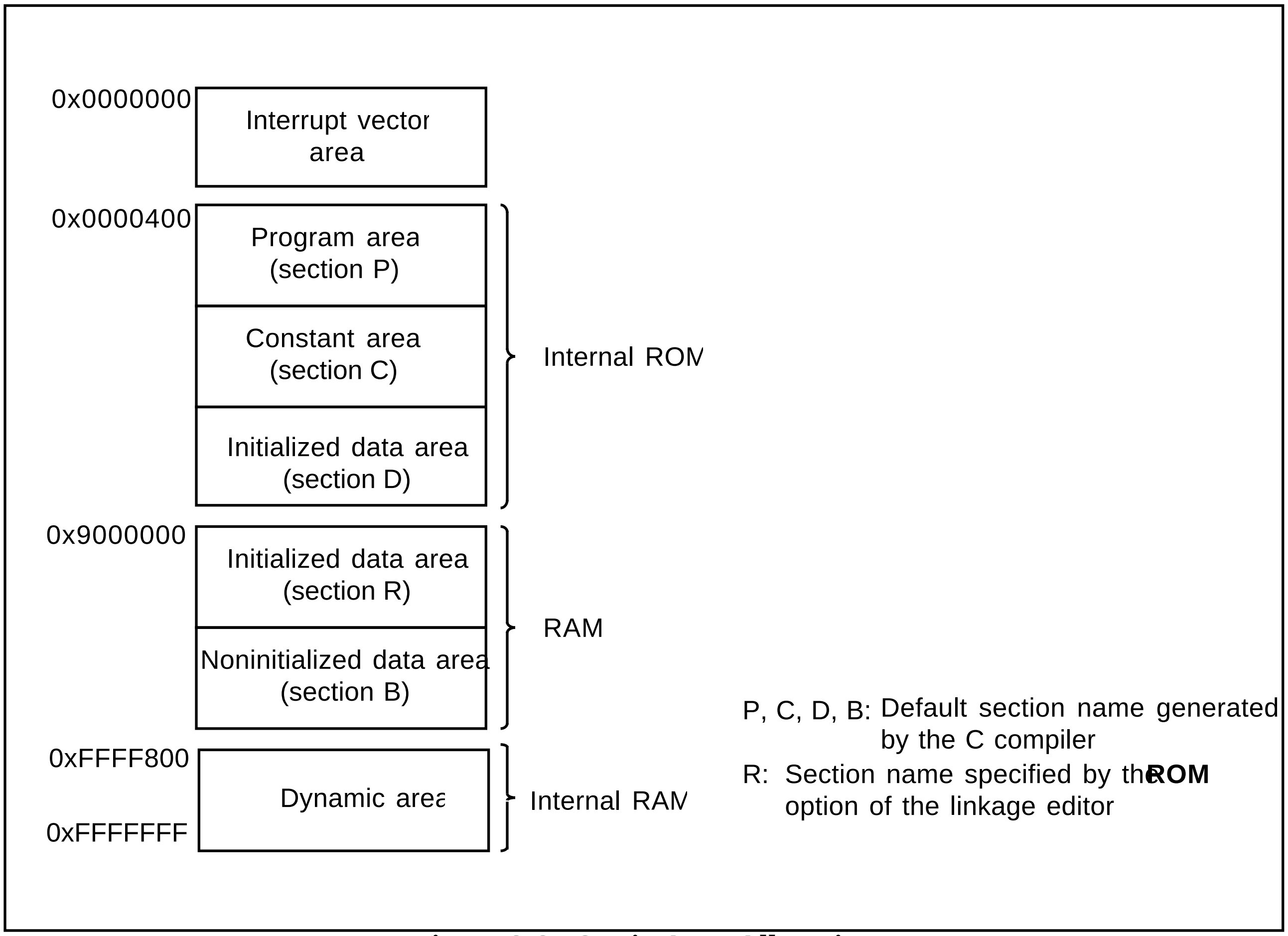


Figure 2-2 Static Area Allocation

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

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

Description:

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

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

2.2 Dynamic Area Allocation

2.2.1 Dynamic Areas

Two types of dynamic areas are used:

① Stack area

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

2.2.2 Dynamic Area Size Calculation

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

- Stack area used by each function

The size of stack used by each function can be determined from the object list (frame size) output by the C compiler. However, note that this does not account for the size of parameters to be pushed onto the stack when a function is called. Accordingly, the parameter size must be added to stack area size.

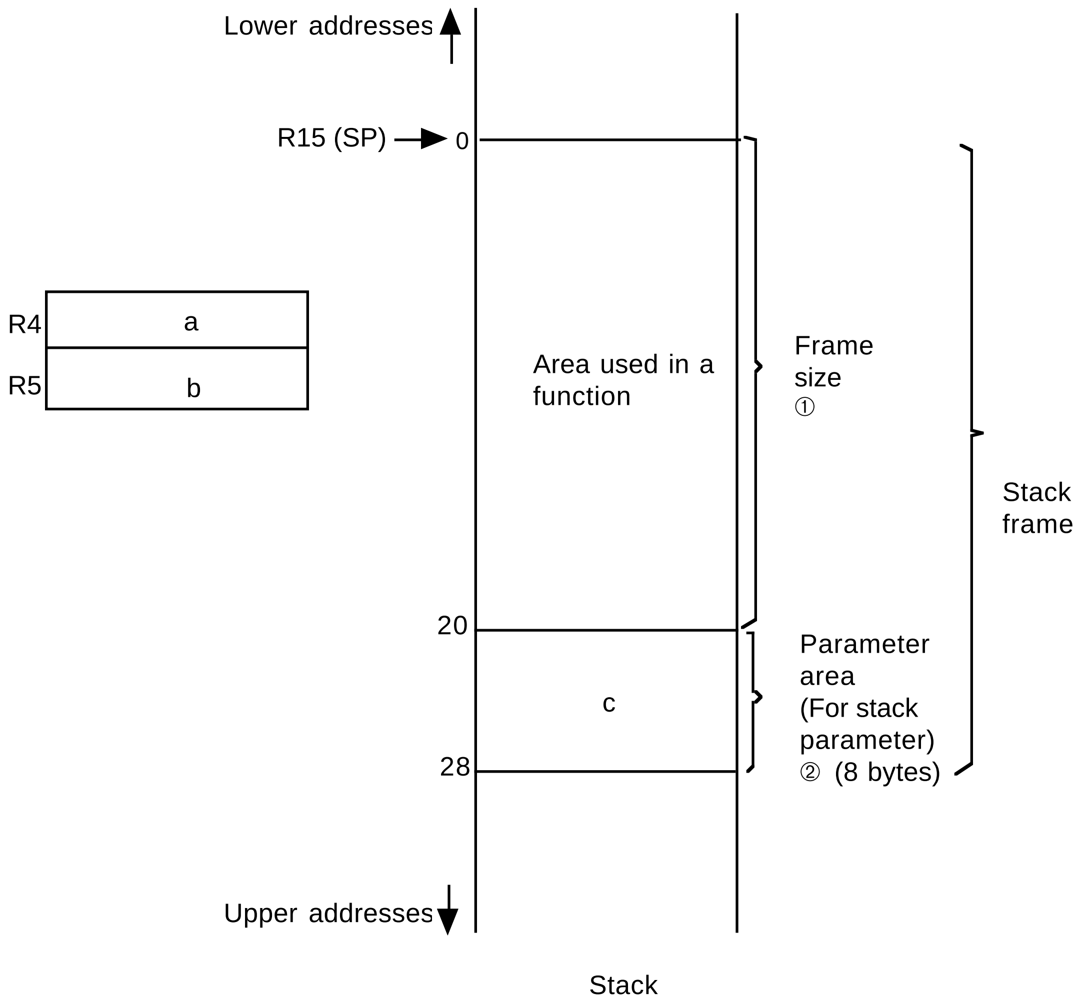
The following example shows the object list, stack allocation, and stack size calculation method.

Example:

The following shows the object list and stack size calculation in a C program.

