User's Manual

CubeSuite Ver.1.40

Integrated Development Environment
User's Manual: Coding for CX Compiler

Target Device
V850 Microcontroller

Renesas Electronics
www.renesas.com

Rev.1.00 Oct 2010

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

How to Use This Manual

This manual describes the role of the CubeSuite integrated development environment for developing applications and systems for V850 microcontrollers, and provides an outline of its features.

CubeSuite is an integrated development environment (IDE) for V850 microcontrollers, integrating the necessary tools for the development phase of software (e.g. design, implementation, and debugging) into a single platform.

By providing an integrated environment, it is possible to perform all development using just this product, without the need to use many different tools separately.

Readers

This manual is intended for users who wish to understand the functions of the CubeSuite and design software and hardware application systems.

Purpose

This manual is intended to give users an understanding of the functions of the Cubesuite to use for reference in developing the hardware or software of systems using these devices.

Organization

This manual can be broadly divided into the following units.

CHAPTER 1 GENERAL
CHAPTER 2 FUNCTIONS
CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS
CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICATIONS
CHAPTER 5 LINK DIRECTIVE SPECIFICATIONS
CHAPTER 6 FUNCTIONAL SPECIFICATIONS
CHAPTER 7 STARTUP
CHAPTER 8 ROMIZATION
CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER
CHAPTER 10 CAUTIONS
APPENDIX A INDEX

How to Read This Manual

It is assumed that the readers of this manual have general knowledge of electricity, logic circuits, and microcontrollers.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	$\overline{XXX}$ (overscore over pin or signal name)
Note:	Footnote for item marked with Note in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Decimal … XXXX
	Hexadecimal … 0xXXXX

Related Documents

The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Document Name		Document No.
CubeSuite Integrated Development Environment User's Manual	Start	R20UT0256E
	Analysis	R20UT0265E
	Programming	R20UT0266E
	Message	R20UT0267E
	Coding for CX compiler	This manual
	Build for CX compiler	R20UT0261E
	78K0 Coding	R20UT0004E
	78K0 Build	R20UT0005E
	78K0 Debug	R20UT0262E
	78K0 Design	R20UT0006E
	78K0R Coding	U19382E
	78K0R Build	U19385E
	78K0R Debug	R20UT0263E
	78K0R Design	R20UT0007E
	V850 Coding	U19383E
	V850 Build	U19386E
	V850 Debug	R20UT0264E
	V850 Design	R20UT0257E

Caution The related documents listed above are subject to change without notice. Be sure to use the latest edition of each document when designing.

[MEMO]

[MEMO]

[MEMO]

TABLE OF CONTENTS

CHAPTER 1 GENERAL

This chapter provides a general outline of the V850 microcontroller's C compiler package (CX).

1.1 Outline

The V850 microcontroller's C compiler package (CX) is a program that converts programs described in C language or assembly language into machine language.

1.2 Special Features

The V850 microcontroller's C compiler package (CX) is equipped with the following special features.

(1) Language specifications in accordance with ANSI standard

The C language specifications conform to the ANSI standard. Coexistence with prior C language specifications (K&R specifications) is also provided.

(2) Advanced optimization

Code size and speed priority optimization for the C compiler are offered.

(3) Improvement to description ability

C language programming description ability has been improved due to enhanced language specifications.

(4) High portability

The single CX supports all microcontrollers. This makes it possible to use a uniform language specification, and facilitates porting between microcontrollers.
In addition, the industry-standard DWARF2 format is used for debugging information.

(5) Multifunctional

Static analysis and other functionality is provided via linking between CubeSuite.

1.3 Limits

(1) Compiler limits

See "(9) Translation Limit" for the limits of the compiler.

(2) Assembler limits

Table 1-1. Assembler Limits

Description	Limit
Symbol length (Token length)	4,294,967,294Note
Label length (Token length)	4,294,967,294Note
Number of symbols	4,294,967,294Note
Number of parameters in LOCAL directive	4,294,967,294Note
Number of automatically generated LOCAL directive symbols	4,294,967,294Note
Nesting levels in INCLUDE directive	4,294,967,294 Note
Total size of TIDATA.BYTE and TIBSS.BYTE relocation attribute sections	128 bytes

Description	Limit
Total size of TIDATA.WORD and TIBSS.WORD relocation attribute sections	256 bytes
ALIGN directive	Even number from 2 to less than 2e31
Number of arguments in IRP directive	4,294,967,294[Note]

Note Depends on memory of host machine on which it is running.

CHAPTER 2 FUNCTIONS

This chapter explains the programming method and how to use the expansion functions for more efficient use of the CX.

2.1 Variables (C Language)

This section explains variables (C language).

2.1.1 Allocating to sections accessible with short instructions

The V850 contains 2-byte instruction length load/store instructions. By allocating variables to sections accessible with these instructions it is possible to reduce the code size.

When defining or referencing a variable use the #pragma section and specify "tidata" as the section type.

```
#pragma section section-type
variable-declaration/definition
#pragma section default
```

Example

```
#pragma section tidata
int a = 1;              /*allocated to tidata.word attribute section*/
int b;                  /*allocated to tibss.word attribute section*/
#pragma section default
```

Remark See "#pragma section directive".

2.1.2 Changing allocated section

The default allocation sections are as follows:

- Variables with no initial value: .sbss section
- Variables with initial value: .sdata section
- const constants: .const section

To change the allocated section, specify the section type using #pragma section.

```
#pragma section section-type
variable-declaration/definition
#pragma section default
```

The relationship between section type and the section generated is as follows.

Section Type	Initial Value	Default Section Name	Section Name Change	Base Register	Access Instruction
data	Yes	.data	Possible	gp	ld/st 2 instruction
	No	.bss	Possible	gp	ld/st 2 instruction
sdata	Yes	.sdata	Possible	gp	ld/st 1 instruction
	No	.sbss	Possible	gp	ld/st 1 instruction
sedata	Yes	.sedata	Impossible	ep	lld/st 1 instruction
	No	.sebss	Impossible	ep	ld/st 1 instruction
sidata	Yes	.sidata	Impossible	ep	ld/st 1 instruction
	No	.sibss	Impossible	ep	ld/st 1 instruction
tidata_byte	Yes	.tidata.byte	Impossible	ep	sld/sst 1 instruction
	No	.tibss.byte	Impossible	ep	sld/sst 1 instruction
tidata_byte	Yes	.tidata.word	Impossible	ep	sld/sst 1 instruction
	No	.tibss.word	Impossible	ep	sld/sst 1 instruction
sconst	Yes	.sconst	Impossible	r0	ld/st 1 instruction
const	Yes	.const	Possible	r0	ld/st 1 instruction
default	After this statement, any previous #pragma section will be ignored, and the default allocation will be used.				

Example

```
#pragma section sdata "mysdata"
int a = 1;              /*allocated to mysdata.sdata attribute section*/
int b;                  /*allocated to mysdata.sbss attribute section*/
#pragma section default
```

When referencing a variable using the #pragma section instruction from a function in another file (i.e. reference file), it is necessary to also specify the #pragma section instruction in the reference file and to define the affected variable as extern format.

Example File that defines a table

```
#pragma section sconst
const unsigned char table_data[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
#pragma section default
```

Example File that references a table

```
#pragma section sconst
extern const unsigned char  table_data[];
#pragma section default
```

Remark See "#pragma section directive".

2.1.3 Defining variables for use during standard and interrupt processing

Specify as volatile variables that are to be used during both standard and interrupt processing.

When a variable is defined with the volatile qualifier, the variable is not optimized and optimization for assigning the variable to a register is no longer performed. When a variable specified as volatile is manipulated, a code that always reads the value of the variable from memory and writes the value to memory after the variable is manipulated is output. The access width of the variable with volatile specified is not changed. A variable for which volatile is not specified is assigned to a register as a result of optimization and the code that loads the variable from the memory may be deleted. When the same value is assigned to variables for which volatile is not specified, the instruction may be deleted as a result of optimization because it is interpreted as a redundant instruction.

Example Source and output code when volatile has been specified

If volatile is specified for "variable a", "variable b", and "variable c", a code that always reads the values of these variables from memory and writes them to memory after the variables are manipulated is output. Even if an interrupt occurs in the meantime and the values of the variables are changed by the interrupt, for example, the result in which the change is reflected can be obtained. (In this case, interrupts may have to be disabled while the variables are manipulated, depending on the timing of the interrupt.)

When volatile is specified, the code size increases compared with when volatile is not specified because the memory has to be read and written.

```
volatile int a;
volatile int b;
volatile int c;

void func(void) {
        if(a <= 0) {
                b++;
        } else {
                c++;
        }
        b++;
        c++;
}
```

```
_func:
.BB.LABEL.0:
        callt   0
        ld.w    $_a, r12
        cmp     r0, r12
        ble     .BB.LABEL.2
.BB.LABEL.1:
        ld.w    $_c, r12
        add     1, r12
        st.w    r12, $_c
        br      .BB.LABEL.3
.BB.LABEL.2:
        ld.w    $_b, r12
        add     1, r12
        st.w    r12, $_b
.BB.LABEL.3:
        ld.w    $_b, r12
        add     1, r12
        st.w    r12, $_b
        ld.w    $_c, r13
        add     1, r13
        st.w    r13, $_c
.BB.LABEL.4:
        callt   30
```

2.1.4 Defining user port

With regards to the user port, specify volatile as in the following example to avoid optimization.

Example Port description process

```
/*1.Port macro (format) definition*/
#define DEFPORTB(addr)  (*((volatile unsigned char *)addr))   /*8-bit port*/
#define DEFPORTH(addr)  (*((volatile unsigned short *)addr))  /*16-bit port*/
#define DEFPORTW(addr)  (*((volatile unsigned int *)addr))    /*32-bit port*/

/*2.Port definition (Example: PORT1 0x00100000 8bit)*/
#define PORT1   DEFPORTB(0x00100000)    /*0x00100000  8-bit port*/

/*3. Port use*/
{
  PORT1 = 0xFF;   /*Write to PORT1*/
  a = PORT1;      /*Read from PORT1*/
}

/*4.C Compiler output code*/
  :
mov     1048576, r10
st.b    r20, [r10]

mov     1048576, r11
ld.bu   [r11], r12
  :
```

Remarks 1. By declaring a structure and assigning that structure variable to a specific section, and then assigning it to the corresponding port address in the link directive, bit access is possible in the same "X.X" format used in the CX internal region I/O register.
However, in the case of 1-bit or 8-bit access both the bit field and byte union are required, so the format becomes "X.X.X".

2. Assigning variables to sections should be performed using #pragma section or the symbol information file.

2.1.5 Defining const constant pointer

The pointer is interpreted differently depending on the "const" specified location.
To assign the const section to the sconst section, specify #pragma section sconst.

- const char *p;
  This indicates that the object (*p) indicated by the pointer cannot be rewritten.
  The pointer itself (p) can be rewritten.
  Therefore the state becomes as follows and the pointer itself is allocated to RAM (.sdata/.data).

```
*p = 0;     /*Error*/
p = 0;      /*Correct*/
```

- char *const p;
  This indicates that the pointer itself (p) cannot be rewritten.
  The object (*p) indicated by the pointer can be rewritten.
  Therefore the state becomes as follows and the pointer itself is allocated to ROM (.sconst/.const).

```
*p = 0;     /*Correct*/
p = 0;      /*Error*/
```

- const char *const p;
  This indicates that neither the pointer itself(p) nor the object (*p) indicated by the pointer can be rewritten.
  Therefore the state becomes as follows and the pointer itself is allocated to ROM (.sconst/.const).

```
*p = 0;     /*Error*/
p = 0;      /*Error*/
```

2.2 Functions

This section explains functions.

2.2.1 Changing area to be allocated to

When changing a function's section name, specify the function using the #pragma text directive as shown below.

```
#pragma text  ["section name"]  [function name[, function name]…]
```

For a text attribute section that has had its section name changed, specify the initial section name from the time the input section was created in a link directive.

Example The link directive coding method for when [#pragma text "sec1" func1] has been coded in the C source, allocating function "func1" to the independently generated text-attribute section "sec1" (segment name: FUNC1):

```
FUNC1:  !LOAD ?RX {
                sec1.text = $PROGBITS ?AX sec1.text;
};
```

When allocating a specific function to an independently specified text-attribute section using the #pragma text directive, the section name actually generated will be "(specified character string)+.text", and the section name must be entered in the link directive.

In the above example it would be "sec1.text section".

Remark See "#pragma text directive".

2.2.2 Calling away function

The C compiler uses the jarl instruction to call functions.

However, depending on the program allocation the address may not be able to be resolved, resulting in an error when linking because the jarl instruction is 22-bit displacement.

In such a case, it is possible to make the function call not depend on the displacement amount by using the C compiler's -Xfar_jump option.

This is called the far jump function.

When calling a function set as far jump, the jarl32 and jr32 instruction rather than the jarl instruction is output.

One function is described per line in the file where the -Xfar_jump option is specified. The names described should be C language function names prefixed with "_" (an underscore).

Example

```
_func_led
_func_beep
_func_motor
    :
_func_switch
```

If the following is described in place of "_function-name", all functions will be called using far jump.

```
{all_function}
```

If the following is described, all interrupt functions will be called using far jump.

```
{all_interrupt}
```

Remark See "far jump function".

2.2.3 Embedding assembler instructions

With the CX assembler instructions can be described in the following formats within C source programs.

- asm declaration

```
__asm(character string constant);
```

- #pragma directive

```
#pragma asm
        Assembler instruction
#pragma endasm
```

To use registers with an inserted assembler, save or restore the contents of the registers in the program because they are not saved or restored by the CX.

Example

```
__asm("nop");
__asm(".str \"string\\0\"");

#pragma asm
        mov     r0, r10
        st.w    r10, $_i
#pragma endasm
```

Assembler instructions written within asm declarations and between #pragma asm and #pragma endasm directives are never expanded even if the assembler source contains material defined by C language #define.

Furthermore assembler instructions written within asm declarations and between #pragma asm and #pragma endasm directives are not expanded even if the -P option is added in the C compiler because they are passed as is to the assembler.

Remark See "Describing assembler instruction".

2.2.4 Executing in RAM

A program allocated to external ROM can be copied to internal RAM and executed in internal RAM while linking and after copying if the relative value of each section and each symbol (TP, EP, GP) is not destroyed.