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

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

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

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

FILE NAME: m0251.c

SCT   OFFSET    CODE          C LABEL    INSTRUCTION OPERAND     COMMENT
P
    00000000                       _h:                           ; function: h
                                                                 ; frame size=20
                                                                          ①
    00000000    2FE6                       MOV.L    R14,@-R15
    00000002    2FD6                       MOV.L    R13,@-R15
                                     :
```



The size of stack used by a function is determined by adding frame size and parameter area size (for stack parameter). Accordingly, in the above example, the stack size used by the function is calculated as follows: 20 (①) + 8 (②) = 28 bytes

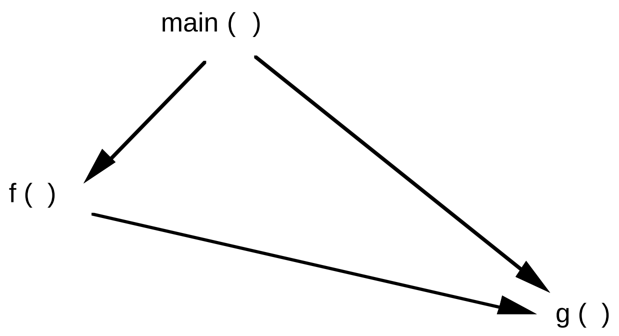
For details on the size of parameters to be pushed onto the stack, refer to the description of parameter and return value setting and referencing in section 2.3.2 of Part II.

- 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	24
f	32
g	24

Figure 2-3 Nested Function Calls and Stack Size

If funtion **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 (24) —> f(32) —> g(24)	80 bytes (Maximum size of stack area)
main (24) —> g(24)	48 bytes

As can be seen from table 2-1, the maximum size of stack area required for the longest function calling route should be determined (80 bytes in this example) and this size of memory should be allocated in RAM.

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

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

Heap Area: The total heap area required is equal to the sum of the areas to be allocated by memory management library functions (**calloc**, **malloc**, or **realloc**) in the C program. An additional 4 bytes must be summed because a 4-byte management area is used every time a memory management library function allocates an area.

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: 516 bytes x (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:

① If possible, allocate the largest area first after program execution is started.

② 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 vector table, and refer to it as SP (stack pointer). The heap area is determined by the initial specification in the low-level interface routine (**sbrk**). For details on stack and heap areas, refer to section 3.1, Vector Table Setting (**VEC_TBL**), and section 4.6, Creating Low-Level Interface Routine, 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 C Library Function Execution Environment, for details on using C library functions.

Figure 3-1 shows an example of program configuration.

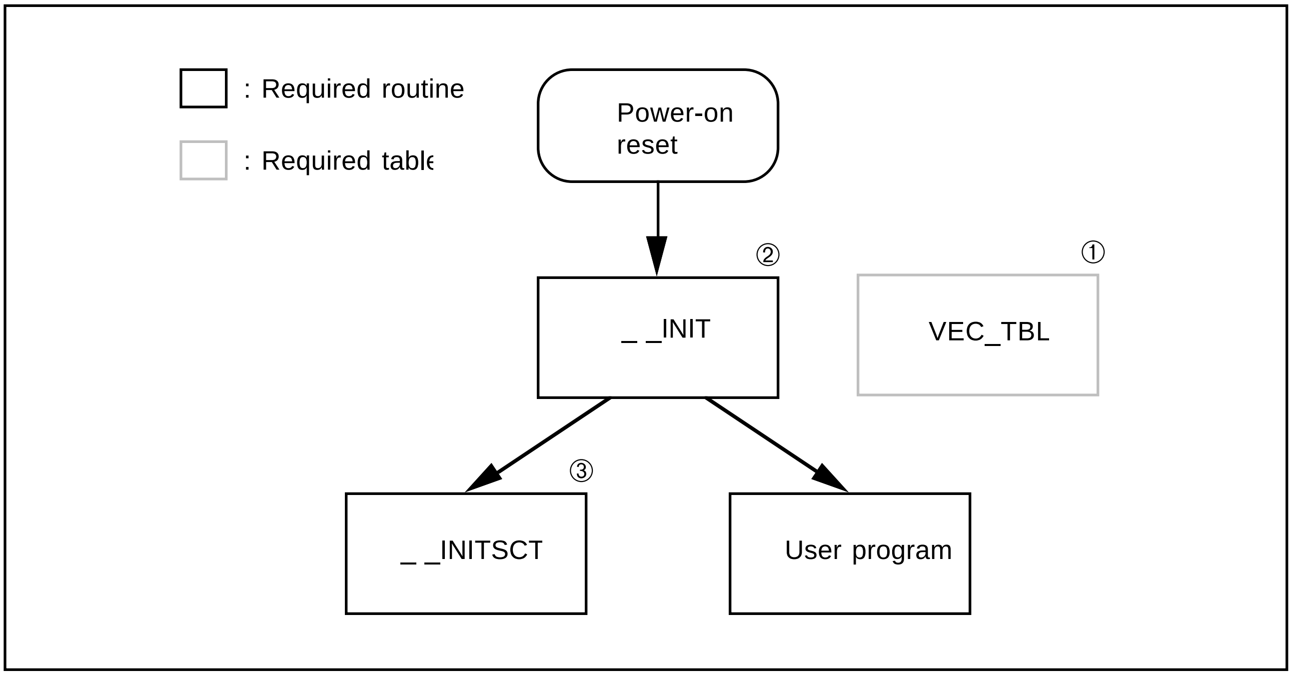


Figure 3-1   Program Configuration (No C Library Function is Used)

Each routine is described below.

① Vector table setting (**VEC_TBL**)

Sets vector table so as to initiate register initialization program **_ _INIT** and set the stack pointer (SP) 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.

How to create the above routines are described as follows.

3.1 Vector Table Setting (VEC_TBL)

To call register initialization routine **_ _INIT** at power-on reset, specify the start address of function **_ _INIT** at address 0 in the vector table. Also to specify the SP, specify the highest address of the stack to address H'4. When the user system executes interrupt handlings, interrupt vector settings are also performed in the **VEC_TBL** routine. The coding example of **VEC_TBL** is shown below.

Example:

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

3.2 Initialization (_ _INIT)

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

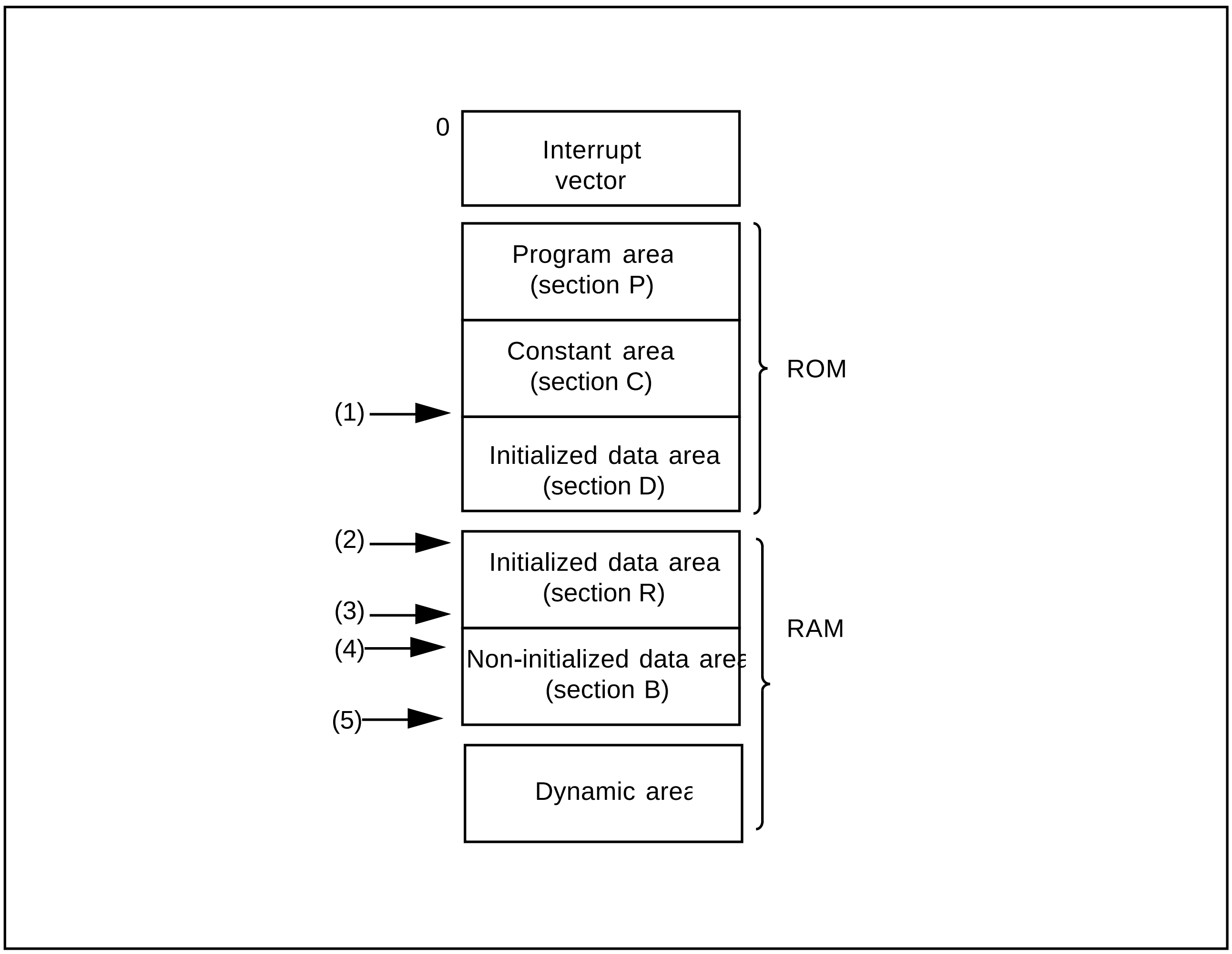
Example:

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

void   _INIT()
{
    _INITSCT();             /* Calls section initialization routine __INITSCT. */
    main();                 /* Calls main routine _main.                        */
    for( ;  ; )             /* Branches to endless loop after executing main    */
        ;                   /* function and waits for reset.                    */
}
```

3.3 Section Initialization (_ _INITSCT)

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

- Start address (1) of initialized data area in ROM.
- Start address (2) and end address (3) of inititalized 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.

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

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

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

Notes: V Section names B and D must be the non-initialized data area and initialized data area section names specified with the compiler option **section**.

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

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

Example:

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

void  _INITSCT( )
{
        short *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, they 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.

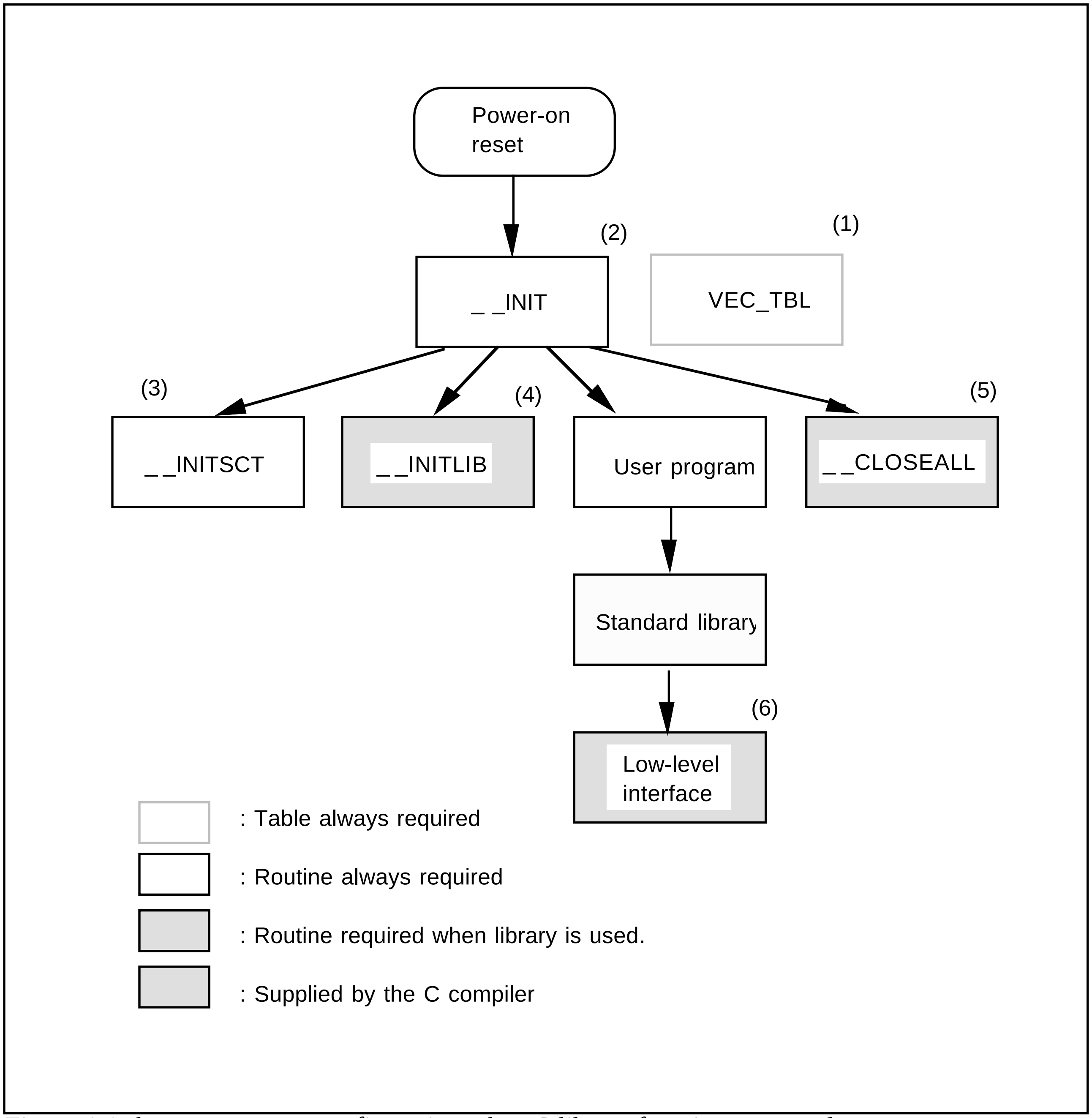


Figure 4-1 shows a program configuration when C library functions are used.

Each routine required to execute library functions as follows.

(1) Setting vector table (**VEC_TBL**)

Sets vector table to initiate register initialization program (**_ _INIT**) and set the stack pointer (SP) at power-on reset.

(2) Initializing registers (**_ _INIT**)

Initializes registers and sequentially calls the initialization routines.

(3) Initializing sections (**_ _INITSCT**)

Clears non-initialized dasta 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 addpendix D, Termination Processing Function Example.

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

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 Initializing Registers (_ _INIT)

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

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

void _INIT(void)
{
    _INITSCT();          /* Calls section initialization routine _ _INITSCT. */
    _INITLIB();          /* Calls library initialization routine _ _INITLIB. */
    main();              /* Calls C program main function.                   */
    _CLOSEALL();         /* Calls file close routine _ _CLOSEALL.            */
    for( ; ; )           /* Branches to endless loop after executing main    */
        ;                /* function and waits for reset.                    */
}
```

4.3 Initializing Sections (__INITSCT)

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

4.4 Initializing C Library Functions (__INITLIB)

Initialization must be performed for related C library functions before being used. The following description assumes the case when the initialization is performed in **__INITLIB** in the program initiation routine.

To perform initialization, the following must be considered.

(1) **errno** indicating the library error status must be initialized for all library functions.

(2) When using each function of **<stdio.h>** and **assert** macro, standard I/O library function must be initialized.

(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.

```
#include <stdlib.h>

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

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

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

Library function initialization program example is shown below.

Example:

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

```
#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 ;
 }
```

```
              /*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];
```

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

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

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

```
#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*/
}
```

Example:

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.

Figure 4-2   FILE-Type Data

4.4.2 Creating Initialization Routine (_INIT_OTHERLIB) for Other Library Function

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

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**.

```
#include <stdio.h>

void _CLOSEALL(void)      /*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]) ;
}
```

Example:

4.6 Creating Low-Level Interface Routines

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

Table 4-1 Low-Level Interface Routines

No.	Name	Explanation
1	**open**	Opens files
2	**close**	Closes files
3	**read**	Reads data from a file
4	**write**	Writes data to a file
5	**lseek**	Sets the file read/write address for data
6	**sbrk**	Allocates a memory area

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

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

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

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.

(1) Concept of I/O Operations

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

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

① File device type (console, printer, disk, etc.)
(For a special device such as a console or printer file, the user chooses a specific file name that can be recognized uniquely by the **open** routine.)

② Information such as the size and address of the buffer used for the file

③ 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.

(2) Low-Level Interface Routine Specifications

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

The interface for each routine is shown using the format below.
Create each interface routine by assuming that the prototype declaration is made.

Example:

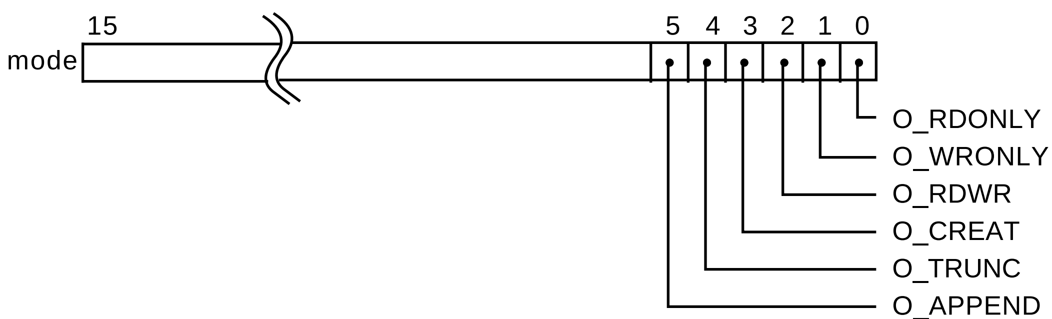
<table>
<tr><td colspan="5">(Routine name)</td></tr>
<tr><td>Purpose</td><td colspan="4">(Purpose of the routine)</td></tr>
<tr><td>Interface</td><td colspan="4">(Shows the interface as a C function declaration)</td></tr>
<tr><td rowspan="3">Parameters</td><td>No.</td><td>Name</td><td>Type</td><td>Meaning</td></tr>
<tr><td>1</td><td>(Parameter name)</td><td>(Parameter type)</td><td>(Meaning of the parameter)</td></tr>
<tr><td>⋮</td><td>⋮</td><td>⋮</td><td>⋮</td></tr>
<tr><td rowspan="3">Return value</td><td>Type</td><td colspan="3">(Type of return value)</td></tr>
<tr><td>Normal</td><td colspan="3">(Return value for normal termination)</td></tr>
<tr><td>Abnormal</td><td colspan="3">(Return value for abnormal termination)</td></tr>
</table>

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

Explanation:

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

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



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

② O_WRONLY (bit 1)
If this bit is 1, the file becomes write only.

③ O_RDWR (bit 2)
If this bit is 1, the file becomes read/write.

④ O_CREAT (bit 3)
If this bit is 1 and the file indicated by the file name does not exist, a new file is created.

⑤ O_TRUNC (bit 4)
If this bit is 1 and the file indicated by the file name exists, the file contents are discarded and the file size is set to zero.

⑥ O_APPEND (bit 5)
If this bit is 1, the read/write position is set to the end of the file. If this bit is 0, the read/write position is set to the beginning of the file.

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

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

(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:

① Base = 0

The second parameter gives the new offset relative to the beginning of the file.

② Base = 1

The second parameter is added to the current position to give the new offset.

③ Base = 2

The second parameter is added to the file size to give the new offset.

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

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

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

The C compiler checks C source programs for errors. This section explains the format and meaning of error messages that may be generated during compile time, and gives appropriate programmer responses.

1.1 Error Message Format

Error messages are output to the standard output file (normally a terminal). Figures 1-1 and 1-2 show the formats used for error messages.

```
"sample.c"   line  23 : 2011   (E)   Line too long
     V             W      X      Y         Z
```

Figure 1-1 Error Messages Format (UNIX Systems)

```
 sample.c   (23) :  2011   (E)    Line too long
     V        W       X      Y      Z
```

Figure 1-2 Error Messages Format (PC Systems)

Explanation:

V File name
File name (sample.c) of the source program in which the error was detected.

W Line number
Line number (23) where the error was detected.

X Error number
This number is unique to the error message. See section 1.3, List of Error Messages, for details on the errors and appropriate programmer responses.

Y Message level
The severity of the error. See section 1.2, Message Levels, for details.

Z Message text
This describes the error.

Note: When an error not related to the source program has occurred (e.g., an error internal to the compiler), the file name is not output; for the line number here, 0 is output in UNIX systems, and nothing is output in PC systems.

1.2 C Compiler Action and Programmer Response for Each Error Level

Error messages are classified into the following four levels according to their severity. Table 1-1 shows C compiler action for each level of errors.

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

No.	Error Level	Error Meaning	Symbol	Error Number	Object Program Output	Processing Continues	User Response
1	Warning	A mistake with respect to language specifications : The compiler has performed error recovery.	(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 necessary, modify and recompile the source program.
2	Error	A mistake in language specifications	(E)	2000 to 2999	No	Yes	Correct the error and recompile the source program.
3	Fatal	The source program exceeds the limit of the C compiler	(F)	3000 to 3999	No	No	Correct the error and recompile the source program.
4	Internal	An error has occurred in an internal process of the C compiler	—	4000 to 4999	No	No	Contact the sales office or representative where the C compiler was purchased.

1.3 List of Error Messages

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

Example:

Error Number	Message	Explanation
V 2226	W `Scalar required for an "operator"`	X Binary operator && or || is used in an expression that is not scalar. Y S: Assumes that the result is **int** and continues processing. Z P: Specify a scalar expression as the operand.

V Error Number

W Error Message
This message is sent to the standard output device (normally a terminal).

X Explanation
This gives more details about the error.