Use caution, as some programs can be copied while others cannot.

After resetting, it is copied to internal RAM, and if the program is not changed, then the ROMization function can be used to easily pack the text section. The CX performs ROMization by default.

The text section can be packed with the CX.

2.3 Using Microcomputer Functions

This section explains using microcomputer functions.

2.3.1 Accessing peripheral I/O register with C language

When reading from and writing to the device's internal peripheral I/O register in C language, adding a pragma directive to the C source makes possible reading and writing using the peripheral I/O register name and bit names.

The peripheral I/O register name can be treated as a standard unsigned external variable. The & operator can also be used to obtain the address of the peripheral I/O register.

```
#pragma ioreg
        register name = ...
        bit name = ...
        ... = &register name
```

After describing the above pragma directive as above, the peripheral I/O register name becomes usable.

Example

```
#pragma ioreg

void func(void) {
        int i;
        unsigned long adr;
        P0 = 1;         /*Writes 1 to P0*/
        i = RXB0;       /*Reads from RXB0*/
        adr = &P1;      /*Obtain the address of P1*/
}
```

For peripheral I/O register bit names, the relevant bit names are limited to ones defined by the CX.
An error will therefore occur if the bit name is undefined.

Remark See "Peripheral I/O register".

2.3.2 Describing interrupt processing with C language

With the CX, the interrupt handler is specified using the "#pragma interrupt directive".
An example of the interrupt handler is shown below.

Example Non-maskable interrupt

```
#pragma interrupt NMI func1   /*non-maskable interrupt*/

void func1(void) {
  :
}
```

Example Multiple interrupt

```
#pragma interrupt INTP0 func2 multi     /*multiple-interrupt*/

void func2(void) {
  :
}
```

Remark See "Interrupt/Exception processing handler".

2.3.3 Using CPU instructions in C language

Some assembler instructions can be described in C source as Embedded functions. However, they are not described exactly as assembler instructions, but rather in the function format prepared by the CX.

Instructions that can be described as functions are shown below.

Assembler Instruction	Function	Embedded Function Description
di	Interrupt control	__DI();
ei		__EI();
nop	No operation	__nop();
halt	Stops the processor	__halt();
satadd	Saturated addition	long a, b; long __satadd(a, b);
satsub	Saturated subtraction	long a, b; long __satsub(a, b);
bsh	Halfword data byte swap	long a; long __bsh(a);
bsw	Word data byte swap	long a; long __bsw(a);
hsw	Word data halfword swap	long a; long __hsw(a);
sxb	Byte data sign extension	char a; long __sxb(a);
sxh	Halfword data sign extension	short a; long __sxh(a);
mul	Instruction that applies result of 32-bit x 32-bit signed multiplication to variable using mul instruction	long a, b; long long __mul(a, b);
mulu	Instruction that applies result of 32-bit x 32-bit signed multiplication to variable using mulu instruction	unsigned long a, b; Unsigned long long __mulu(a, b);
mul32	Instruction that assigns higher 32 bits of multiplication result to variable using mul32 instruction	long a, b; long __mul32(a, b);
mul32u	Instruction that assigns higher 32 bits of unsigned multiplication result to variable using mul32u instruction	unsigned long a, b; unsigned long __mul32u(a, b);
sasf	Flag condition setting with logical left shift	long a; unsigned int b; long __sasf(a, b);
sch0l	Bit (0) search from MSB side [V850E2V3]	long a; long __sch0l(a);
sch0r	Bit (0) search from LSB side [V850E2V3]	long a; long __sch0r(a);

Assembler Instruction	Function	Embedded Function Description
sch1l	Bit (1) search from MSB side [V850E2V3]	long a; long __sch1l(a);
sch1r	Bit (1) search from LSB side [V850E2V3]	long a; long __sch1r(a);
ldsr	Loads to system register [V850E2V3]	long a; void __ldsr(regID[Note], a);
stsr	Stores contents of system register [V850E2V3]	unsigned long __stsr(regID[Note]);
ldgr	Loads to general-purpose register [V850E2V3]	long a; void __ldgr(regID[Note], a);
stgr	Stores contents of general-purpose register [V850E2V3]	unsigned long __stgr(regID[Note]);
caxi	Compare and Exchange [V850E2V3]	long *a; long b, c; void __caxi(a, b, c);

Note Specified the system register number (0 to 31) in regID.
But, don't specify 0 as regID in ldsr.

Example

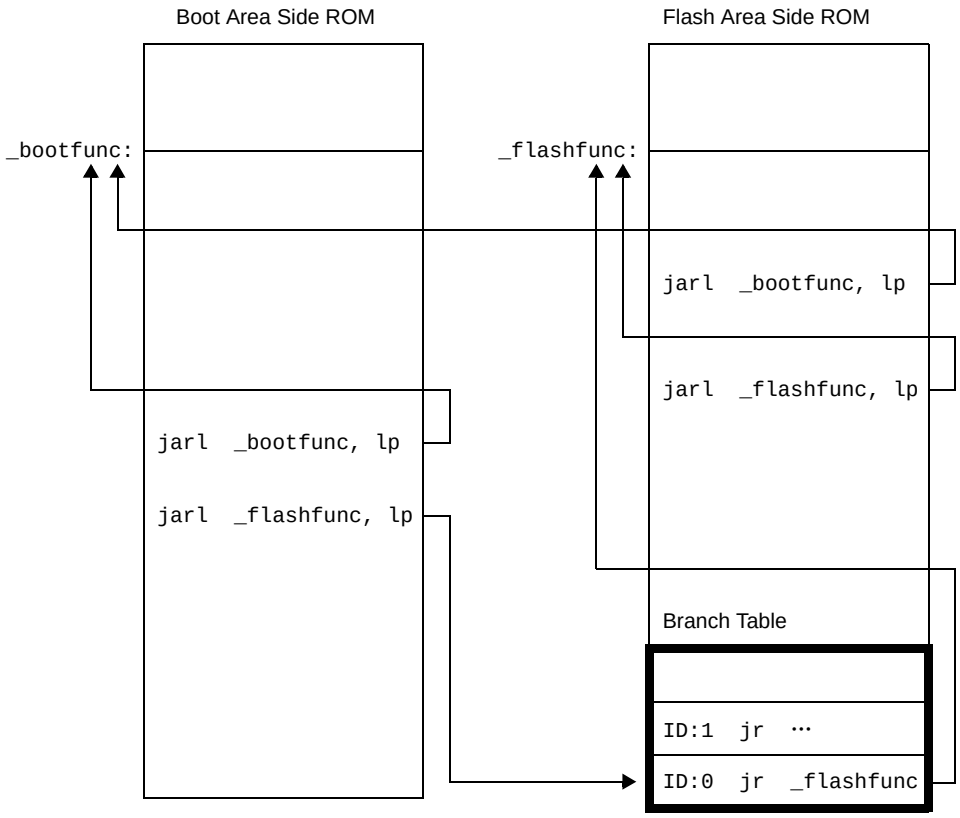
```
long a, b, c;

void func(void) {
        :
        c = __satsub(a, b);     /*The result of the saturated operation of a and b is
                                  stored in c (c = a - b)*/
        :
        __nop();
        :
}
```

2.3.4 Creating self-programming boot area

Variables and functions can be referenced between the flash area and boot area with the following operations.

- Boot area functions can be called directly from the flash area.
- Calling a function from the boot area to the flash area is performed via a branch table.
- External boot area variables can be referenced from the flash area.
- External flash area variables cannot be referenced from the boot area.
- Common external variables as well as global functions can be defined for use by both boot area programs and flash area programs. In this case the variable or function on the same area side is referenced.

Figure 2-1. Image of Flash Area/Boot Area



Flash area functions called from the boot area are defined with the ext_func directive.

```
.ext_func function name, ID number
```

Example Within a C language program

```
#pragma asm
        .ext_func _func_flash0, 0
        .ext_func _func_flash1, 1
        .ext_func _func_flash2, 2
#pragma endasm
```

Additional specifications such as options must be made.

Remark See "Boot-flash re-link function" in the "CubeSuite Build for CX Compiler" for details.

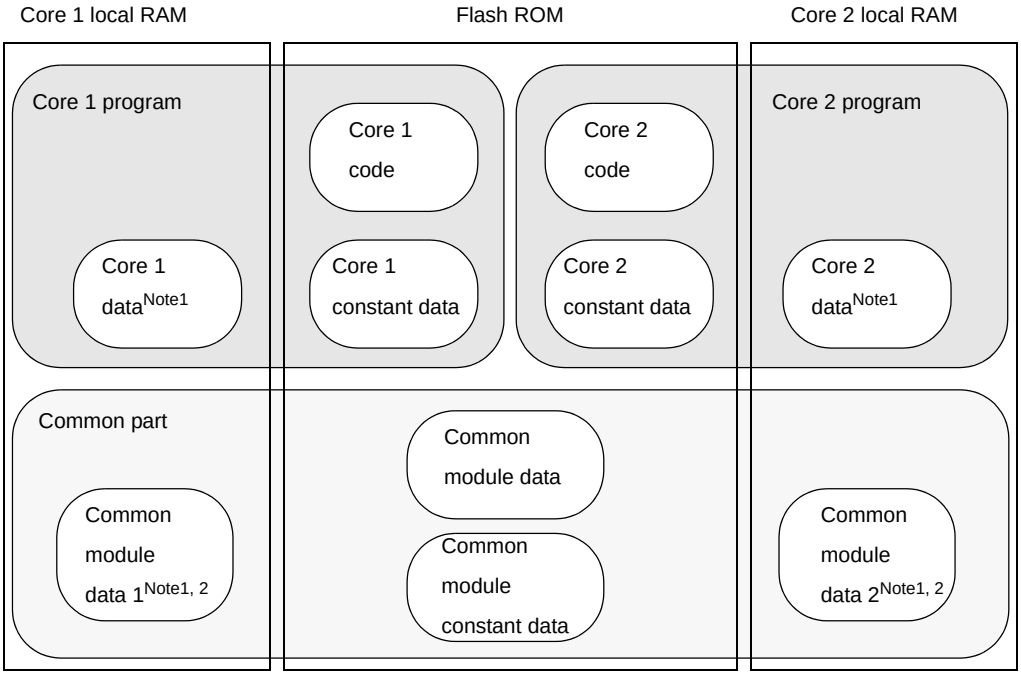
2.3.5 Creating multi-core programs

This section describes how to create multi-core programs using CX. Below is described the case when the target CPU is the uPD70F3515 (two cores).

(1) Multi-core programs

Multi-core programs output by CX are programs that run on multiple cores, which are combined into a single load module file. A multi-core program consists of programs (code/data) for each core, and a common module containing code and data that is referenced from each of the multiple cores (below, each of the core programs and the common module are called "sub-programs"). The following figure shows a sample structure of a multi-core program.

Figure 2-2. Sample Structure of a Multi-core Program



Notes 1. Core 1 data, core 2 data, and common module data can also be placed in external RAM.

2. The common module data can also be placed on the local RAM of core 1 or core 2, instead of splitting it up.

A CX multi-core program has the following features.

- Although the program has a common execution start address (0), the program subsequently branches to each of the core programs.
- Data for each core's program can be allocated to sections of all attributes in the same way as a single-core program.
- All data in the common module (except for data with const and sconst attributes) is allocated to the data-attribute section. Data and code in the common module are accessed via r0 relative instructions, rather than gp/ep/tp relative instructions.
- Data and code defined in a sub-program are accessed from other sub-programs via r0 relative instructions.
- Data and code defined in a sub-program can be accessed from that sub-program, as well as from other sub-programs. We recommend, however, that you generally use core data and code only from the sub-program in which they are defined, in consideration for the independence of core programs, and security of data access.

Care is needed when programming data that can be accessed from multiple cores, in order to prevent data from being overwritten by one core while another core is referencing it.

- Code and data are assigned to each sub-program at the source-file level (for example, it is not possible to define data for core 1 and core 2 in a single source file).

(2) Important points for coding

Take care of the following points when coding a multi-core program.

(a) C source program

Take care of the following points when coding a multi-core program in the C language.

- It is not possible to define functions with the same name in different core programs. For this reason, if you are using "main" as the name of your main functions, change the name (the default startup routine assumes that the core 1 main function will be named "main", and the core 2 main function will be named "main_pe2").
- When referencing variables or functions defigned in a core program from another sub-program, include the statement "#pragma nopic" before the extern declaration of that variable or function (in the common module, it is assumed that "#pragma nopic" is included by default). Include a "#pragma pic" statement to return to the default.
  Care is needed, however, when surrounding an extern declaration with "#pragma nopic/#pragma pic" in an include file that is used by all sub-programs. If you simply surround the extern declaration with a "#pragma nopic/#pragma pic", you could get a compilation error in your common module, or an r0 relative instruction could be generated for variable references in the same sub-program. In this case, use the pre-processor macros automatically defined when "-Xmulti" is specified to switch the source coding.
- It is not possible to specify relocation attributes other than data with "#pragma section" directives for variables defined in the common module.
  Other attributes specified in the symbol file or via the "-Xsdata" option will be ignored.

(b) Assembler source program

Take care of the following points when coding a multi-core program in assembly language.

- All data in the common module (except for data with const and sconst attributes) is allocated to the data-attribute section. Data and code in the common module must be accessed as r0 relative, rather than gp/ep/tp relative.

(3) Procedures for building a multi-core-compatible program

This section provides an example of building when there are two cores. As shown here, when there are two cores, then CX is launched four times. If there are N cores, then it will be launched N+2 times.

(a) Build the program for core 1

First, compile (assemble) and build the program for core 1. Although you do not need to perform linking at this time, be sure to specify "-Xmulti=pe1". At this stage, linking will resolve the references of symbols defined in core 1, but the references of symbols defined in core 2 and the common module will remain unresolved.
If you have a dedicated library for core 1, then perform linking at this time. However, since the "-l" option is ignored when the "-Xmulti" option is specified, you must specify the library file name directly.

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe1 file_pe1_1.c file_pe1_2.c -ope1.lmf
```

(b) Build the program for core 2

Next, compile (assemble) and build the program for core 2. This procedure is the same as for the core 1 program, but specify the option "-Xmulti=pe2".

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe2 file_pe2_1.c file_pe2_2.c -ope2.lmf
```

(c) Build the common module

Next, build the common module. As with the programs for core 1 and core 2, although you do not need to perform linking at this time, be sure to specify "-Xmulti=cmn".