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

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

(1) Warning-Level Messages

Error No.	Message	Explanation
1000	`Illegal pointer assignment`	A pointer is assigned to a pointer with a different data type. S: Sets the left hand side to the internal representation of the right hand side pointer. The resultant 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 in "`operator`"`	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 for "`operator`"`	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 a 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.
1020	`Illegal constant`	The operands of the binary operator – in a

Error No.	Message	Explanation
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.
1010	`Character constant too long`	The length of a character constant exceeds four characters. S: Uses the first four characters and ignores the rest. P: Use character constant with four 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 literal exceeds 255. S: Uses the low order byte. P: Change the value of the escape sequence to 255 or lower.

Error No.	Message	Explanation
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.
1016	`Argument mismatch`	The data type assigned to a pointer specified as a formal parameter in a prototype declaration differs from the data type assigned to a pointer used as the corresponding actual parameter in a function call. S: Uses the internal representation of the pointer used for the function call actual parameter. P: Use a cast operator for the function call actual parameter to convert the formal 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 but 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 a cast operator for the expression specified in the **return** statement expression to convert it to the type of the function return value.
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.

Error No.	Message	Explanation
	`expression of "-"`	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.
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.

(2) 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 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 0. P: Specify the correct preprocessor directive or macro call.
2002	`Missing ","`	A comma (,) is not used to delimit two arguments in a #define directive. S: Assumes that there is a comma. P: Insert a comma.
2003	`Missing ")"`	A right parenthesis ")" does not follow a name in a defined expression. The defined expression determines whether the name is defined by a #define directive. S: Assumes that there is a right parenthesis. P: Insert a right parenthesis.
2004	`Missing ">"`	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.

Error No.	Message	Explanation
2005	`Cannot open include file "file name"`	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 that the file does not have write only status.
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	`Processor directive #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	`Processor directive #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 arguments is not equal to the number of macro definition arguments. S: Ignores the excess arguments if there are too many, or assumes blank string literals if the number of arguments is insufficient. P: Specify the correct number of macro arguments.

Error No.	Message	Explanation
2011	`Line too long`	After macro expansion, a source program line exceeds the limit of 4095 characters for UNIX systems, and 512 characters for PC systems. S: Ignores the 4096th and subsequent characters. P: Separate the line so that the length of each resulting line is within the limit after macro expansion.
2012	`Keyword as a macro name`	A preprocessor keyword is used as a macro name in a #define or #undef directive. S: Ignores the #define or #undef directive P: Change the macro name.
2013	`Processor directive #endif mismatches`	There is no #if, #ifdef, or #ifndef directive corresponding to an #endif directive. S: Ignores the #endif directive. P: Check that the #endif directive is used correctly.
2014	`Missing #endif`	There is no #endif directive corresponding to an #if, #ifdef, or #ifndef directive, and the end of 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 an #if or #elif directive exceeds the limit of 512 for UNIX systems, and 210 for PC systems. S: Assumes the value of the constant expression to be 0. P: Correct the constant expression so that the number of operators and operands is less than or equal to the limit.

Error No.	Message	Explanation
2017	`Missing ”`	A closing double quotation mark (") does not follow a file name in an #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 a #line directive exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. S: Ignores the #line directive. P: Modify the program so that the line count is less than or equal to the limit.
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.
2020	`System identifier "name" redefined`	The name of the defined symbol is the same as that of the run time routine. S: Continues processing as a unique symbol. P: Define the symbol with a different name from that of the run time routine.
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.

Error No.	Message	Explanation
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 an integer or it is negative. 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.
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.

Error No.	Message	Explanation
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: Declare the correct member type.

Error No.	Message	Explanation
2112	`Illegal bit field`	The type specifier for a bit field is illegal. **char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, 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, 16, or 32 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`	

Error No.	Message	Explanation
		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.
2119	`Not a 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 with 0`	The width of a bit field which is a member of a **struct** or **union** is 0. 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

Error No.	Message	Explanation
		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.
2126	`Illegal array size`	The value that specifies the number of elements in an array is other than an integer between 1 and 2147483647. S: Assumes the number of array elements to be one. P: Specify a valid 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 element is one. P: Specify the number of array elements.

2128	`Illegal pointer declaration for "*"`	A type specifier other than **const** or **volatile** is

Error No.	Message	Explanation
		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 of 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.
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

Error No.	Message	Explanation
		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 an 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 initialization directives. P: Delete any redundant directives.
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.
2137	`Null declaration for parameter`	An identifier is not specified in the function

Error No.	Message	Explanation
		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.
2142	`Illegal void type`	**void** is used illegally.

Error No.	Message	Explanation
		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.
2144	`Type mismatch`	**extern** variables or functions with the same names are declared with different data types in different valid ranges. S: The currently declared variable or function type is valid within the range that can be referenced. However, when linked with another file, the valid data type is determined as shown below. (1) If there is a declaration that acts as a definition, that data type is valid. (2) If there is no declaration that acts as a definition: — The previously declared data type is valid when the current declaration is in the function. — The currently declared data type is valid when the current declaration is not in the function. P: Declare the same data types for **extern** variables or functions.
2200	`Index not integer`	An array index expression type is not an

Error No.	Message	Explanation
		integer. S: Assumes that the type is **int**. P: Specify an integer expression for the array index.
2201	`Cannot convert parameter`	The nth 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.
2203	`Illegal member reference for "."`	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 for "->"`	The expression to the left of the –> operator is

Error No.	Message	Explanation
2205	`Undefined member name`	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.
2206	`Modifiable lvalue required for "`operator`"`	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.
2207	`Scalar required for "!"`	The operand for a unary prefix or suffix operator ++ or – – has a left value that cannot be assigned (a left value whose type is not array or **const**). S: Assumes that the expression with a left value that can be assigned is specified as an operand and continues processing. P: Specify an expression, whose left value can be assigned, as an operand.
2208	`Pointer required for "*"`	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.
2209	`Arithmetic type required for "`operator`"`	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**. The unary operator + or – is used on a non-arithmetic expression.

Error No.	Message	Explanation
		S: Assumes that the operand type is **int** and continues processing. P: Change the expression to an arithmetic expression.
2210	`Integer required for "~"`	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 for "operator"`	The binary operator *, /, *=, or /= is used in an expression that is not arithmetic.

Error No.	Message	Explanation
		S: Assumes **int** as the result and continues processing. P: Specify arithmetic expressions as the operands.
2214	`Integer required for "operator"`	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 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

Error No.	Message	Explanation
		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.
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 in "?:"`	The types of the second and third operands of the conditional operator ?: do not match with each other.

Error No.	Message	Explanation
		S: Assumes that the result type is **int** 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 0 or another pointer that is assigned to **void** that was converted from an integer constant whose value is 0 (5) A pointer and another pointer assigned to **void** (6) Two **struct** or **union** variables with the same data type
2220	`Modifiable lvalue required for "`operator`"`	An expression whose left value cannot be assigned (a left value whose type is not array or **const**) is used as an operand of an assignment operator =, *=, /=, %=, +=, –=, <<=, >>=, &=, ^=, or | =. 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.
2221	`Illegal type for "`operator`"`	The operand of the unary suffix operator ++ or – – is function type, a pointer assigned to **void**, or not scalar type.
2222	`Type not compatible for`	

Error No.	Message	Explanation
	"="	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. The operand types for the assignment operator = do not match.
2223	Incomplete tag used in expression	S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following five type 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 0 or another pointer that is assigned to **void** that was converted from an integer constant whose value is 0. (4) A pointer and another pointer assigned to **void** (5) Two **struct** or **union** variables with the same data type An incomplete tag name is used for a **struct** or **union** in an expression.
2224	Illegal type for assign	S: Assumes that the incomplete tag name is **int** and continues processing. P: Declare the tag name.
		The operand types of the assignment operator += or –= are illegal. S: Assumes that the result type is **int** and

<table>
<tr><th>Error No.</th><th>Message</th><th>Explanation</th></tr>
<tr><td></td><td></td><td>continues processing.
P: Specify a correct type combination of operands. Only the following two type 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.</td></tr>
<tr><td>2225</td><td>Undeclared name</td><td>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.</td></tr>
<tr><td>2226</td><td>Scalar required for "operator"</td><td>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.</td></tr>
</table>

Error No.	Message	Explanation
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 0 or another pointer assigned to **void**
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 argument list and the parentheses which indicate this list. P: Specify a function type expression or pointer assigned to a function type correctly.

Error No.	Message	Explanation
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 "`operator`"`	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.
2234	`Illegal typedef name reference`	A **typedef** name is used as a variable in an expression. S: Ignores the expression. P: Use **typedef** 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.

Error No.	Message	Explanation
2236	`Illegal cast in constant`	An attempt is made to cast a pointer with a **char** or **short**. S: Ignores the cast operation. P: Use an expression other than a constant one.
2237	`Illegal constant expression`	In a constant expression, a pointer constant is cast with an integer and the result is manipulated. 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 for "&"`	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.
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 a **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 a **switch** statement.

Error No.	Message	Explanation
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 a **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 a **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 a **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.
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.

Error No.	Message	Explanation
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 a **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 an **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 (that 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 **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.
2400	`Illegal character "`character`"`	An illegal character is detected. S: Assumes that the character is a blank character and continues processing. P: Delete the illegal character.

Error No.	Message	Explanation
2401	`Incomplete character constant`	An end of line indicator 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 indicator 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 commment`	An end of file indicator 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 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`	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	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.

Error No.	Message	Explanation
2650	`Invalid pointer reference`	The specified address does not match the required byte alignment. S: Uses the address with the lowest bit masked when accessing word data, and the address with the lowest two bits masked when accessing long word data. P: Specify the address so as to match the byte alignment.
2700	`Function "`function name`" in #pragma interrupt already declared`	A function already declared as a normal function has been specified with the interrupt function declaration #pragma interrupt. S: Ignores the interrupt function declaration. P: Declare the function as an interrupt function before it is declared as a normal function.
2701	`Multiple interrupt for one function`	A function has been declared as an interrupt function with #pragma interrupt more than once. S: Ignores the interrupt function declaration. P: Delete the declarations following the first one.
2702	`Multiple #pragma interrupt options`	The same type of interrupt specifications have been specified more than once. S: Ignore the interrupt function declaration. P: Delete one of the interrupt specifications.
2703	`Illegal #pragma interrupt declaration`	The specifications for the interrupt function declaration #pragma interrupt are not correct. S: Ignores the interrupt function declaration. P: Specify correctly.