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=cmn file_cmn_1.c file_cmn_2.c -ocmn.lmf
```

(d) Build each sub-program (final linking)

Finally, link each sub-program to create a single load module file. Symbol references that were unresolved in steps (a) to (c) will be resolved at this point. The startup routine and library will also be linked at this point. At this time as well, ROMization will be performed, and the hex file will be generated.

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xstartup=cstartM.obj -Xmulti_link pe1.lmf pe2.lmf cmn.lmf -otarget.lmf -
lmulti_lib
```

Remark See "CubeSuite Build for CX Compiler" for details of this option.

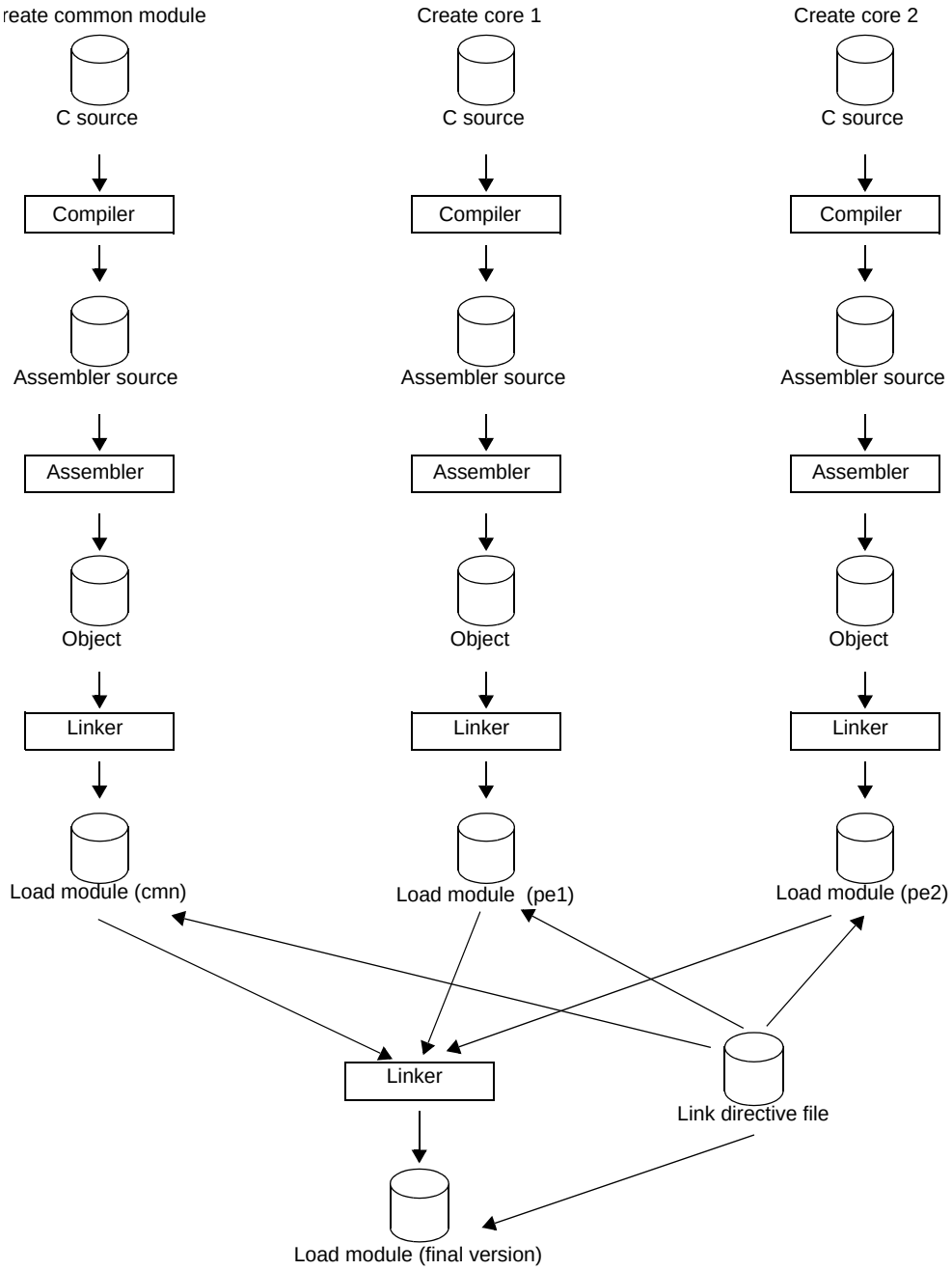
(4) The development workflow of multi-core applications

This section describes the development workflow of multi-core applications.

The development sequence described here is an example with three components: a common module, core 1 module, and core 2 module.

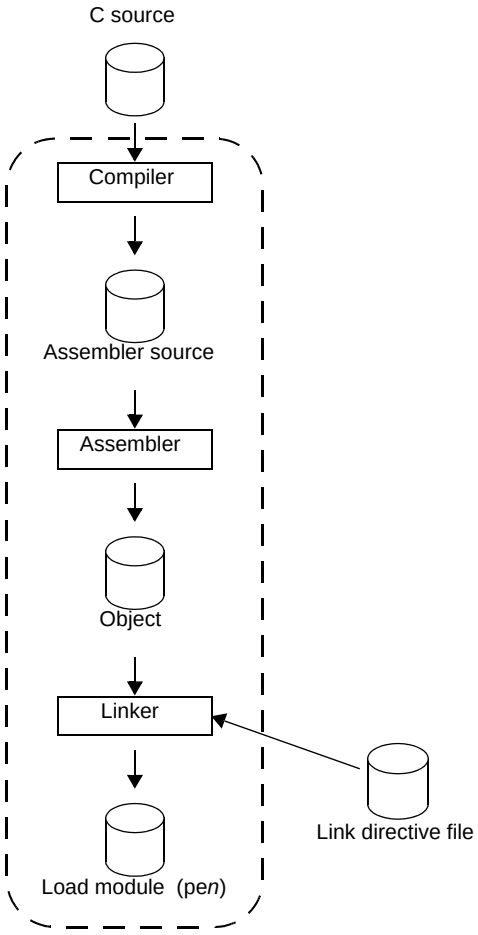
Remark A three-component architecture is not a requirement for linking. For example, it is possible to create multiple load module files for the core 1 module, and it is possible to create an application by creating the load module file for the common module or a core module only. Even in this case, however, it is not possible to omit the final process of creating a load module file by specifying the "-Xmulti_link" option.

(a) Overall development workflow



(b) Development workflow for creating a program for core *n*

> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe*n* file_pe*n*_1.c file_pe*n*_2.c -ope*n*.lmf
> (A specification of "-Xmulti=pen" is interpreted as "-Xno_startup -Xno_romize -Xrelinkable_object" also being specified simultaneously on the driver side.)



Example of C source

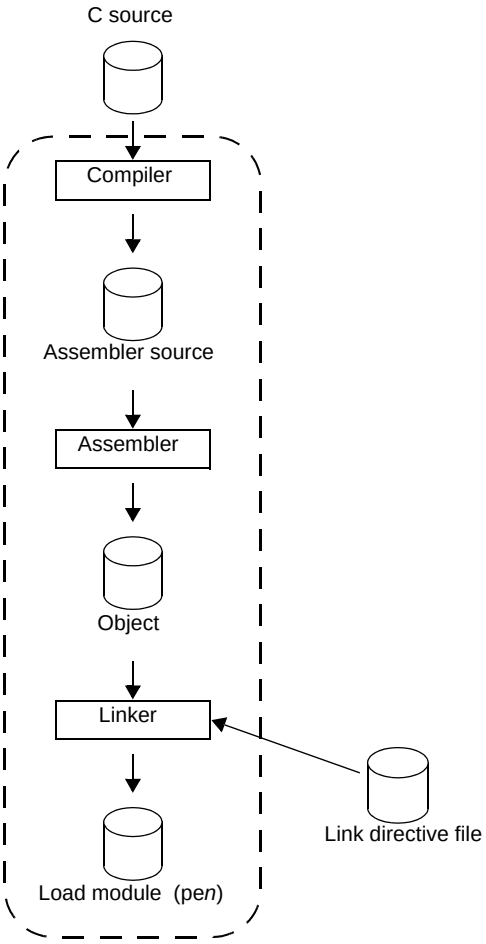
```
extern void func();

void main()
{
        func();
}

int var1 = 0;
```

Example of assembler source

```
        .extern _func
        .dseg   sdata
        .public _var1, 4
        .align  4
_var1:
        .dw     0
        .cseg   text
        .func   _main, _main.end-_main, 4
        .public _main
        .align  2
_main:
        callt   0
        jarl    _func, lp
        callt   30
_main.end:
```

(c) Development workflow for creating the common module program

> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=cmn file_cmn_1.c file_cmn_2.c -ocmn.lmf
> (A specification of "-Xmulti=cmn" is interpreted as "-Xno_startup -Xno_romize -Xrelinkable_object" also being specified simultaneously on the driver side.)



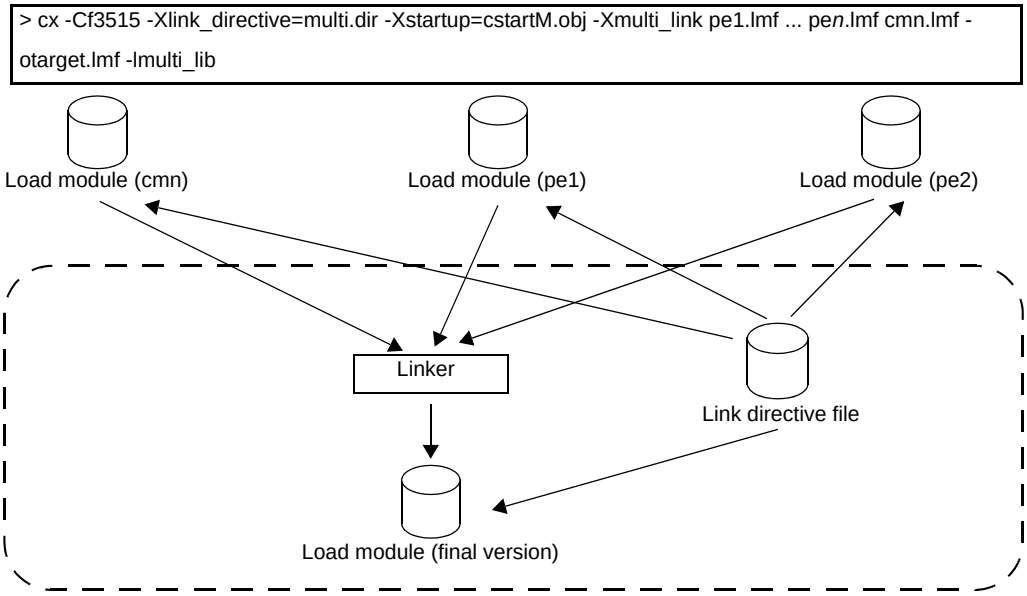
Example of C source

```
int var_cmn = 0;


void func()
{
        ...     // Processing func
}
```

Example of assembler source

```
        .dseg   data
        .public _var_cmn, 4
        .align  4
_var_cmn:
        .dw     0
        .cseg   text
        .func   _func, _func.end-_func, 0
        .public _func
        .align  2
_func:
        ...      ; Processing func
        jmp     [lp]
_func.end:
```

(d) Development workflow for creating the final load module file



(e) Link directive file example

```
SCONST_CMN: !LOAD ?R {
        .sconst             = $PROGBITS ?A      .sconst;
        .sconst.cmn         = $PROGBITS ?A      .sconst.cmn;
};
SCONST_PE1: !LOAD ?R {
        .sconst.pe1         = $PROGBITS ?A      .sconst.pe1;
};
SCONST_PE2: !LOAD ?R {
        .sconst.pe2         = $PROGBITS ?A      .sconst.pe2;
};
CONST_CMN: !LOAD ?R {
        .const.cmn          = $PROGBITS ?A      .const.cmn;
        .const              = $PROGBITS ?A      .const;
};
CONST_PE1: !LOAD ?R {
        .const.pe1          = $PROGBITS ?A      .const.pe1;
};
CONST_PE2: !LOAD ?R {
        .const.pe2          = $PROGBITS ?A      .const.pe2;
};
TEXT_CMN: !LOAD ?RX {
        .pro_epi_runtime    = $PROGBITS ?AX     .pro_epi_runtime;
        .text.cmn           = $PROGBITS ?AX     .text.cmn;
        .text               = $PROGBITS ?AX     .text;
};
TEXT_PE1: !LOAD ?RX {
```

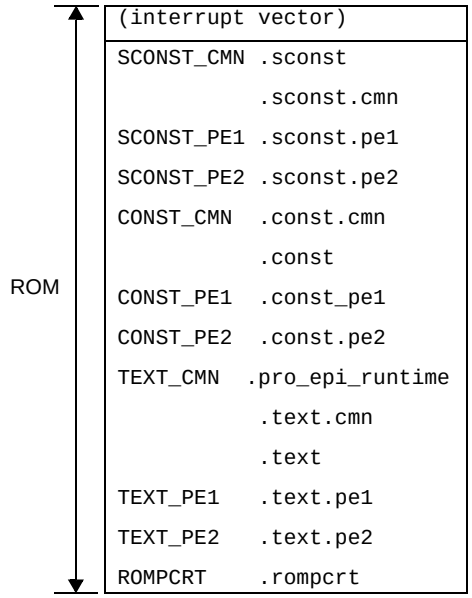
```
        .text.pe1           = $PROGBITS ?AX     .text.pe1;
};
TEXT_PE2: !LOAD ?RX {
        .text.pe2           = $PROGBITS ?AX     .text.pe2;
};
ROMPCRT: !LOAD ?RX {
        .rompcrt            = $PROGBITS ?AX     .text {rompcrt.obj};
};
DATA_PE2: !LOAD ?RW {
        .data.pe2           = $PROGBITS ?AW     .data.pe2;
        .sdata.pe2          = $PROGBITS ?AWG    .sdata.pe2;
        .sbss.pe2           = $NOBITS   ?AWG    .sbss.pe2;
        .bss.pe2            = $NOBITS   ?AW     .bss.pe2;
};
SEDATA_PE2: !LOAD ?RW {
        .sedata.pe2         = $PROGBITS ?AW     .sedata.pe2;
        .sebss.pe2          = $NOBITS   ?AW     .sebss.pe2;
};
SIDATA_PE2: !LOAD ?RW {
        .tidata.byte.pe2    = $PROGBITS ?AW     .tidata.byte.pe2;
        .tibss.byte.pe2     = $NOBITS   ?AW     .tibss.byte.pe2;
        .tidata.word.pe2    = $PROGBITS ?AW     .tidata.word.pe2;
        .tibss.word.pe2     = $NOBITS   ?AW     .tibss.word.pe2;
        .tidata.pe2         = $PROGBITS ?AW     .tidata.pe2;
        .tibss.pe2          = $NOBITS   ?AW     .tibss.pe2;
        .sidata.pe2         = $PROGBITS ?AW     .sidata.pe2;
        .sibss.pe2          = $NOBITS   ?AW     .sibss.pe2;
};
DATA_CMN: !LOAD ?RW {
        .data.cmn           = $PROGBITS ?AW     .data.cmn;
        .bss.cmn            = $NOBITS   ?AW     .bss.cmn;
};
DATA_PE1: !LOAD ?RW {
        .data.pe1           = $PROGBITS ?AW     .data.pe1;
        .sdata.pe1          = $PROGBITS ?AWG    .sdata.pe1;
        .sbss.pe1           = $NOBITS   ?AWG    .sbss.pe1;
        .bss.pe1            = $NOBITS   ?AW     .bss.pe1;
        .data               = $PROGBITS ?AW     .data;
        .sdata              = $PROGBITS ?AWG    .sdata;
        .sbss               = $NOBITS   ?AWG    .sbss;
        .bss                = $NOBITS   ?AW     .bss;
};
SEDATA_PE1: !LOAD ?RW {
        .sedata.pe1         = $PROGBITS ?AW     .sedata.pe1;
        .sebss.pe1          = $NOBITS   ?AW     .sebss.pe1;
```

```
};
SIDATA_PE1: !LOAD ?RW {
        .tidata.byte.pe1    = $PROGBITS ?AW     .tidata.byte.pe1;
        .tibss.byte.pe1     = $NOBITS   ?AW     .tibss.byte.pe1;
        .tidata.word.pe1    = $PROGBITS ?AW     .tidata.word.pe1;
        .tibss.word.pe1     = $NOBITS   ?AW     .tibss.word.pe1;
        .tidata.pe1         = $PROGBITS ?AW     .tidata.pe1;
        .tibss.pe1          = $NOBITS   ?AW     .tibss.pe1;
        .sidata.pe1         = $PROGBITS ?AW     .sidata.pe1;
        .sibss.pe1          = $NOBITS   ?AW     .sibss.pe1;
};
__tp_TEXT_PE1@%TP_SYMBOL {TEXT_PE1};
__tp_TEXT_PE2@%TP_SYMBOL {TEXT_PE2};
__gp_DATA_PE1@%GP_SYMBOL &__tp_TEXT_PE1 {DATA_PE1};
__gp_DATA_PE2@%GP_SYMBOL &__tp_TEXT_PE2 {DATA_PE2};
__ep_DATA_PE1@%EP_SYMBOL;
__ep_DATA_PE2@%EP_SYMBOL;
```

(f) Image of alignment of a multi-core program

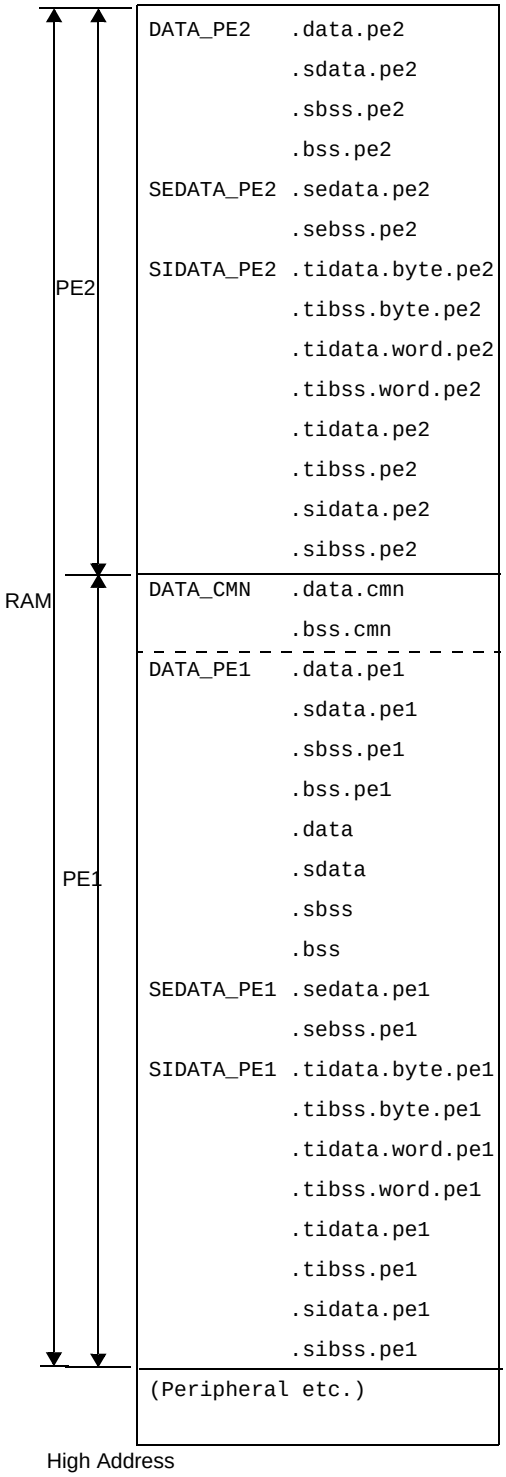
Visualizes the alignment of (e) Link directive file example (this example is for the μPD70F3515).

Image of alignment of segment/section



Link directive information

```
SCONST_CMN: !LOAD ?R {
  .sconst          = $PROGBITS ?A   .sconst;
  .sconst.cmn      = $PROGBITS ?A   .sconst.cmn;
};
SCONST_PE1: !LOAD ?R {
  .sconst.pe1      = $PROGBITS ?A   .sconst.pe1;
};
SCONST_PE2: !LOAD ?R {
  .sconst.pe2      = $PROGBITS ?A   .sconst.pe2;
};
CONST_CMN: !LOAD ?R {
  .const.cmn       = $PROGBITS ?A   .const.cmn;
  .const           = $PROGBITS ?A   .const;
};
CONST_PE1: !LOAD ?R {
  .const.pe1       = $PROGBITS ?A   .const.pe1;
};
CONST_PE2: !LOAD ?R {
  .const.pe2       = $PROGBITS ?A   .const.pe2;
};
TEXT_CMN: !LOAD ?RX {
  .pro_epi_runtime = $PROGBITS ?AX  .pro_epi_runtime;
  .text.cmn        = $PROGBITS ?AX  .text.cmn;
  .text            = $PROGBITS ?AX  .text;
};
TEXT_PE1: !LOAD ?RX {
  .text.pe1        = $PROGBITS ?AX  .text.pe1;
};
TEXT_PE2: !LOAD ?RX {
   text.pe2        = $PROGBITS ?AX  .text.pe2;
};
ROMPCRT: !LOAD ?RX {
  .rompcrt        = $PROGBITS ?AX .text {rompcrt.obj};
};
```




```
DATA_PE2: !LOAD ?RW {
  .data.pe2       = $PROGBITS ?AW  .data.pe2;
  .sdata.pe2      = $PROGBITS ?AWG .sdata.pe2;
  .sbss.pe2       = $NOBITS   ?AWG .sbss.pe2;
  .bss.pe2        = $NOBITS   ?AW  .bss.pe2;
};
SEDATA_PE2: !LOAD ?RW {
  .sedata.pe2     = $PROGBITS ?AW  .sedata.pe2;
  .sebss.pe2      = $NOBITS   ?AW  .sebss.pe2;
};
SIDATA_PE2: !LOAD ?RW {
  .tidata.byte.pe2 = $PROGBITS ?AW  .tidata.byte.pe2;
    :
  .sibss.pe2      = $NOBITS   ?AW  .sibss.pe2;
};
DATA_CMN: !LOAD ?RW {
  .data.cmn       = $PROGBITS ?AW  .data.cmn;
  .bss.cmn        = $NOBITS   ?AW  .bss.cmn;
};
DATA_PE1: !LOAD ?RW {
  .data.pe1       = $PROGBITS ?AW  .data.pe1;
    :
  .bss            = $NOBITS   ?AW  .bss;
};
SEDATA_PE1: !LOAD ?RW {
  .sedata.pe1     = $PROGBITS ?AW  .sedata.pe1;
  .sebss.pe1      = $NOBITS   ?AW  .sebss.pe1;
};
SIDATA_PE1: !LOAD ?RW {
  .tidata.byte.pe1 = $PROGBITS ?AW  .tidata.byte.pe1;
    :
  .sibss.pe1       = $NOBITS   ?AW  .sibss.pe1;
};
__tp_TEXT_PE1@%TP_SYMBOL {TEXT_PE1};
__tp_TEXT_PE2@%TP_SYMBOL {TEXT_PE2};
__gp_DATA_PE1@%GP_SYMBOL &__tp_TEXT_PE1 {DATA_PE1};
__gp_DATA_PE2@%GP_SYMBOL &__tp_TEXT_PE2 {DATA_PE2};
__ep_DATA_PE1@%EP_SYMBOL;
__ep_DATA_PE2@%EP_SYMBOL;
```

(5) Cautions

Care is needed with the following points when creating a CX multi-core program.

- Symbols with the same name cannot be defined in more than one of the load module files of the core programs or the common module. Defining symbols with the same name will cause an error during final linking.
- When creating an independent link directive file, we recommend using the same link directive file for all linking.
- If the default multi-core startup routine is used, then areas starting with the labels "__stack.pe1" and "__stack.pe2" must be secured (defined) as the stack areas for core 1 and core 2.

2.4 Variables (Assembler)

This section explains variables (Assembler).

2.4.1 Defining variables with no initial values

Use the .ds directive in a section with no initial value to allocate area for a variable with no initial value.

```
[label:]        .ds     (absolute-expression)
```

In order that it may be referenced from other files as well, it is necessary to define the label with the .public directive.

```
[label:]        .public label name[, size]
```

Example

```
        .dseg   sbss
        .public _val0, 4        -- Sets _val0 as able to be referenced from other files
        .public _val1, 2        -- Sets _val1 as able to be referenced from other files
        .public _val2, 1        -- Sets _val2 as able to be referenced from other files
        .align  4
_val0:  .ds  (4)                -- Allocates 4 bytes of area for val0
_val1:  .ds  (2)                -- Allocates 2 bytes of area for val1
_val2:  .ds  (1)                -- Allocates 1 byte of area for val2
```

2.4.2 Defining const constants with initial values

To define a const with an initial value, use the .db directives/.db2/.dhw directives/.db4/.dw directives within the .const or .sconst section.

- 1-byte values

```
[label:]        .db value
```

- 2-byte values

```
[label:]        .db2 value
```

```
[label:]        .dhw value
```

- 4-byte values

```
[label:]        .db4 value
```

```
[label:]        .dw  value
```

Example Allocates 1 halfword and stores 100

```
        .cseg   const
        .public _p, 2
        .align  4
_p:     .db2    100
```

2.4.3 Referencing section addresses

Symbols such as .data and .sdata (reserved symbols) which point to the beginnings and ends of sections are available. Therefore, utilize the appropriate symbol name when using the address value of a specified section from the assembler source.

Start symbol: __s*section-name*
End symbol: __e*section-name*

For example, the start symbol for the .sbss section is __ssbss, and its end symbol is __esbss.

These symbols can be used to retrieve the section start address and end address, but these symbol names cannot be used to make direct references with C language labels.

To retrieve these symbol values, create global variables to store these values then store the symbol values in the variables in assembler source such as that of the startup module.

By referencing these variables in the C source this can be realized.

The same applies to symbols such as __gp_DATA.

For example, the method for retrieving the start and end addresses of a .data section is as follows.

[In assembler source]

```
        .extern __sdata, 4
        .extern __edata, 4
        .dseg   sdata
        .public _data_top, 4
        .public _data_end, 4
        .align  4
_data_top:
        .ds     (4)
_data_end:
        .ds     (4)
        .cseg   text
        mov     #__sdata, r12
        st.w    r12, $_data_top
        mov     #__edata, r13
        st.w    r13, $_data_end
```

[In C source]

```
extern  int data_top;   /*extern defines data_top*/
extern  int data_end;   /*extern defines data_end*/

void func1(void) {
        int top, end;
        top = data_top;
        end = data_end;
}
```

Try using this method in cases where a C language label is used to initialize only a specified section.

2.5 Startup Routine

This section explains startup routine.

2.5.1 Securing stack area

When setting a value to the stack pointer (sp), it is necessary to pay attention to the following points.

- The stack frame is generated downwards starting from the sp set value.
- Be sure to set the sp to point at the of 4-byte boundary position.
  When the compiler references memory relative to a stack, it generates code based on the assumption the stack pointer points at the 4-byte boundary position.
  Allocate it to a data section (bss attribute section) as far as possible from gp.
  If it is near the gp, there is a chance that the program data area will be destroyed.

Example Setting sp

```
STACKSIZE       .set    0x3F0
        .dseg   bss
        .align  4
__stack:
        .ds     (STACKSIZE)
        .cseg   text
        mov     #__stack + STACKSIZE, sp
```

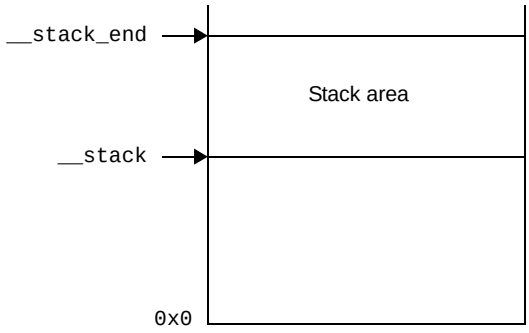
In the above example, the size of the stack frame used by the application is set to 0x3F0 bytes and area is secured.
The label "__stack" points to the lowest position (start) of the stack frame.