Error No.	Message	Explanation
2704	`Illegal reference to interrupt function`	An interrupt function is illegally referenced. S: Ignores the attempt to reference the interrupt function. P: An interrupt function cannot normally be referenced. Define another function for referencing.
2705	`Illegal parameter in interrupt function`	There are different parameter types in an interrupt function. S: Ignores the interrupt function declaration. P: Specify correct parameter types.
2706	`Missing parameter declaration in interrupt function`	The variables used in the option specification by the interrupt function are not specified. S: Ignores the interrupt function declaration. P: Declare the variables before declaring the interrupt function declaration #pragma interrupt.
2707	`Parameter out of range in interrupt function`	Parameter tn in an interrupt function exceeds 256. S: Ignores the value of parameter tn. P: Modify the value of parameter tn so it does not exceed 256.
2800	`Illegal parameter number in in-line function`	The number of parameters used in an intrinsic function does not match the required number. S: Ignores the intrinsic function. P: Specify the correct number of parameters.
2801	`Illegal parameter type in in-line function`	There are different parameter types in an intrinsic function. S: Ignores the intrinsic function. P: Specify the correct parameter types.
2802	`Parameter out of range in`	A parameter exceeds the range that can be

Error No.	Message	Explanation
	`in-line function`	specified by an intrinsic function. S: Ignores the intrinsic function. P: Check the range that can be specified for the parameter and specify it correctly.
`2803`	`Invalid offset value in in-line function`	A parameter is specified improperly by an intrinsic function. S: Ignores the intrinsic function. P: Check the intrinsic function specifications and specify it correctly.

(3) Fatal-Level Messages

Error No.	Message	Explanation
3000	`Statement nest too deep`	The nesting level of an **if**, **while**, **for**, **do**, and **switch** statements exceeds the limit of 32 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3001	`Block nest too deep`	The nesting level of compound statements exceeds the limit of 32 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3002	`#if nest too deep`	The conditional compilation (#if, #ifdef, #ifndef, #elif, and #else) nesting level exceeds the limit of 32 for UNIX systems, and 6 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3003	`Too many external identifiers`	The number of external identifiers exceeds the limit of 4096 for UNIX systems, and 511 for PC systems. S: Terminates processing. P: Divide the program so that the number of external identifiers is less than or equal to the limit.

The number of effective identifiers (internal

Error No.	Message	Explanation
3004	`Too many local identifiers`	identifiers) in one function exceeds the limit of 4096 for UNIX systems, and 512 for PC systems. S: Terminates processing. P: 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 a #define directive exceeds the limit of 4096 for UNIX systems, and 1024 for PC systems. S: Terminates processing. P: Divide the program so that the number of macro names is less than or equal to the limit.
3006	`Too many parameters`	The number of parameters in either a function declaration or a function call exceeds the limit of 63 for UNIX systems, and 31 for PC systems. S: Terminates processing. P: Divide the compound statements so that the number of identifiers declared in one compound statement is less than or equal to the limit.
3007	`Too many macro parameters`	The number of parameters in a macro definition or a macro call exceeds the limit of 64 for UNIX systems, and 31 for PC systems. S: Terminates processing. P: Modify the program so that the number of macro parameters is less than or equal to the limit.

After a macro expansion, the length of a line

Error No.	Message	Explanation
3008	`Line too long`	exceeds the limit of 4095 characters for UNIX systems, and 512 characters for PC systems. S: Terminates processing. P: Divide the line so that its length does not exceed the limit after macro expansion.
3009	`String literal too long`	The length of string literals exceeds 512 characters. The length of string literals is the byte number generated after the specified string is connected continuously. The length of string literals 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 string literals does not exceeds 512 bytes.
3010	`Processor directive #include nest too deep`	The nesting level of the #include directive exceeds the limit of 8 for UNIX systems, and 5 for PC systems. S: Terminates processing. P: Ensure that the file inclusion nesting level does not exceed the limit.
3011	`Macro expansion nest too deep`	The nesting level of macro expansion performed by a #define directive exceeds the limit of 32 for UNIX systems, and 16 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level of macro expansion never exceeds the limit. Note that a macro may be defined recursively.
		The number of function definitions exceeds the

Error No.	Message	Explanation
3012	Too many function definitions	limit of 512 for UNIX systems, and 256 for PC systems. S: Terminates processing. P: Divide the program so that the number of function definitions is less than or equal to the limit in one compile unit.
3013	Too many switches	The number of **switch** statements exceeds the limit of 256 for UNIX systems, and 128 for PC systems. S: Terminates processing. P: Divide the program so that the number of **switch** statements is less than or equal to the limit in one compile unit.
3014	For nest too deep	The nesting level of **for** statements exceeds the limit of 16 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Ensure that the **for** nesting level does not exceed the limit.
3015	Symbol table overflow	The number of symbols to be generated by the C compiler exceeds the limit of 8192 for UNIX systems, and 1024 for PC systems. S: Terminates processing. P: 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 of 16384 for UNIX systems, and 2048 for PC systems. S: Terminates processing. P: Divide the file so that the number of internal labels does not exceed the limit.
		The number of **case** labels in one **switch**

Error No.	Message	Explanation
3017	`Too many case labels`	statement exceeds the limit of 511 for UNIX systems, and 255 for PC systems. S: Terminates processing. P: Ensure that the number of **case** labels does not exceed the limit.
3018	`Too many goto labels`	The number of **goto** labels defined in one function exceeds the limit of 511 for UNIX systems, and 256 for PC systems. S: Terminates processing. P: Ensure that the number of **goto** labels defined in a function does not exceed the limit.
3019	`Cannot open source file "`file name`"`	A source file cannot be opened. S: Terminates processing. P: Specify the correct file name.
3020	`Source file input error "`file name`"`	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** statements exceeds the limit of 16 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Ensure that the **switch** nesting level does not exceed the limit.

The number of types (pointer, array, and

Error No.	Message	Explanation
3023	`Type nest too deep`	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.
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 exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. 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 the limit.

The number of physical lines (including the

Error No.	Message	Explanation
3029	`Physical line overflow`	include files) exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. S: Terminates processing. P: Divide the file so that the number of physical lines does not exceed the limit.
3031	`Data size overflow`	The size of an array or a structure exceeds 2147483647. S: Terminates processing. P: Reduce the size of the array or the structure until it is less than or equal to 2147483647.
3033	`Symbol table overflow`	The number of symbols used for debug information exceeds 30719. S: Terminates processing. P: Divide the file so that the number of symbols does not exceed 30719.
3201	`Object size overflow`	The size of the object program exceeds 4 Gbytes. S: Terminates processing. P: Divide the program so that the size of the object program does not exceed 4 Gbytes.

An error has occurred in either one of the

Error No.	Message	Explanation
3300	`Cannot open internal file`	following cases: (1) An intermediate file internally generated by the C compiler cannot be opened. (2) A file having the same name as the intermediate file already exists. (3) The path name for listing file specifications exceeds 128 characters. (4) A file used internally by the C compiler cannot be opened. S: Terminates processing. P: (1) Check that the intermediate file generated by the C compiler is not being used. (2) Do not use the intermediate file name for other files. (3) Ensure that the path name for listing file specifications does not exceed 128 characters. (4) Check that the disk has sufficient capacity for files.
3301	`Cannot close internal file`	An intermediate file internally generated by the C compiler cannot be closed. S: Terminates processing. P: (1) Check that there are no mistakes in the compiler installation procedure. (2) Check that there are no abnormalities on the hard disk.
3302	`Cannot input internal file`	An intermediate file internally generated by the C compiler cannot be read. S: Terminates processing. P: (1) Check that there are no mistakes in the compiler installation procedure. (2) Check that there are no abnormalities on the hard disk. An intermediate file internally generated by

Error No.	Message	Explanation
3303	`Cannot output internal file`	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 versions in the C compiler do not match. S: Terminates processing. P: Refer to the Install Guide for the installation procedure, and reinstall the C compiler.
3320	`Command parameter buffer overflow`	The command line specification exceeds 256 characters. S: Terminates processing. P: Ensure that the command line does not exceed 256 characters.