Because __stack is not external variable defined (via .public declaration) in the default startup module, __stack cannot be referenced from other files.
If a .public declaration is executed to __stack it becomes possible to be referenced by other files.
The stack area defines the __stack symbol to the lowest position address and sets the sum address and size of __stack to the stack pointer.
Therefore there is no symbol for the end address.
By doing the following, it becomes possible to define the next address after the stack area end address.
Use caution, as it is not the last address in the stack area.

```
STACKSIZE       .set    0x3F0
        .dseg   bss
        .public __stack         -- Add
        .public __stack_end     -- Add
        .align  4
__stack:
        .ds     (STACKSIZE)
__stack_end:
```

With the above definition, it is possible to refer to _stack and _stack_end symbols in the C source.
The mapping image becomes as follows.

Figure 2-3. Mapping Image of Stack Area



The size of the __stack symbol is specified in the startup module and should therefore be defined in C source in an array as follows.

Use caution because it is not the last address in the stack area.

```
extern unsigned long  _stack[];
```

Remark When using a label defined in the assembler in C language, one underscore is removed from the start of its name.

Assembly language definition: __stack
Reference with C language: _stack

The stack usage tracer can be used to measure C source program stack area.

2.5.2 Securing stack area and specifying allocation

This section explains securing stack area and specifying allocation.

(1) Secure stack area

In the startup routine, secure a stack in a section of a variable with no initial value with a specified section name.

Example Securing area

```
STACKSIZE       .set    0x3F0
        .stack  .dseg   bss
        .align  4
__stack:
        .ds     (STACKSIZE)
```

In the above example the section of the stack frame to be used by the application is set to .stack, the size is specified as 0x3F0 bytes and the area is secured.
The label "__stack" points to the lowest position (start) of the stack frame.

(2) Specify stack area allocation

In the link directive file specify the allocation of the section created in (1).

Example Allocation specification

```
STACK:  !LOAD ?RW V0x3FFEE00 {
        .stack  = $NOBITS ?AW .stack;
};
```

In the above example the stack segment is called STACK, and is allocated to the address 0x3FFEE00.

2.5.3 Initializing RAM

This section explains initializing RAM.

(1) Variables with no initial value

Processing to clear the .sbss and .bss sections with 0 is embedded in the default startup routine.
When clearing sections other than those above is desired, add such processing to the startup routine. When clearing, use the symbols that indicate the section start and end.

Example Clear the .tibss.byte section

```
        .extern __stibss.byte, 4        -- .tibss.byte area start symbol
        .extern __etibss.byte, 4        -- .tibss.byte area end symbol
        mov     #__stibss.byte, r13
        mov     #__etibss.byte, r12
        cmp     r12, r13
        jnl     .L20
.L21:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L21
.L20:
```

(2) RAM initialization

When a load module has been downloaded to the in-circuit emulator without performing ROMization, data with initialized values placed in regions such as the data and sdata areas are set to their values at the time of download. When using the load module without performing ROMization to debug, it is necessary to remove the RAM area initialization routine.
In the case of a ROMization load module, it is necessary to use the _rcopy copy function to perform operations such as copying data with initial values.
This processing is possible not in the startup routine but also before accessing a main function variable with an initial value, so perform it upon full completion of peripheral settings.

2.5.4 Preparing function and variable access

The text pointer is used when accessing a function, and either the global pointer or the element pointer is used when accessing a variable.

(1) Preparations for accessing function

The text pointer (tp) is a pointer prepared to implement referencing (PIC: Position Independent Code) independent of the position at which the text area of an application, i.e., program code is allocated when the program code is referenced. For example, if it is necessary to reference a specific location in the code during program execution, the CX outputs the code to be accessed in tp-relative mode.
Since the code is output on the assumption that tp is correctly set, tp must be correctly set in the startup routine.
The text pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the text pointer is described as follows.

```
__tp_TEXT@%TP_SYMBOL {TEXT};
```

The text pointer value is the beginning of the TEXT segment, and is in "__tp_TEXT".
Describe as follows to set tp in the startup routine.

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

(2) Variable access preparations (Setting global pointer)

External variables or data defined in an application are allocated to the memory. The global pointer (gp) is a pointer prepared to implement referencing independent of location position (PID: Position Independent Data) when the variables or data allocated to the memory are referenced. The CX outputs a code for the section that is to be accessed in gp-relative mode.

Since the code is output on the assumption that gp is correctly set, gp must be correctly set in the startup routine. The global pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the global pointer is described as follows.

```
__gp_DATA@%GP_SYMBOL {DATA};
```

The gp symbol value can be defined the beginning of "data segment" of the DATA segment as shown above, or offset from a text symbol. A gp symbol can be specified not only by specifying the start address of a data segment (such as the DATA segment), but also by using an offset value from the text symbol as its address.

Using the second method, the gp symbol value is determined by adding value of tp and offset value from tp. In other words, a code that is independent of location can be generated. To copy a program code and data used by that code to the RAM area simultaneously and execute them, the value of gp can be acquired immediately if the start address of the copy destination is known. In this case, the symbol directive is described as follows.

```
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT {DATA};
```

The global pointer value is "__tp_TEXT to which the value of __gp_DATA is added", and the value to be added, i.e., offset value, is stored in "__gp_DATA". Therefore, describe as follows to set gp in the startup routine.

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

This sets the correct value of the global pointer to gp.

(3) Variable access preparations (Setting element pointer)

The element pointer (ep) is a pointer that is provided to realize faster access by allocating data (variables) that are globally declared within an application to RAM area in V850 core device.

Of the external variables or data defined in an application, those that are allocated to the following sections are accessed from the element pointer (ep) in relative mode.

- sedata/sebss attribute section
- sidata/sibss attribute section
- tidata/tibss attribute section
- tidata.byte/tibss.byte section
- tidata.word/tibss.word section

If these sections exist, the CX outputs a code to access these areas in ep-relative mode.

Since the code is output on the assumption that ep is correctly set, ep must be correctly set in the startup routine.

The element pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the element pointer is described as follows.

```
__ep_DATA@%EP_SYMBOL;
```

The element pointer value is the beginning of the SIDATA segment by default, and its value is in "__ep_DATA". Therefore, describe as follows to set ep in the startup routine.

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

Reference the absolute address of __ep_DATA and set that value to ep.

2.5.5 Preparing to use code size reduction function

This setting is necessary to reduce code size when the V850Ex core is used or when the prologue/epilogue runtime library is used (i.e. When execution speed priority optimization (-Ospeed option) is not specified or when "-Xpro_epi_runtime=on" is specified).

Since the CALLT instruction is used when the prologue/epilogue runtime library of functions is called by the V850Ex core, the value of CTBP necessary for the CALLT instruction must be set at the beginning of the function table of the prologue/epilogue runtime library of functions.

The prologue/epilogue runtime library is used in the following case.

- Compiler option "-Xpro_epi_runtime=on" is set.

If a compiler option except "-Ospeed" is specified for optimization, "-Xpro_epi_runtime=on" is automatically specified.

The start symbol for the function prologue/epilogue runtime library function table is as follows.

- ___PROLOG_TABLE

Describe the following code using this symbol.

```
mov     #___PROLOG_TABLE, r12
ldsr    r12, 20
```

Remark CTBP is system register 20. Set a value to it using the ldsr instruction.

2.5.6 Ending startup routine

The final process in the startup routine differs depending on whether or not a real-time OS is used.

(1) When not using real-time OS

When the processing necessary for the startup routine has been completed, execute an instruction that branches to the main function.
Describe the following code to branch to the main function.

```
jarl    _main, lp
```

When the main function has been executed, execution returns to the 4 bytes subsequent to this branch instruction.
The following instruction can also be used if it is known that execution does not return.

```
jr      _main
```

```
mov     #_main, lp
jmp     [lp]
```

The entire 32-bit space can be accessed using the jmp instruction. When the "jarl_main, lp" instruction is used, execution returns after the main function is executed. It is recommended to take appropriate action to prevent deadlock from occurring when execution returns.

(2) When using real-time OS (RX850V4)

In an application using a real-time OS, execution branches to the initialization routine when the processing that must be performed by the startup routine has been completed.

```
        .extern __kernel_sit
        .extern __kernel_start
        mov     #__kernel_sit, r6
        jarl    __kernel_start, lp

__boot_error:
        jbr     __boot_error
```

2.6 Link Directives

This section explains link directives.
Link directive files can be generated automatically in CubeSuite.

Remark For information about how to automatically generate link directive files, see the "CubeSuite Build for CX Compiler" user's Manual.

2.6.1 Adding function section allocation

To perform function section allocation, divert the .text section setting portion and change the segment name and section name.

```
TEXT:           !LOAD ?RX {
                .pro_epi_runtime    = $PROGBITS ?AX .pro_epi_runtime;
                .text               = $PROGBITS ?AX .text;
};
```

Example Setting allocation for USRTEXT segment and usr.text section

```
USRTEXT:        !LOAD ?RX {
                usr.text            = $PROGBITS ?AX usr.text;
};
```

2.6.2 Adding section allocation for variables

To add allocation settings for a variable section, divert the specification part for a section with the same attributes and change the segment name and section name.
The section attributes specify the section type when the section is set to a variable in #pragma section.

Section Type	Section to Be Diverted
data	.data/.bss
sdata	.sdata/.sbss
sconst	.sconst
const	.const

Example Setting allocation for USRCONST segment and usr.const section

```
USRCONST:       !LOAD ?R {
                usr.const           = $PROGBITS ?A usr.const;
};
```

2.6.3 Distributing section allocation

The following three methods for distributing section allocation are available.

(1) Distribute by section name

In the C source or assembler source, specify separate names for the sections to be allocated.
By specifying individual input section names within the link directive, the section of each name will be allocated to its specified part.

Example

```
TEXT:   !LOAD ?RX {
        .text           = $PROGBITS ?AX .text;
                                            <- The .text section is allocated.
};

FUNC1:  !LOAD ?RX {
        funcsec1.text   = $PROGBITS ?AX funcsec1.text;
                                            <- The funcsec1.text section is allocated.
};
```

(2) Distribute by object module files

By specifying individual object names within the link directive, the section with the relevant attributes within each object will be allocated to the specified part.

Example

```
TEXT1:  !LOAD ?RX {
        .text1 = $PROGBITS ?AX .text {file1.obj file2.obj};
                        <- The .text sections in file1.obj and file2.obj are allocated.
};

TEXT2:  !LOAD ?RX {
        .text2 = $PROGBITS ?AX .text {file3.obj};
                        <- The .text section in file3.obj is allocated.
};
```

When specifying the name an object module file in a library (.lib file), specify the .lib file name including its path within parentheses.

Example

```
.text3 = $PROGBITS ?AX .text {strcmp.obj(libc.lib)};
```

(3) Distribute by section attributes

Specify allocation only by attributes without specifying the input section and input object. Because this setting has a lower priority level than the part where settings such as section name and object name are made, it can be used to specify allocation for all parts where section and object names are not already specified.

Example

```
TEXT4:  !LOAD ?RX {
        .text4 = $PROGBITS ?AX  {file1.obj file2.obj};
               <- The TEXT ATTRIBUTE sections in file1.obj and file2.obj are allocated.
};

TEXT5:  !LOAD ?RX {
        .text5 = $PROGBITS ?AX;
               <- The TEXT ATTRIBUTE sections in objects other than file1.obj and
                  file2.obj are allocated.
};
```

(4) Allocation specification priority level

There are priority levels depending on the presence or lack of input section and input object specifications. When allocating sections, the linker allocates starting with the highest priority specification.

The relationship between priority level and specifications is shown below. (A lower the priority level number represents a higher priority.)

Priority Level	Specified Names	Output
1	Input section name + object module file name	The specified input section is extracted from the specified object and is then output.
2	Input section name only	The specified input section is extracted from all objects and is then output.
3	Object module file name only	Sections having the same attribute as the output section to be created are extracted from the specified object and are then output.
4	No names specified	Sections having the same attribute as the output section to be created are extracted from all objects and are then ouput.

2.7 Reducing Code Size

This section explains reducing code size.

2.7.1 Reducing code size (C language)

This section explains reducing cord size by C language.

(1) Access to variables

Because 4 bytes are needed each for external variable access loading and storing, even in non-assignment cases it is possible to reduce code size by assigning the external variable into a temporary variable and using that temporary variable so as to change memory access to register access.
In the following example s is an external variable

Before change:	After change:

`if(x != 0) {`
`  if((s & 0x00F00F00) != MASK1) {`
`  return;`
`  }`
`  s >>= 12;`
`  s &= 0xFF;`
`} else {`
`  if((s & 0x00FF0000) != MASK2) {`
`  return;`
`  }`
`  s >>= 24;`
`}`
	`unsigned int  tmp = s;`
	
	`if(x != 0) {`
	`  if((tmp & 0x00F00F00) != MASK1) {`
	`  return;`
	`  }`
	`  tmp >>= 12;`
	`  tmp &= 0xFF;`
	`} else {`
	`  if((tmp & 0x00FF0000) != MASK2) {`
	`  return;`
	`  }`
	`  tmp >>= 24;`
	`}`
	`s = tmp;`

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(2) Number of loops in loop processing

As in the following example, expanding a function may make its size smaller if the number of times to execute is few and body of each loop is small.
In this case, the execution speed also increases.

Before change:	After change:

for(i = 0; i < 4; i++) {
  array[i] = 0;
}
	long *p;
	p = array;
	*p = 0;
	*(p + 1) = 0;
	*(p + 2) = 0;
	*(p + 3) = 0;

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(3) auto variable initialization

When an auto variable is used within a function without being initialized, because that variable is not allocated to a register and remains in memory, the code size may increase.

In the following example if neither switch case applies then variable a is referenced in the return statement without being initialized.

Even if in actuality it will certainly apply to one of the cases it may not to be initialized because when the C compiler allocates to register it is not understood when the program is analyzed.

In a case such as this, it cannot be allocated with the CX register allocation.

By adding initialization it becomes able to be allocated to a register and the code size is reduced.

Before change:

```
int func(int x) {
        int a;

        switch(x) {
                case  0:
                        a = VAL0;
                        break;
                case  1:
                        a = VAL1;
        }
        return(a);
}
```

After change:

```
int func(int x) {
        int a = 0;

        switch(x) {
                case  0:
                        a = VAL0;
                        break;
                case  1:
                        a = VAL1;
        }
        return(a);
}
```

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(4) switch statements

With respect to switch statements, if there are four or more case labels and the difference between each variable's low limit and high limit is up to 3 times the number of cases, the CX generates code in table branch format.
In such an instance, if the number of cases is approximately 16 or less (this number varies depending on factors such as the switch expression format and the label value distribution), changing them to equivalent if-else statements and putting comparison and branch instructions in line will cause the code size to decrease.
In cases such as when the switch expression is an external variable reference or is a complex expression, it is necessary to once substitute the value to a temporary variable and make the if expression refer to the temporary variable.
In the following example x is an auto variable.

Before change:	After change:
switch(x) {	if(x == VAL0)
case VAL0:	return(RETVAL0);
return(RETVAL0);	else if(x == VAL1)
case VAL1:	return(RETVAL1);
return(RETVAL1);	else if(x == VAL2)
case VAL2:	return(RETVAL2);
return(RETVAL2);	else if(x == VAL3)
case VAL3:	return(RETVAL3);
return(RETVAL3);	else if(x == VAL4)
case VAL4:	return(RETVAL4);
return(RETVAL4);	else if(x == VAL5)
case VAL5:	return(RETVAL5);
return(RETVAL5);	
}	

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

4. With the CX it is possible to specify the switch statement development code with the -Xswitch option.

- -Xswitch=ifelse
  Outputs the code in the same format as the if-else statement along a string of case statements.
- -Xswitch=binary
  Outputs the code in the binary search format.
- -Xswitch=table
  Outputs the code in a table jump format.

(5) if statements

When executing the same processing to multiple cases with an if-else combination, if using a separate set of conditions would make the "multiple cases" combine into one case, then combine them.
This will delete redundant parts.
In the example below, if the conditions "the initial value of x is 0 and the values of s as well as t are either 0 or 1" are set, the code can be changed as follows.

Before change:

```
if(!s) {
        if(t) {
                x = 1;
        }
} else {
        if(!t) {
                x = 1;
        }
}
if(x) {
        if((++u) >= v) {
                u = 0;
        } else {
                x = 0;
        }
}
```

After change:

```
if((s^t)) {
        if((++u) >= v) {
                u = 0;
                x = 1;
        }
}
```

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

If an assigned value is referenced immediately following its assignment statement, the part referred to is substituted by the assignment statement and combined into one.
This makes possible deletion of excess register transferring and reduction in code size.
In most cases, however, redundant register transferring is deleted by the C compiler's optimization, so the code size would not change.

Before change:	After change:

--s;
if(s == 0) {
  :
}
	if(--s == 0)) {
	  :
	}

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(6) if-else statements

As in the following example, if each branch destination of an if-else statement includes only statements that assign differing values to the same variable, it is possible to reduce the code size by moving one of the branch destinations ahead of the if statement, because the else block will be erased and the jump instruction from the if the block to after the else block is eliminated.

Before change:	After change:

if(x == 10) {
  s = 1;
} else {
  s = 0;
}
	s = 0;
	if(x == 10) {
	  s = 1;
	}

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

As in the following example, if the branch destinations of if-else statements contain only return statements and those return values are the results of the branch conditions themselves, change it to return the branch condition expression and delete the if-else statement.

Before change:	After change:

if(s1 == s2) {
  return(1);
}
return(0);
	return(s1 == s2);

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

If after each respective branch a function is called using differing arguments for the same function, move the function call to after the branches converge if possible.
To do this, assign the differing arguments of the original function calls to temporary variables and use these temporary variables as arguments when calling the function.

Before change:	After change:

`if(s) {`
`  :`
`  func(0, 1, 2);`
`} else {`
`  :`
`  func(0, 1, 3);`
`}`
	`int tmp;`
	`if(s) {`
	`  :`
	`  tmp = 2;`
	`} else {`
	`  :`
	`  tmp = 3;`
	`}`
	`func(0, 1, tmp);`

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

In the case that after respective branches an identical assignment statement or function call exists, move it to before the branch if possible.
If that statement's evaluation result is referenced, assign it once to a temporary variable and reference the temporary variable.
The following example is a case of a function call.

Before change:	After change:

`if(x >= 0) {`
`  if(x > func(0, 1, 2)) {`
`  :`
`  }`
`} else {`
`  if(x < -func(0, 1, 2)) {`
`  :`
`  }`
`}`
	`long  tmp;`
	`tmp = func(0, 1, 2);`
	`if(x >= 0) {`
	`  if(x > tmp) {`
	`  :`
	`  }`
	`} else {`
	`  if(x < -tmp) {`
	`  :`
	`  }`
	`}`

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

In the case that after respective branches an identical assignment statement or function call exists, if it cannot be moved to before the branch but can be moved to after the merge, move it to after the merge.
The following example is an assignment statement case.

Before change:	After change:

if(tmp & MASK) {
  :
  j++;
} else {
  :
  j++;
}
	if(tmp & MASK) {
	  :
	} else {
	  :
	}
	j++;

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(7) switch/if-else statements

As in the following example, in the case where differing values are assigned to the same external variable at the respective branch destinations of a switch statement or an if-else statement, it is possible to reduce code size by assigning the values to a temporary variable at each branch and then reassigning the temporary variable value back to the original external variable after the branches merge.

This is because, assigning to an external variable requires a memory store instruction (4 bytes) because external variables are rarely allocated to registers, while in most cases assigning to a temporary variable uses a register transfer (2 bytes).

In the following example s is an external variable.

Before change:	After change:
switch(x) {	int tmp;
case 0:	if(x == 0) {
s = 0;	tmp = 0;
break;	} else if (x == 1) {
case 1:	tmp = 0x5555;
s = 0x5555;	} else if(x == 2) {
break;	tmp = 0xAAAA;
case 2:	} else if(x == 3) {
s = 0xAAAA;	tmp = 0xFFFF;
break;	} else {
case 3:	goto label;
s = 0xFFFF;	}
}	s = tmp;
	label:

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.

(8) Functions with no return values

Define functions with no return values as "void."

2.7.2 Reducing variable area with variable definition method

This section explains reducing variable area with the variable definition method.

(1) Variable format

Because by ANSI-C specifications variables in short integer ((unsigned) short and (unsigned) char) formats are expanded to int format or unsigned int format during operation, many format change instructions are generated with respect to programs that use these variables (particularly in cases where these variables are allocated to registers).

Since making them (unsigned) int format makes this format change unnecessary, the code size is reduced.

Particularly with respect to stack intervals that are relatively easy to allocate to registers, it is recommended to use (unsigned) int format as much as possible.

Before change:	After change:

`unsigned char i;`
 	
`for(i = 0; i < 4; i++) {`
`  array[2 + i] = *(p + i);`
`}`
	`int i;`
	`for(i = 0; i < 4; i++) {`
	`  array[2 + i] = *(p + i);`
	`}`

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.
  In such a case, the code size will increase by the save/restore code amount (8 bytes).

(2) Allocating and referencing automatic variables

As in the following example, if there is a time interval between when a value is assigned to a stack variable and when that value is actually referenced, during that interval a register is occupied and the chance for other variables to be allocated to registers decreases.

In such a case, changing the value assignment to immediately before it is actually referenced increases the chance for other variables to be allocated to registers increases, decreases memory access, and decreases the code size.

Before change:

```
int i = 0, j = 0, k = 0, m = 0;


/*There is a function call in this
interval*/
/*These variables are not used*/


while((k & 0xFF) != 0xFF) {
    k = s1;
    if(k & MASK) {
        if(m != 1) {
            s2 += 2;
            m = 1;
            array[15+i+j] = 0xFF;
            j++;
        }
    }
}
    :
```

After change:

```
int i, j, k, m;
  :
i = 0;
j = 0;
k = 0;
m = 0;
while((k & 0xFF) != 0xFF) {
    k = s1;
    if(k & MASK) {
        if(m != 1) {
            s2 += 2;
            m = 1;
            array[15+i+j] = 0xFF;
            j++;
        }
    }
}
      :
```

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.
  In such a case, the code size will increase by the save/restore code amount (8 bytes).

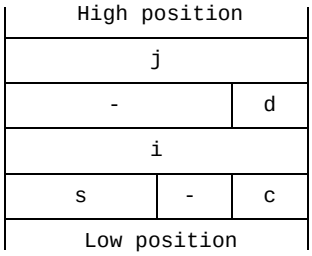
(3) Variable types and order of definition

It is best to perform definitions in groups beginning with long data length values.

With the V850 microcontroller, word data in formats such as int format must be aligned to word boundaries, and halfword data in formats such as short format must be aligned to halfword boundaries.

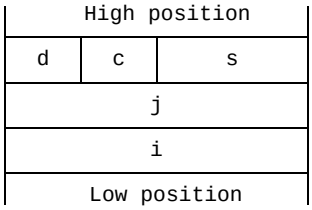
Due to this, source such as the following causes padding areas to be generated for alignment.

```
char  c = 'c';
short s = 0;
int   i = 1;
char  d = 'b';
int   j = 2;
```



In order to avoid the generation of such padding areas, define definitions of variables and structure members grouped by format beginning with longer data lengths.

```
int   i = 1;
int   j = 2;
short s = 0;
char  c = 'a';
char  d = 'b';
```



2.8 Accelerating Processing

This section explains accelerating processing.

2.8.1 Accelerating processing with description method

This section explains accelerate processing with the description method

(1) Loop processing pointer

A variable that controls a loop as in the example below is called an induction variable.
"Deleting the induction variable" refers to optimization that deletes the induction variable by using a different variable to control the loop.
The CX includes this optimization, but because applicable conditions are limited, not all cases are able to be optimized.
By modifying the program in the following manner, this optimization can be performed "manually".
In the lines below, induction variable i is deleted through the use of temporary variable (pointer) p.

Before change:	After change:
`int i;`	`const unsigned short  *p;`
`for(i = 0; *(table + i) != NULL; ++i) {`	`for(p = table; *p != NULL; ++p) {`
`    if((*(table + i) & 0xFF) == x) {`	`    if((*p & 0xFF) == x) {`
`        return(*(table + i) & 0xFF00);`	`        return(*p & 0xFF00);`
`    }`	`    }`
`}`	`}`

Remarks 1. The amount of reduction is specific to this example, and will vary case by case.

2. As a result of changing the source, output instructions may be reduced and execution speed may be increased.

3. Pay attention to the following points when changing the source.

- Changing the source causes the state of register usage to change. It is therefore possible that in unintended places register transfers that had up until that point remained without being optimized may be erased or, alternatively, that optimization may become ineffective causing redundant register transfers to remain.
- By adding temporary variables, a new register for register variables may come to be used, resulting in code for saving and restoring that register being added to the function entrance and exit.
  In such a case, the code size will increase by the save/restore code amount (8 bytes).

(2) Auto variable declaration

Keep the number of auto variables to within ten; of preferably to six or seven.
Auto variables are assigned to registers.
The CX allows a total of 20 registers, 10 work registers and 10 register variable registers, to be used for variables (in the 32-bit register mode).
It is recommended to use many auto variables if processing in one function takes time.
If the processing does not take much time, use only the 10 work registers whenever possible.
The register variable registers require overhead when they are saved or restored.
The C compiler automatically judges whether or not to use register variables.
Therefore, use six to seven registers for auto variables and leave three or four to be able to be used for work by the C compiler.

(3) Function arguments

Four argument registers, r6 to r9, are available.

If the number of arguments is five or more, the stack is used for the fifth and subsequent arguments.

Therefore, keep the number of arguments to within four whenever possible.

If five or more arguments must be used, pass the arguments using the pointer of a structure.

2.9 Compiler and Assembler Mutual References

This section explains compiler and assembler mutual references.

2.9.1 Mutually referencing variables

This section explains mutually referencing variables.

(1) Reference variable defined in C language

Define extern when referencing an external variable defined in a C language program from an assembly language routine.

Prefix "_" (an underscore) to a variable defined in an assembly language routine.

Example C source

```
extern void subf(void);
char    c = 0;
int     i = 0;

void main(void) {
        subf ();
}
```

Example Assembler source

```
        .public _subf
        .extern _c, 1
        .extern _i, 4
        .cseg   text
        .align  4
_subf:
        mov     4, r10
        st.b    r10, $_c
        mov     7, r10
        st.w    r10, $_i
        jmp     [lp]
```

(2) Reference variable defined in assembly language

Define extern when referencing in a C language routine an external variable defined in an assembly language program.

Prefix "_" (an underscore) to a variable defined in an assembly language routine.

Example C source

```
extern char c;
extern int  i;

void subf (void) {
        c = 'A';
        i = 4;
}
```

Example Assembler source

```
        .public _i, 4
        .public _c, 1
        .dseg   sbss
        .align  4
_i:
        .ds     (4)
_c:
        .ds     (1)
```

2.9.2 Mutually referencing functions

This section explains mutually referencing functions.

(1) Reference function defined in C language

Note the following points when calling a function described in C language from an assembly language routine.

- Stack frame
  Code is generated on the assumption that the stack pointer (sp) always indicates the lowest address of the stack frame. Therefore, set sp so that it indicates the higher address of an unused area of the stack area when execution branches from an assembler function to a C function.
- Work register
  Values of the register variable registers before and after a C function is called are retained, but the values of the work registers are not. Therefore, do not leave a value that must be retained assigned to a work register.
- Return address to return to assembler function
  Code is generated on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to a C function, therefore, the return address of the function must be stored in lp.

(2) Reference function defined in assembly language

Note the following points when calling an assembly language routine from a function described in C language.

- Identifier
  Prefix "_" to the name.
- Stack frame
  Code is output based on the assumption that the stack pointer (sp) always indicates the lowest address of the stack frame. Therefore, the address area lower than the address indicated by sp can be freely used in the assembler function after branching from a C source to an assembler function. Conversely, if the contents of the higher address area are changed, the area used by a C function may be lost and the subsequent operation cannot be guaranteed. To avoid this, change sp at the beginning of the assembler function before using the stack.
  At this time, however, make sure that the value of sp is retained before and after calling.
- Register variable register
  When using a register variable register in an assembler function, make sure that the register value is retained before and after the assembler function is called. In other words, save the value of the register variable register before calling the assembler function, and restore the value after calling.
- Return address to C language function
  Code is generated on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to an assembler function, the return address of the function is stored in lp. Execute the jmp [lp] instruction to return to a C function.

CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS

This chapter explains language specifications supported by the CX.

3.1 Basic Language Specifications

The CX supports the language specifications stipulated by the ANSI standards. These specifications include items that are stipulated as processing definitions. This chapter explains the language specifications of the items dependent on the processing system of the V850 microcontrollers.

The differences between when options strictly conforming to the ANSI standards are used and when those options are not used are also explained.

See "3.2 Extended Language Specifications" for extended language specifications explicitly added by the CX.

3.1.1 Unspecified behavior

This section describes behavior that is not specified by the ANSI standard.

(1) Execution environment - initialization of static storage

Static data is output during compilation as a data section.

(2) Meanings of character displays - backspace (\b), horizontal tab (\t), vertical tab (\t)

This is dependent on the design of the display device.

(3) Types - floating point

IConforms to IEEE754[Note].

Note IEEE: Institute of Electrical and Electronics Engineers
IEEE754 is a system for handling floating-point calculations, providing a uniform standard for data formats, numerical ranges, and the like handled.

(4) Expressions - evaluation order

In general, expressions are evaluated from left to right. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.

(5) Function calls - parameter evaluation order

In general, function arguments are evaluated from first to last. The behavior when optimization has been applied, however, is undefined. Options or other settings could change the order of evaluation, so please do not code expressions with side effects.

(6) Structure and union specifiers

These are adjusted so that they do no span bit field type alignment boundaries. If packing has been conducting using options or a #pragma, then bit fields are packed, and not adjusted to alignment boundaries.

(7) Function definitions - storage of formal parameters

These are assigned to the stack and register. For the details, see "3.3.1 Calling between C functions".

(8) # operator

These are evaluated left to right.

3.1.2 Undefined behavior

This section describes behavior that is not defined by the ANSI standard.

(1) Character set

A message is output if a source file contains a character not specified by the character set.

(2) Lexical elements

A message is output if there is a single or double quotation mark (‘/") in the last category (a delimiter or a single non-whitespace character that does not lexically match another preprocessing lexical type).

(3) Identifiers

Since all identifier characters have meaning, there are no meaningless characters.

(4) Identifier binding

A message is output if both internal and external binding was performed on the same identifier within a translation unit.

(5) Compatible type and composite type

All declarations referencing the same object or function must be compatible. Otherwise, a message will be output.

(6) Character constants

Specific non-graphical characters can be expressed by means of extended notation, consisting of a backslash (\) followed by a lower-case letter. The following are available: \a, \b, \f, \n, \r, \t, and \v. There is no other extended notation; other letters following a backslash (\) become that letter.

(7) String literals - concatenation

When a simple string literal is adjacent to a wide string literal token, simple string concatenation is performed.

(8) String literals - modification

Users modify string literals at their own risk. Although the string will be changed if it is allocated to RAM, it will not be changed if it is allocated to ROM.

(9) Header names

If the following characters appear in strings between the delimiter characters < and >, or between two double quotation marks ("), then they are treated as part of the file name: characters, comma (,), double quote ("), two slashes (//), or slash-asterisk (/*). The backslash (\) is treated as a folder separator.

(10) Floating point type and integral type

If a floating-point type is converted into an integral type, and the integer portion cannot be expressed as an integral type, then the value is truncated until it can.

(11) lvalues and function specifiers

A message is output if an incomplete type becomes an lvalue.

(12) Function calls - number of arguments

If there are too few arguments, then the values of the formal parameters will be undefined. If there are too many arguments, then the excess arguments will be ignored when the function is executed, and will have no effect.
A message will be output if there is a function declaration before the function call.

(13) Function calls - types of extended parameters

If a function is defined without a function prototype, and the types of the extended arguments do not match the types of the extended formal parameters, then the values of the formal parameters will be undefined.

(14) Function calls - incompatible types

If a function is defined with a type that is not compatible with the type specified by the expression indicating the called function, then the return value of the function will be invalid.

(15) Function calls - incompatible types

If a function is defined in a form that includes a function prototype, and the type of an extended argument is not compatible with that of a formal parameter, or if the function prototype ends with an ellipsis, then it will be interpreted as the type of the formal parameter.

(16) Addresses and indirection operators

If an incorrect value is assigned to a pointer, then the behavior of the unary * operator will either obtain an undefined value or result in an illegal access, depending on the hardware design and the contents of the incorrect value.

(17) Cast operator - function pointer casts

If a typecast pointer is used to call a function with other than the original type, then it is possible to call the function. If the parameters or return value are not compatible, then it will be invalid.

(18) Cast operator - integral type casts

If a pointer is cast into an integral type, and the amount of storage is too small, then the storage of the cast type will be truncated.

(19) Multiplicative operators

A message will be output if a divide by zero is detected during compilation.
During execution, a divide by zero will raise an exception. If an error-handling routine has been coded, it will be handled by this routine.

(20) Additive operators - non-array pointers

If addition or subtraction is performed on a pointer that does other than indicate elements in an array object, the behavior will be as if the pointer indicates an array element.

(21) Additive operators - subtracting a pointer from another array

If subtraction is performed using two pointers that do not indicate elements in the same array object, the behavior will be as if the pointers indicate array elements.

(22) Bitwise shift operators

If the value of the right operand is negative, or greater than the bit width of the extended left operand, then the result will be the shifted value of the right operand, masked by the bit width of the left operand.

(23) Function operators - pointers

If the objects referring to by the pointers being compared are not members of the same structure or union object, then the relationship operation will be performed for pointers referring to the same object.

(24) Simple assignment

If a value stored in an object is accessed via another object that overlaps that object's storage area in some way, then the overlapping portion must match exactly. Furthermore, the types of the two objects must have modified or non-modified versions with compatible types. Assignment to non-matching overlapping storage could cause the value of the assignment source to become corrupted.

(25) Structure and union specifiers

If the member declaration list does not include named members, then a message will be output warning that the list has no effect. Note, however, that the same message will be output accompanied by an error if the -Xansi option is specified.

(26) Type modifiers - const

A message will be output if an attempt is made to modify an object defined with a const modifier, using an lvalue that is the non-const modified version. Casting is also prohibited.

(27) Type modifiers - volatile

A message will be output if an attempt is made to modify an object defined with a volatile modifier, using an lvalue that is the non-volatile modified version.

(28) return statements

A message will be output if a return statement without an expression is executed, and the caller uses the return value of the function, and there is a declaration. If there is no declaration, then the return value of the function will be undefined.

(29) Function definitions

If a function taking a variable number of arguments is defined without a parameter type list that ends with an ellipsis, then the values of the formal parameters will be undefined.

(30) Conditional inclusion

If a replacement operation generates a "defined" token, or if the usage of the "defined" unary operator before macro replacement does not match one of the two formats specified in the constraints, then it will be handled as an ordinary "defined".

(31) Macro replacement - arguments not containing preprocessing tokens

A message is output if the arguments (before argument replacement) do not contain preprocessing tokens.

(32) Macro replacement - arguments with preprocessing directives

A message is output if an argument list contains a preprocessor token stream that would function as a processing directive in another circumstance.

(33) # operator

A message is output if the results of replacement are not a correct simple string literal.

(34) ## operator

A message is output if the results of replacement are not a correct simple string literal.

3.1.3 Processing system dependent items

This section explains items dependent on processing system in the ANSI standards.

(1) Data types and sizes

The byte order in a word (4 bytes) is "from least significant to most significant byte" Signed integers are expressed by 2's complements. The sign is added to the most significant bit (0 for positive or 0, and 1 for negative).

- The number of bits of 1 byte is 8.
- The number of bytes, byte order, and encoding in an object module files are stipulated below.

Table 3-1. Data Types and Sizes

Data Types	Sizes
char	1 byte
short	2 bytes
int, long, float	4 bytes
double, long double, long long	8 bytes
pointer	Same as unsigned int

(2) Translation stages

The ANSI standards specify eight translation stages (known as "phases") of priorities among syntax rules for translation. The arrangement of "non-empty white space characters excluding line feed characters" which is defined as processing system dependent in phase 3 "Decomposition of source file into preprocessing tokens and white space characters" is maintained as it is without being replaced by single white space character.

(3) Diagnostic messages

When syntax rule violation or restriction violation occurs on a translation unit, the compiler outputs as error message containing source file name and (when it can be determined) the number of line containing the error. These error messages are classified: "Warning", "Abort error", "Fatal error" and "other" messages. For output formats of messages, see the "CubeSuite Message" user's Manual.

(4) Free standing environment

(a) The name and type of a function that is called on starting program processing are not stipulted in a free-standing environment[Note]. Therefore, it is dependent on the user-own coding and target system.

Note Environment in which a C source program is executed without using the functions of the operating system.
In the ANSI Standard two environments are stipulated for execution environment: a free-standing environment and a host environment. The CX does not supply a host environment at present.

(b) The effect of terminating a program in a free-standing environment is not stipulated. Therefore, it is dependent on the user-own coding and target system.

(5) Program execution

The configuration of the interactive unit is not stipulated.
Therefore, it is dependent on the user-own coding and target system.

(6) Character set

The values of elements of the execution environment character set are ASCII codes.

(7) Multi-byte characters

Supported multi-byte characters are ECU, SJIS and UTF-8.
Japanese description in comments and character strings is supported.

(8) Significance of character display

The values of expanded notation are stipulated as follows.

Table 3-2. Expanded Notation and Meaning

Expanded Notation	Value (ASCII)	Meaning
\a	07	Alert (Warning tone)
\b	08	Backspace
\f	0C	Form feed (New Page)
\n	0A	New line (Line feed)
\r	0D	Carriage return (Restore)
\t	09	Horizontal tab
\v	0B	Vertical tab

(9) Translation Limit

A maximum of 2,000 files can be linked. Specifying more than 2,000 files for linking will cause an E0511138 error. There are no other limits on translation. The maximum translatable value depends on the memory of the host machine on which the program is running.

(10) Quantitative limit

(a) The limit values of the general integer types (limits.h file)

The limits.h file specifies the limit values of the values that can be expressed as general integer types (char type, signed/unsigned integer type, and enumerate type).
Because multi-byte characters are not supported, MB_LEN_MAX does not have a corresponding limit. Consequently, it is only defined with MB_LEN_MAX as 1.
If the -Xchar=unsigned option of the CX is specified, CHAR_MIN is 0, and CHAR_MAX takes the same value as UCHAR_MAX.
The limit values defined by the limits.h file are as follows.

Table 3-3. Limit Values of General Integer Type (limits.h File)

Name	Value	Meaning
CHAR_BIT	+8	The number of bits (= 1 byte) of the minimum object not in bit field
SCHAR_MIN	-128	Minimum value of signed char
SCHAR_MAX	+127	Maximum value of signed char
UCHAR_MAX	+255	Maximum value of unsigned char
CHAR_MIN	-128	Minimum value of char

Name	Value	Meaning
CHAR_MAX	+127	Maximum value of char
SHRT_MIN	-32768	Minimum value of short int
SHRT_MAX	+32767	Maximum value of short int
USHRT_MAX	+65535	Maximum value of unsigned short int
INT_MIN	-2147483648	Minimum value of int
INT_MAX	+2147483647	Maximum value of int
UINT_MAX	+4294967295	Maximum value of unsigned int
LONG_MIN	-2147483648	Minimum value of long int
LONG_MAX	+2147483647	Maximum value of long int
ULONG_MAX	+4294967295	Maximum value of unsigned long int
LLONG_MIN	-9223372036854775807	Minimum value of long long int
LLONG_MAX	+9223372036854775807	Maximum value of long long int
ULLONG_MAX	18446744073709551615	Minimum value of unsigned long long int

(b) The limit values of the floating-point type (float.h file)

The limit values related to characteristics of the floating-point type are defined in float.h file.
The limit values defined by the float.h file are as follows.

Table 3-4. Definition of Limit Values of Floating-point Type (float.h File)

Name	Value	Meaning
FLT_ROUNDS	+1	Rounding mode for floating-point addition. 1 for the V850 microcontrollers (rounding in the nearest direction).
FLT_RADIX	+2	Radix of exponent (b)
FLT_MANT_DIG	+24	Number of numerals (p) with FLT_RADIX of floating- point mantissa as base
DBL_MANT_DIG	+53	
LDBL_MANT_DIG	+53	
FLT_DIG	+6	Number of digits of a decimal number (q) that can round a decimal number of q digits to a floating-point number of p digits of the radix b and then restore the decimal number of q
DBL_DIG	+15	
LDBL_DIG	+15	
FLT_MIN_EXP	-125	Minimum negative integer (e_{min}) that is a normalized floating-point number when FLT_RADIX is raised to the power of the value of FLT_RADIX minus 1.
DBL_MIN_EXP	-1021	
LDBL_MIN_EXP	-1021	
FLT_MIN_10_EXP	-37	Minimum negative integer$\log_{10}b^{e_{min-1}}$ that falls in the range of a normalized floating-point number when 10 is raised to the power of its value.
DBL_MIN_10_EXP	-307	
LDBL_MIN_10_EXP	-307	
FLT_MAX_EXP	+128	Maximum integer (e_{max}) that is a finite floating-point number that can be expressed when FLT_RADIX is raised to the power of its value minus 1.
DBL_MAX_EXP	+1024	
LDBL_MAX_EXP	+1024	

Name	Value	Meaning
FLT_MAX_10_EXP	+38	Maximum value of finite floating-point numbers that can be expressed $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP	+308	
LDBL_MAX_10_EXP	+308	
FLT_MAX	3.40282347E + 38F	Maximum value of finite floating-point numbers that can be expressed $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX	1.7976931348623158E+308	
LDBL_MAX	1.7976931348623158E+308	
FLT_EPSILON	1.19209290E - 07F	Difference between 1.0 that can be expressed by specified floating-point number type and the lowest value which is greater than 1. $b^{1 - p}$
DBL_EPSILON	2.2204460492503131E-016	
LDBL_EPSILON	2.2204460492503131E-016	
FLT_MIN	1.17549435E - 38F	Minimum value of normalized positive floating-point number $b^{e_{min - 1}}$
DBL_MIN	2.2250738585072014E-308	
LDBL_MIN	2.2250738585072014E-308	

(11) Identifier

All identifiers are considered to have meaning. There are no restrictions on identifier length.
Uppercase and lowercase characters are distinguished.