An error has occurred in either of the

Error No.	Message	Explanation
3321	`Illegal environment variable`	following cases: (1) The environment variable SHC_LIB is not specified. (2) The file name does not satisfy file name specification rules or the path name exceeds 118 characters. S: Terminates processing. P: (1) Specify the environment variable SHC_LIB. (2) Specify the file name according to file name specification rules. (3) Ensure that the path name does not exceed 118 characters.
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 the 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     <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
}
```

Description:

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

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

Error No.	Message	Explanation	Functions to Set Error Numbers
1100	`Data out of range`	An overflow occurs.	**atan, cos, sin, tan, cosh, sinh, tanh, exp, fabs, frexp, ldexp, modf, ceil, floor, strtol, atoi, fscanf, scanf, sscanf, atol**
1101	`Data out of domain`	Results for mathematical parameters are not defined.	**acos, asin, atan2, log, log10, sqrt, fmod, pow**
1102	`Division by zero`	Division by zero was performed.	**divbs, divws, divls, divbu, divwu, divlu**
1104	`Too long string`	The length of the character string exceeds 512 characters.	**strtol, strtod, atoi, atol, atof**
1106	`Invalid file pointer`	**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**
1200	`Invalid radix`	An invalid radix was specified.	**strtol, atoi, atol**
1202	`Number too long`	The specified number exceeds 17 digits.	**strtod, fscanf, scanf, sscanf, atof**
1204	`Exponent too large`	The specified exponent exceeds three digits.	**strtod, fscanf, scanf, sscanf, atof**
1206	`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**

Error No.	Message	Explanation	Functions to Set Error Numbers
1210	`Overflow out of float`	A float-type decimal value is out of range (overflow).	**strtod, fscanf, scanf, sscanf, atof**
1220	`Underflow out of float`	A float-type decimal value is out of range (underflow).	**strtod, fscanf, scanf, sscanf, atof**
1250	`Overflow out of double`	A double-type decimal value is out of range (overflow).	**strtod, fscanf, scanf, sscanf, atof**
1260	`Underflow out of double`	A double-type decimal value is out of range (underflow).	**strtod, fscanf, scanf, sscanf, atof**
1270	`Overflow out of long double`	A long double-type decimal value is out of range (overflow).	**fscanf, scanf**
1280	`Underflow out of long double`	A long double-type decimal value is out of range (underflow).	**fscanf, scanf**
1300	`File not open`	The file is not open.	**fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen**
1302	`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**
1304	`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 of 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 C programming language).

A.1 Language Specifications of the C Compiler

A.1.1 Compilation Specifications

Table A-1 Compilation Specifications

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

A.1.2 Environmental Specifications

Table A-2 Environmental Specifications

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

A.1.3 Identifiers

Table A-3 Identifier Specifications

Item	C Compiler Specification
Number of valid characters of internal identifiers not used for external linkage	The first 31 characters are valid
Number of valid characters of external identifiers 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 with the same first 31 characters 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 character set and codes used during program execution	ASCII character set Kanji used in host environment can be used for source program comment.
Shift state used for encoding multiple-byte characters	Shift state is not supported
The number of bits used to indicate a character sets during program execution	Eight bits are used for each character.
Correspondence between the program compilation character set and the execution	ASCII is used for both.
Extended representation that appears either in a character constant or a string literal and that is not defined in the language specifications	Characters and extended representation other than that specified by the language are not supported.
Character constant or wide character constant of two or more characters	The upper four characters of the character constant is valid, and the upper two characters of the wide character is valid. If a wide character of more than one character is specified, a warning error message is output.
locale specifications used to converting multiple-byte character to wide character	**locale** is not supported
Simple **char** having normal the value range same as **signed char**	The same range as the **signed char** or **unsigned char**.

A.1.5 Integer

Table A-5 Integer Specifications

Item	C Compiler Specification
Integer-type data representation and value	Table A-6 shows data representation and value.
Effect when an integer is too large to be converted into a **signed** integer-type value or **signed char**	The lower one or two bytes of the integer is used as the conversion result.
The result of bitwise operations on **signed** integers	**signed** value
Sign of the remainder for integer division	Same as the sign of the dividend
Effect of a right shift operation on the sign bit of signed integer-type data	The sign bit is unchanged by the shift operation.

Table A-6 Integer Types and Their Corresponding Data Range

Type	Range of Values	Data Size
char	–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	–2147483648 to 2147483647	4 bytes
unsigned int	0 to 4294967295	4 bytes
long	–2147483648 to 2147483647	4 bytes
unsigned long	0 to 4294967295	4 bytes

A.1.6 Floating-Point Numbers

Table A-7 Floating-Point Number Specifications

Item	C Compiler Specification
Data that can be represented as floating-point type and value	The **float**, **double**, and **long double** are provided as floating-point types.
Data converted from **double** or **long double** to **float**	See section A.3, Floating-Point Number Specifications, for details on floating-point numbers (internal representation, conversion specifications, and operation specifications). Table A-8 shows the limits on representing floating-point numbers.
Internal representation of floating-point data	

Table A-8 Limits on Floating-Point Numbers

	Limit	
Item	**Decimal** *1	**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: *1. Limits on decimal is non-zero minimum value or maximum value not infinitive value. Values within ( ) indicate theoritical values.

A.1.7 Arrays and Pointers

Table A-9 Array and Pointer Specifications

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

A.1.8 Register

Table A-10 Register Specifications

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

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

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

Item	C Compiler Specification
Effect of 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, while structures consisting of **short** members are aligned in 2-byte units. Structures consisting of any other members are aligned in 4-byte units.*1
Sign of an **int** bit field	Assumed to be **signed int**
Allocation order of bit fields in **int** area	Beginning from the high order bit to low order bit.*2
Result when a bit field has been allocated in an **int** area and the next bit field to be allocated is larger than the remaining **int**	The next bit field is allocated to the next **int** area.*2
Type specifier allowed for bit field	**char, unsigned char, short, unsigned short, int, unsigned int, long,** and **unsigned long**
Integer describing enumeration	**int**

Notes: *1. See section 2.2 (2), Aggregate Data, in part II for details on structure member allocation.

*2. See section 2.2 (3), Bit Fields, in part II for details on bit field allocation.

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 the basic types (pointer, array, and function)	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 label specified by a switch statement	Up to 511 labels can be specified.

A.1.13 Preprocessor

Table A-15 Preprocessor Specifications

Item	C Compiler Specification
Correspondence between single character constant and execution environment characters in the conditional compilation	Character strings in the preprocessor statement match the execution environment characters
Reading an include file	The file within < > is read from a path specified by the **include** option. (Defalut: The path specified by environment variable SHC_LIB)
Supporting an include file whose name is enclosed in a pair of double quotation marks	The C compiler supports include files whose names are delimited by double quotation marks. The C compiler reads these include files from the current directory. If the include files are not in the current directory, the C compiler reads them from the directory specified in advance.
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 interrupt** is supported.[*1]
Value of _ _DATE_ _, _ _TIME_ _	Data depending on the host machine timer when the compilation starts.

Note: *1. See section 3.1, Interrupt Functions, in part II for details on #pragma interrupt specifications.

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 value 0 of pointer to **void**
Contents of macro **ptrdiff_t**	**long**

A.2.2 assert.h

Table A-17 assert.h Specifications

Item	C Compiler Specification
Information output and terminal operation of **assert.h**	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 for which the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions	Table A-19 shows the character set that results in a true return value.

Table A-19 Set of Characters that Returns True

Function Name	Characters That Become True
isalnum	'0' to '9', 'A' to 'Z', 'a' to 'z'
isalpha	'A' to 'Z', 'a' to 'z'
iscntrl	'\0' to '\037', '\177'
islower	'a' to 'z'
isprint	'\40' to '\176'
isupper	'A' to 'Z'

A.2.4 math.h

Table A-20 math.h Specifications

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

Item	C Compiler Specification
Value returned by a mathematical function if an input parameter is out of the range	Returns a nonnumeric value
Is errno set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function?	Yes, it is set.
Does a range error occur if the 2nd parameter in the fmod function is 0	A range error occurs

A.2.4 stdio.h

Table A-21 stdio.h Specifications

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

(a) Messages generated by a **perror** function follow this format:
<string-literal> : <error-message correpsonding to the error number indicated by errno>

(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 Not Supported 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
signal.h	signal, raise
stdio.h	remove, rename, tmpfile, tmpnam
stdlib.h	getenv, system
time.h	clock, difftime, time, asctime, ctime, gmtime, localtime

A.3 Floating-Point Number Specifications

A.3.1 Internal Representation of Floating-Point Numbers

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

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

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

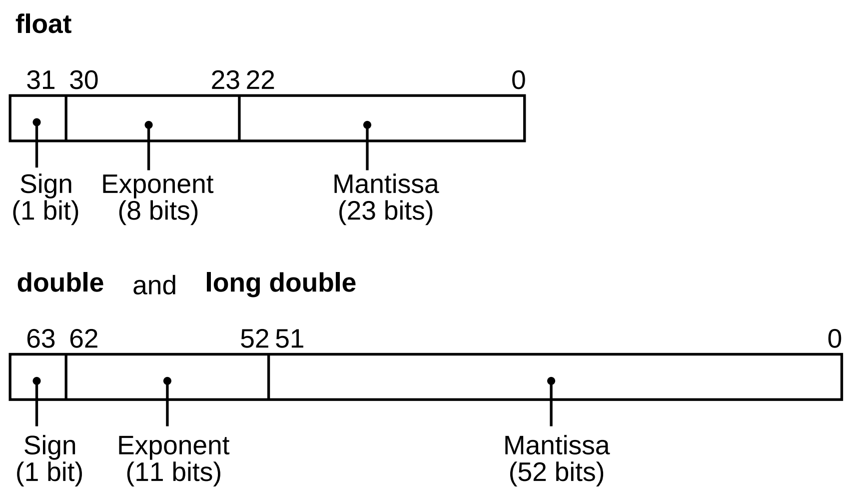


Figure A-1 Structure for the Internal Representation of Floating-Point Numbers

The elements of the structure have the following meanings.

(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 by 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, ∞/∞, ∞ − ∞).

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 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:

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

Example:

31	30 … 23	22 … 0
1	10000000	11000000000000000000000

Sign: −

Exponent: $10000000_{(2)} - 127 = 1$ ((2) indicates decimal 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:

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

Example:

31	30 … 23	22 … 0
0	00000000	11000000000000000000000

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 Specifications, for differences in each operation depending on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) (i.e., +∞ and –∞ 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 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:

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

Example:

```
63         52 51                                                  0
|0|01111111111|1110000000000000000000000000000000000000000000000000|
```

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:

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

Example:

63	62–52	51–0
1	00000000000	1110000000000000000000000000000000000000000000000000

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., +∞ and –∞ 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 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) Overflow and Underflow Handling

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 infinity has a different sign, (ii) infinity is subtracted from infinity and each infinity has the same sign, (iii) zero is multiplied by infinity, (iv) zero is divided by zero, or (v) infinity is divided by infinity. In each case, the result is not a number.

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 positive zero.

③ The operation result is always a not a number if one or both operands are not a numbers.

④ Positive zero is equal to a negative zero for 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 internal representation. The conversion between decimal representation and internal representation of ASCII string literal floating-point numbers by library functions is also explained.

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

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

To convert a floating-point number from decimal representation to normalized decimal representation, the length of the original ASCII 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 internal representation to 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 decimal representation and internal representation is too large or too small, a precise result cannot be obtained. This section explains the range of exponents for precise conversion and the error that results from exceeding the range.

a) Range of Exponents for Precise Conversion

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

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.

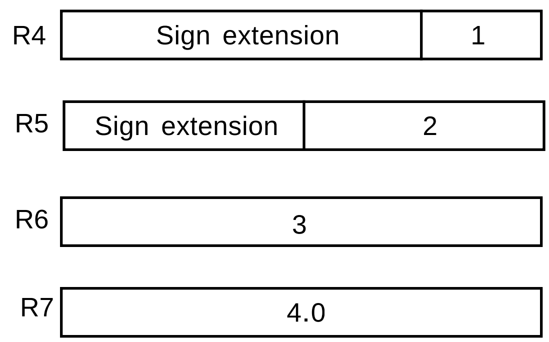
b) Conversion and Rounding Error

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 Example

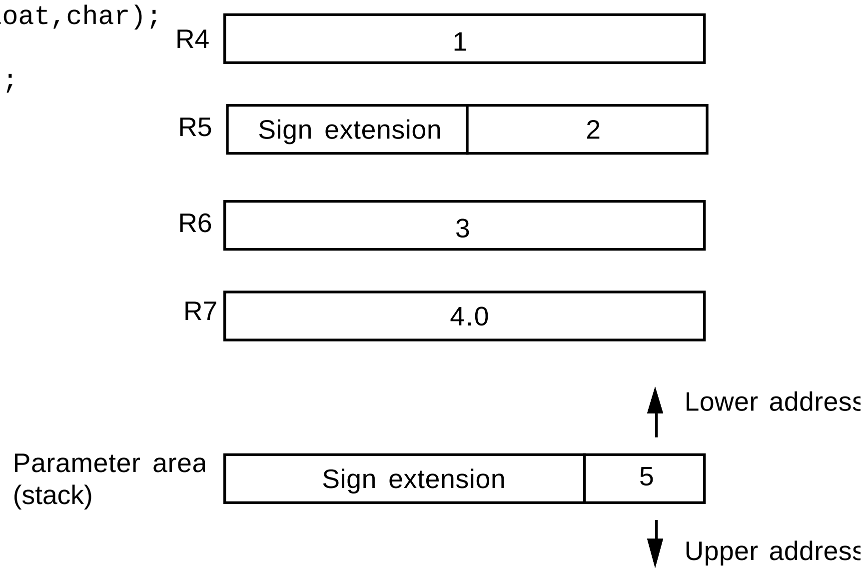
Example 1: Register parameters are allocated to registers from R4 to R7 depending on the order of declaration.

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

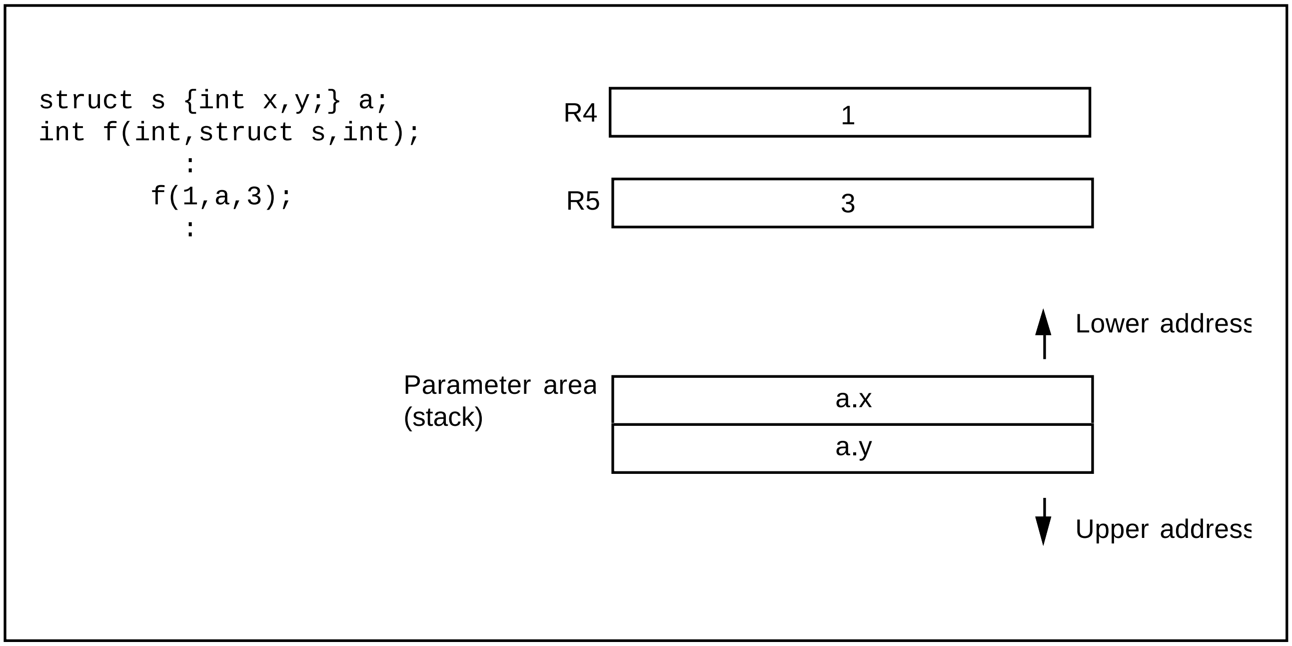


Example 2: Parameters which could not be allocated to registers from R4 to R7 are allocated to the stack area as shown below. If a **char** (**unsigned**) or **short** (**unsigned**) type parameter is allocated to a parameter area on a stack, it is extended to a 4-byte area.

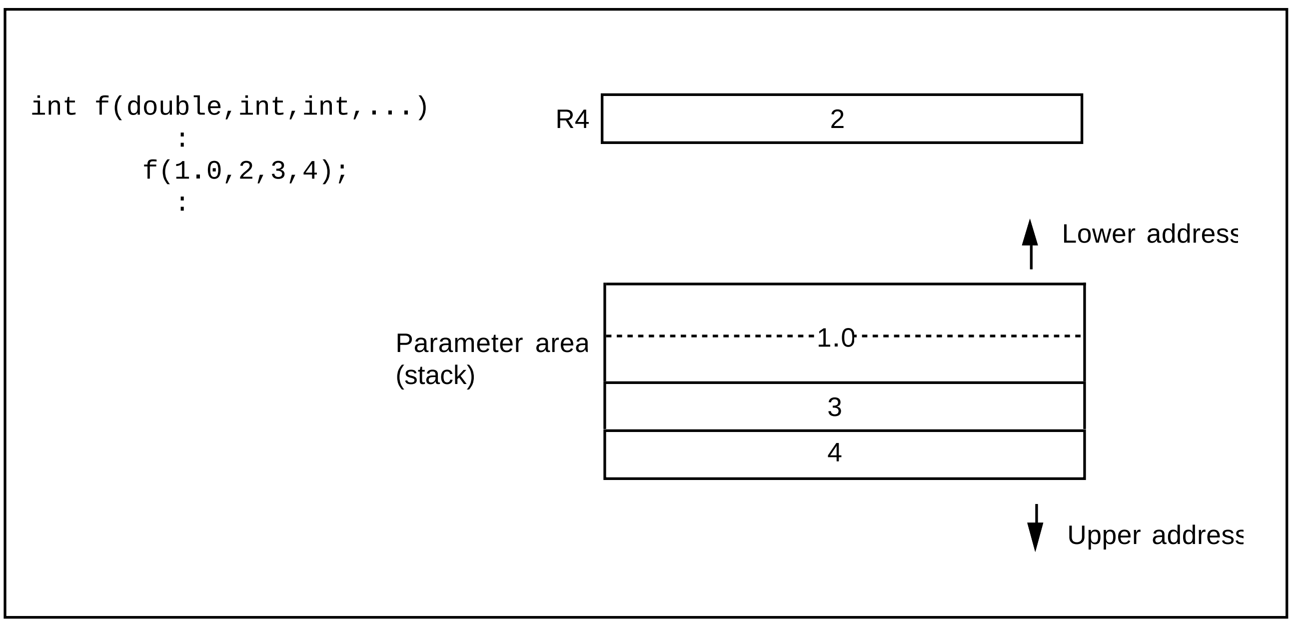
```
int f(int,short,long,float,char);
          :
      f(1,2,3,4.0,5);
          :
```



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

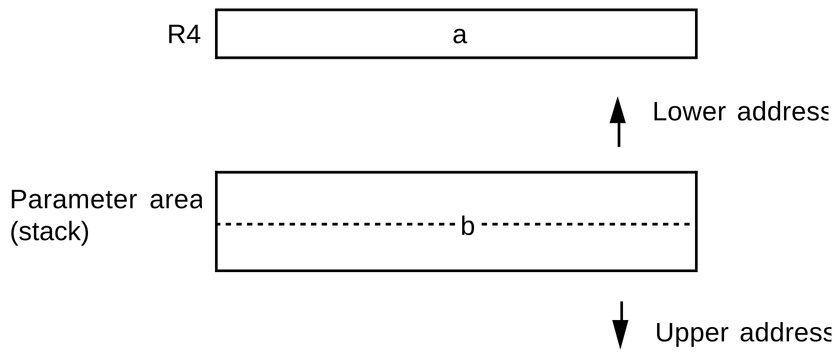


Example 4: If a function whose number of parameters changes is specified by prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding paramters are allocated to a stack.



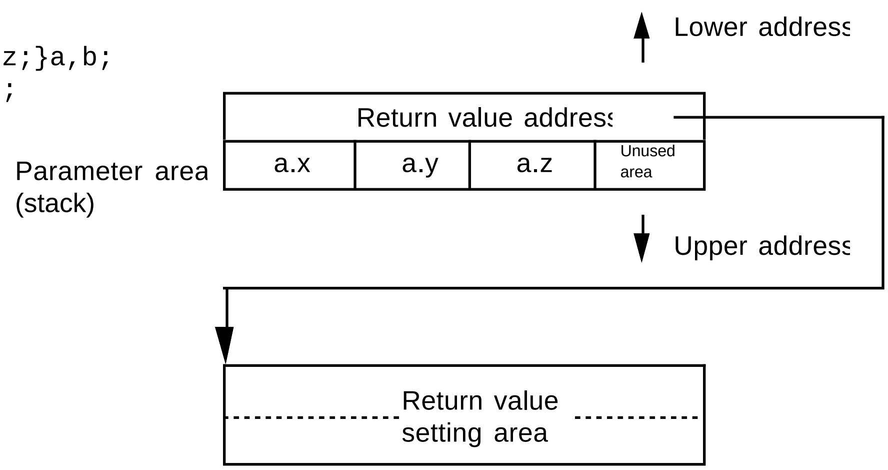
Example 5: If no prototype is declared, **char** and **float** types are extended to **int** and **double** types, respectively.

```
int f( );
char a;
float b;
          :
         f(a,b);
          :
```



Example 6: If a value returned by a function exceeds 4 bytes, or is a structure type, a return value is specified just before parameter area. If structure size is not a multiple of four, an unused area is generated.

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



Appendix C   Usage of Registers and Stack Area

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

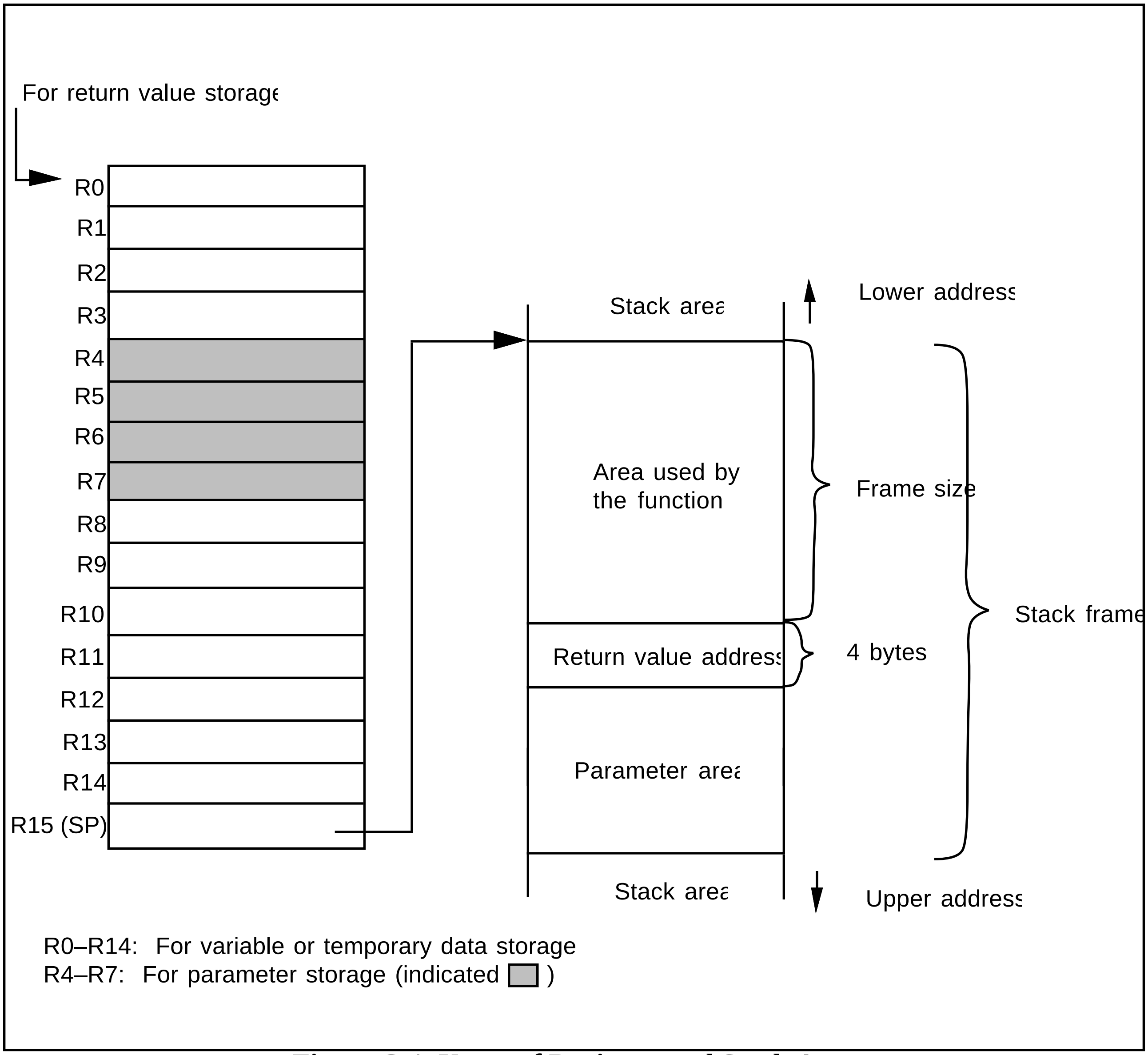


Figure C-1 Usage of Registers and Stack Area

Appendix D Creating Termination Functions

D.1 Creating Library onexit Function

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

Example:

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

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

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

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

      for(i=0; i<_onexit_count ; i++)     /*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_count++ ;
          _onexit_buf[_onexit_count]=f ;  /*Defines the function address*/
          return &_onexit_buf[_onexit_count];
     }
}
```

D.2 Creating exit Function

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

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

(1) Sets a termination code in an external variable

(2) Returns to the environment that is saved by the setjmp function immediately before calling the main function

An example of the exit function is shown below.

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

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

extern jmp_buf _init_env ;
extern int _exit_code ;

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

void exit(code)
int code ;
{
      int i;

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

      for(i=_onexit_count-1; i>0; i--) /*Sequencially executes functions defined by onexit*/
             (*_onexit_buf[i])();

      _CLOSEALL();                     /*Closes all files opened*/

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

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

```
#include <setjmp.h>

jmp_buf _init_env;
int     _exit_code;

void callmain()
{

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

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

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

D.3 Creating abort Routine

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

Example:

```
#include <stdio.h>

extern void abort(void);

extern void _CLOSEALL();

void abort()
{

      printf("program is abort !!\n"); /*Outputs message    */

      _CLOSEALL();                     /*Closes all files   */

      while(1);                        /*Enters endless loop */

}
```

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

/* file number */

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

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

/* file flag */

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

/* special character code */

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

/* size of area managed by sbrk */

#define HEAPSIZE 1024

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

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

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

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

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

```
/***********************************************************************/
/*       open:file open                                                */
/*          Return value:File number (Pass)                            */
/*                       -1          (Failure)                         */
/***********************************************************************/
int open(char *name,                    /* File name                   */
     int mode)                          /* File mode                   */
{
    /* Check mode 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 (Filure)                                    */
/***********************************************************************/
int close(int fileno)                      /* File number              */
{
      if(fileno<FLMIN || FLMAX<fileno)     /* File number range check  */
           return -1;

      flmod[fileno]=0;                     /* File mode reset          */
      return 0;
}
```

```
/*************************************************************************/
/* read:Data read                                                        */
/*      Return value:Number of read characters (Pass)                    */
/*                   -1                       (Failure)                  */
/*************************************************************************/
int read(int  fileno,                     /* File number                 */
        char *buf,                        /* Destination buffer address  */
        unsigned int  count)              /* Number of read characters   */
{
      unsigned int i;

   /*Check mode according to file name and 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)                */
/*************************************************************************/
int write(int  fileno,                    /* File number                 */
         char *buf,                       /* Destination buffer address  */
         unsigned int  count)             /* Number of write characters  */
{
       unsigned int  i;
       char c;

       /* Check mode according to file name and output each character    */

       if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   c=*buf++;
                   charput(c);
             }
             return count;
       }
       else
             return -1;
}
```

```
/*************************************************************************/
/* lseek:Definition of file read/write position                         */
/*      Return value:Offset from the top of file read/write position(Pass)*/
/*                   -1              (Failure)                           */
/*      (lseek is not supported in the console input/output)             */
/*************************************************************************/
long lseek(int  fileno,                     /* File number               */
           long offset,                     /* Read/write potision       */
           int  base)                       /* Origin of offset          */
{
     return -1;
}


/*************************************************************************/
/*     sbrk:Data write                                                   */
/*          Return value:Start address of the assigned area (Pass)       */
/*                       -1                                 (Failure)    */
/*************************************************************************/
char  *sbrk(unsigned long size)             /* Assigned area size        */

      char  *p ;

      if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size           */
           return (char *)-1 ;

      p=brk ;                               /* Area assignment           */
      brk += size ;                         /* End address update        */
      return p ;
}
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                            lowlvl.src                                |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;            SH-series simulator debugger interface routine            |
;                         -Input/output one character-                 |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          .EXPORT     _charput
          .EXPORT     _charget
SIM_IO:   .EQU            H'0080        ;Specifies TRAP_ADDRESS

          .SECTION    P, CODE, ALIGN=4

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

_charput:
          MOV.L       A_PARM, R1
          MOV         R4, R0            ;Specifies data
          MOV.B       R0, @(3, R1)
          MOV         #H'21, R0         ;Specifies function code
          MOV.B       R0, @R1
          MOV.L       A_FILENO, R0      ;Specifies file number
          MOV.B       @R0, R0
          MOV.B       R0, @(2, R1)
          MOV         R1, R0            ;Specifies parameter block address
          TRAPA       #SIM_IO
          NOP
          RTS
          NOP
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; _charget: One character input                                        |
;      C program interface: char charget(void)                         |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
          MOV.L        A_PARM, R1
          MOV          #H'20, R0        ;Specifies function code
          MOV.B        R0, @R1
          MOV.L        A_FILENO, R0     ;Specifies file number
          MOV.B        @R0, R0
          MOV.B        R0, @(2, R1)
          MOV          R1, R0           ;Specifies parameter block address
          TRAPA        #SIM_IO
          NOP
          MOV.L        A_PARM, R1
          MOV.B        @(3, R1), R0     ;References data
          RTS
          NOP

          .ALIGN       4
A_PARM:   .DATA.L      PARM             ;Parameter block address
A_FILENO: .DATA.L      FILENO           ;File number area address

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                        I/O buffer definition                         |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION    B,DATA,ALIGN=4

PARM:     .RES.L      1              ; Parameter block area
FILENO:   .RES.B      1              ; File number area

          .END
```

Appendix F ASCII Codes

UPPER 4 BITS / LOWER 4 BITS	0	1	2	3	4	5	6	7
0	NUL	LE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Index