(12) char type

A char type with no type specifier (signed, unsigned) specified is treated as a signed integer as the default assumption.
However, a simple char type can be treated as an unsigned integer by specifying the -Xchar=unsigned option of the CX.
The types of those that are not included in the character set of the source program required by the ANSI standards (escape sequence) is converted for storage, in the same manner as when types other than char type are substituted for a char type.

```
char    c = '\777';     /*Value of c is -1*/
```

(13) Floating-point constants

The floating-point constants conform to IEEE754[Note].

Note IEEE: Institute of Electrical and Electronics Engineers
IEEE754 is a system for handling floating-point calculations, providing a uniform standard for data formats, numerical ranges, and the like handled.

(14) Character constants

(a) Both the character set of the source program and the character set in the execution environment are basically ASCII codes, and correspond to members having the same value.
However, for the character set of the source program, character codes in Japanese can be used (see "(8) Significance of character display").

(b) The last character of the value of an integer character constant including two or more characters is valid.

(c) A character that cannot be expressed by the basic execution environment character set or escape sequence is expressed as follows.

<1> An octal or hexadecimal escape sequence takes the value indicated by the octal or hexadecimal notation

\777	511

<2> The simple escape sequence is expressed as follows.

\'	'
\"	"
\?	?
\\	\

<3> Values of \a, \b, \f, \n, \r, \t, \v are same as the values explained in "(8) Significance of character display**".**

(d) Character constants of multi byte characters are not supported.

(15) Character string

A character string can be described in Japanese.
The default character code is Shift JIS.
A character code in input source file can be selected by using the -Xcharacter_set option of the CX.

[Option specification]

-Xcharacter_set=[none | euc_jp | sjis | utf8]

(16) Header file name

The method to reflect the string in the two formats (< > and " ") of a header file name on the header file or an external source file name is stipulated in "(33) Loading header file".

(17) Comment

A comment can be described in Japanese. The character code is the same as the character string in "(15) Character string".

(18) Signed constants and unsigned constants

If the value of a general integer type is converted into a signed integer of a smaller size, the higher bits are truncated and a bit string image is copied.
If an unsigned integer is converted into the corresponding signed integer, the internal representation is not changed.

(19) Floating-points and general integers

If the value of a general integer type is converted into the value of a floating-point type, and if the value to be converted is within a range that can be expressed but not accurately, the result is rounded to the closest expressible value.

When the result is just a middle value, it can be rounded to the even number (with the least significant bit of the mantissa being 0).

(20) double type and float type

In the CX, a double type is treated as 64-bit (double-precision) data and a float type is treated as 32-bit (single-precision) data.

(21) Signed type in operator in bit units

The characteristics of the shift operator conform to the stipulation in "(27) Shift operator in bit units".

The other operators in bit units for signed type are calculated as unsigned values (as in the bit image).

(22) Members of structures and unions

If the value of a member of a union is stored in a different member, it is stored according to an alignment condition. Therefore, the members of that union are accessed according to the alignment condition (see "(6) Structure type" and "(7) Union type").

In the case of a union that includes a structure sharing the arrangement of the common first members as a member, the internal representation is the same, and the result is the same even if the first member common to any structure is referred.

(23) sizeof operator

The value resulting from the "sizeof" operator conforms to the stipulation related to the bytes in an object in "(1) Data types and sizes".

For the number of bytes in a structure and union, it is byte including padding area.

(24) Cast operator

When a pointer is converted into a general integer type, the required size of the variable is the same as the size of the unsigned long type. The bit string is saved as is as the conversion result.

Any integer can be converted by a pointer. However, the result of converting an integer smaller than an int type is expanded according to the type.

(25) Division/remainder operator

The result of the division operator ("/") when the operands are negative and do not divide perfectly with integer division, is as follows: If either the divisor or the dividend is negative, the result is the smallest integer greater than the algebraic quotient.

If both the divisor and the dividend are negative, the result is the largest integer less than the algebraic quotient.

If the operand is negative, the result of the "%" operator takes the sign of the first operand in the expression.

(26) Addition and subtraction operators

If two pointers indicating the elements of the same array are subtracted, the type of the result is unsigned long type.

(27) Shift operator in bit units

If E1 of "E1 >> E2" is of signed type and takes a negative value, an arithmetic shift is executed.

(28) Storage area class specifier

Optimize for the fastest possible access, regardless of whether there is a storage-class area specifier "register" declaration.

(29) Structure and union specifier

(a) A simple int type bit field without signed or unsigned appended is treated as a signed field, and the most significant bit is treated as the sign bit. However, the simple int type bit field can be treated as an unsigned field by specifying the -Xbitfield option (Specifying sign of simple int type bit field) of the CX.

(b) To retain a bit field, a storage area unit to which any address with sufficient size can be assigned can be allocated. If there is insufficient area, however, the bit field that does not match is packed into to the next unit according to the alignment condition of the type of the field.

(c) The allocation sequence of the bit field in unit is from lower to higher.

(d) Each member of the non-bit field of one structure or union is aligned at a boundary as follows:

char, unsigned char type, and its array	Byte boundary
short, unsigned short type, and its array	2-byte boundary
Others (including pointer)	4-byte boundary

(30) Enumerate type specifier

The type of an enumeration specifier is signed int.
However, when the -Xenum_type=auto option is specified, each enumerated type is treated as the smallest integer type capable of expressing all the enumerators in that type.

(31) Type qualifier

The configuration of access to data having a type qualified to be "volatile" is dependent upon the address (I/O port, etc.) to which the data is mapped.

(32) Condition embedding

(a) The value for the constant specified for condition embedding and the value of the character constant appearing in the other expressions are equal.

(b) The character constant of a single character must not have a negative value.

(33) Loading header file

(a) A preprocessing directive in the form of "#include <character string>"

A preprocessing directive in the form of "#include <character string>" searches for a header file from the folder specified by the -I option if "character string" does not begin with "\"[Note], and then searches standard include file folder (..\inc folder with a relative path from the bin folder where the cx is placed).
If a header file uniformly identified is searched with a character string specified between delimiters "<" and ">", the whole contents of the header file are replaced.

Note "/" are regarded as the delimiters of a folder.

```
#include    <header.h>
```

The search order is as follows.

- Folder specified by -I
- Standard include file folder

(b) A preprocessing directive in the form of "#include "character string""

A preprocessing directive in the form of "#include "character string"" searches for a header file from the folder where the source file exists, then searches specified folder (-I option) and then searches standard include file folder (..\inc folder with a relative path from the bin folder where the cx is placed).
If a header file uniformly identified is searched with a character string specified between delimiters " " " and " " ", the whole contents of the header file are replaced.

Example

```
#include    "header.h"
```

The search order is as follows.

- Folder where source file exists
- Folder specified by -I
- Standard include file folder

(c) The format of "#include preprocessing character phrase string"

The format of "#include preprocessing character phrase string" is treated as the preprocessing character phrase of single header file only if the preprocessing character phrase string is a macro that is replaced to the form of <character string> or "character string".

(d) A preprocessing directive in the form of "#include <character string>"

Between a string delimited (finally) and a header file name, the length of the alphabetic characters in the strings is identified,

And the file name length valid in the compiler operating environment is valid.

The folder that searches a file conforms to the above stipulation.

(34) #pragma directive

The CX can specify the following #pragma directives.

(a) Describing assembler instruction

```
#pragma asm
        assembler instruction
#pragma endasm
```

Assembler directives can be described in a C source program.
For the details of description, see "(5) Describing assembler instruction".

(b) Inline expansion specification

```
#pragma inline  function-name[, function-name ...]
```

A function that is expanded inline can be specified.
For the details of expansion specification, see "(9) Inline expansion".

(c) Data or program memory allocation

```
#pragma section section-type ["section-name"]
#pragma text    ["section-name"] [function-name[, function-name]...]
```

<1> section

Allocates variables to an arbitrary section.
For details about the allocation method, see "(2) Allocation of data to section".

<2> text

A function to be allocated in a text section with an arbitrary name can be specified.
For details about the allocation specification, see "(3) Allocating functions to sections".

(d) Peripheral I/O register name validation specification

```
#pragma ioreg
```

The peripheral I/O registers of a device are accessed by using peripheral function register names. When programming using peripheral I/O registers names as it is, #pragma directives are needed to be specified.

(e) Interrupt/exception handler specification

```
#pragma interrupt interrupt-request-name function-name [allocation-method]  [Option
[Option]...]
```

Interrupt/Exception handlers are described in C language.
For the details of description, see "(c) Describing interrupt/exception handler".

(f) Interrupt disable function specification

```
#pragma block_interrupt function-name
```

Interrupts are disabled for the entire function.

(g) Task specification

```
#pragma rtos_task   function-name
```

The task of operating on the realtime OS is described by C language.
For the details of description, see "(a) Description of task".

(h) Structure type packing specification

```
#pragma pack([1248])
```

Specifies the packing of a structure type. The packing value, which is an alignment value of the member, is specified as the numeric value. A value of 1, 2, 4, or 8 can be specified. When the numeric value is not specified, it is by default (8)[Note].

Note Alignment values "4" and "8" are treated as the same in this Version.

(i) Smart correction specification

```
#pragma smart_correct function-name function-name
```

Specifies the function of smart correction.
For the details of description, see "(13) Smart correction function".

(35) Predefined macro names

All the following macro names are supported.
Macros not ending with "_ _" are supplied for the sake of former C language specifications (K&R specifications).
To perform processing strictly conforming to the ANSI standards, use macros with "_ _" before and after.

Table 3-5. List of Supported Macros

Macro name	Definition
__LINE__	Line number of source line at that point (decimal).
__FILE__	Name of assumed source file (character string constant).
__DATE__	Date of translating source file (character string constant in the form of "Mmm dd yyyy"). Here, the name of the month is the same as that created by the asctime function stipulated by ANSI standards (3 alphabetic characters with only the first character is capital letter) (The first character of dd is blank if its value is less than 10).
__TIME__	Translation time of source file (character string constant having format "hh:mm:ss" similar to the time created by the asctime function).
__STDC__	Decimal constant 1 (defined when the -Xansi option is specified)[Note]
__v850 __v850__	Decimal constant 1.
__v850e __v850e__	Decimal constant 1 (defined by the CX, if V850Ex is specified as a target device).
__v850e2 __v850e2__	Decimal constant 1 (defined by the CX, if V850E2 is specified as a target device).
__v850e2v3 __v850e2v3__	Decimal constant 1 (defined by the CX, if device with an instruction set of V850E2V3 is specified as a target device).
__K0R __K0R__	Decimal constant 1 (defined by the CX, if 78K0R is specified as a target device).
__CX __CX__	Decimal constant 1.

Macro name	Definition
__CHAR_SIGNED__	Decimal constant 1 (defined if signed is specified by the -Xchar option and when the -Xchar option is not specified).
__CHAR_UNSIGNED__	Decimal constant 1 (defined when unsigned is specified by the -Xchar option).
__DOUBLE_IS_64BITS__	Decimal constant 1.
__CPUmacro_ __CPUmacro__	Decimal constant 1 of a macro indicating the target CPU. A character string indicated by "product type specification" in the device file with "_ _" prefixed and "_" or "_ _"suffixed is defined.
Register mode macro	Decimal constant 1 of a macro indicating the target CPU. Macro defined with register mode is as follows. 32 register mode: __reg32__ 26 register mode: __reg26__ 22 register mode: __reg22__ Universal register mode: __reg_common__
__MULTI_CORE__	Decimal constant 1 (defined when specified by the -Xmulti option).
__MULTI_CMN__ __MULTI_PE*n*__	Decimal constant 1 (defined when core specified by the -Xmulti option (*n* is the numerical value.)).

Note For the processing to be performed when the -Xansi option is specified, see "3.1.5 ANSI option".

3.1.4 C99 language function

This section describes the C99 language functions supported by the CX.

(1) Macros with variable numbers of arguments

The compiler supports C preprocessor macros with variable numbers of arguments.

```
#define pr (fmt, ...)         printf (fmt, __VA_ARGS__)
```

The notation above can be used to describe an arbitrary number of arguments.

```
pr ("%s%d\n", "aa", 1)  ->    printf ("%s%d\n", "aa", 1)
pr ("%d\n", 2)          ->    printf ("%d\n", 2)
```

(2) _Bool type

_Bool type is supported.

(3) Comment by //

Text from two slashes (//) until a newline character is a comment. If there is a backslash character (\) immediately before the newline, then the next line is treated as a continuation of the current comment.

(4) Inline keyword (inline function)

Inline function is supported.

This can also be specified using a pragma directive, via the following format.

```
#pragma inline  function-name[, function-name, ...]
```

For the details of expansion specification, see "(9) Inline expansion".

(5) long long int type

long long int type is supported. long long int type is 8-byte of integer type.
Appending "LL" to a constant value is also supported. It is also possible to specify this for bit field types.

(6) Comma permission behind the last enumeration child of a enum definition

When defining an enum type, it is permissible for the last enumerator in the enumeration to be followed by a comma (,).

```
enum EE {a, b, c,};
```

3.1.5 ANSI option

When the -Xansi option is specified by the CX, process strictly conforming to ANSI standards is executed.
The differences between when the -Xansi option is specified and when not specified are as follows.

Table 3-6. Processing When -Xansi Option Strictly Conforming to Language Specifications is Specified

Item	With -Xansi Specification	Without -Xansi Specification
Bit field	An error[Note 1] occurs if type other than int is specified for bit field.	Permits.
# line number	An error occurs.	Treated in same manner as "#line line number".[Note 2]
Character # in middle of line	An error occurs if character # appears in the middle of the line.	Outputs warning message and permits.
__STDC__	Defines value as macro with value 1.	Does not define.
Binary Constants	An error occurs if "0b" or "0B" is followed by one or more "0" or "1".	Treats "0b" or "0B" followed by one or more "0" or "1" as a binary constant.

Notes 1. Normal error beginning with "E". The same applies hereafter.
2. See the ANSI standards.

3.1.6 Internal representation and value area of data

This section explains the internal representation and value area of each type for the data handled by the CX.

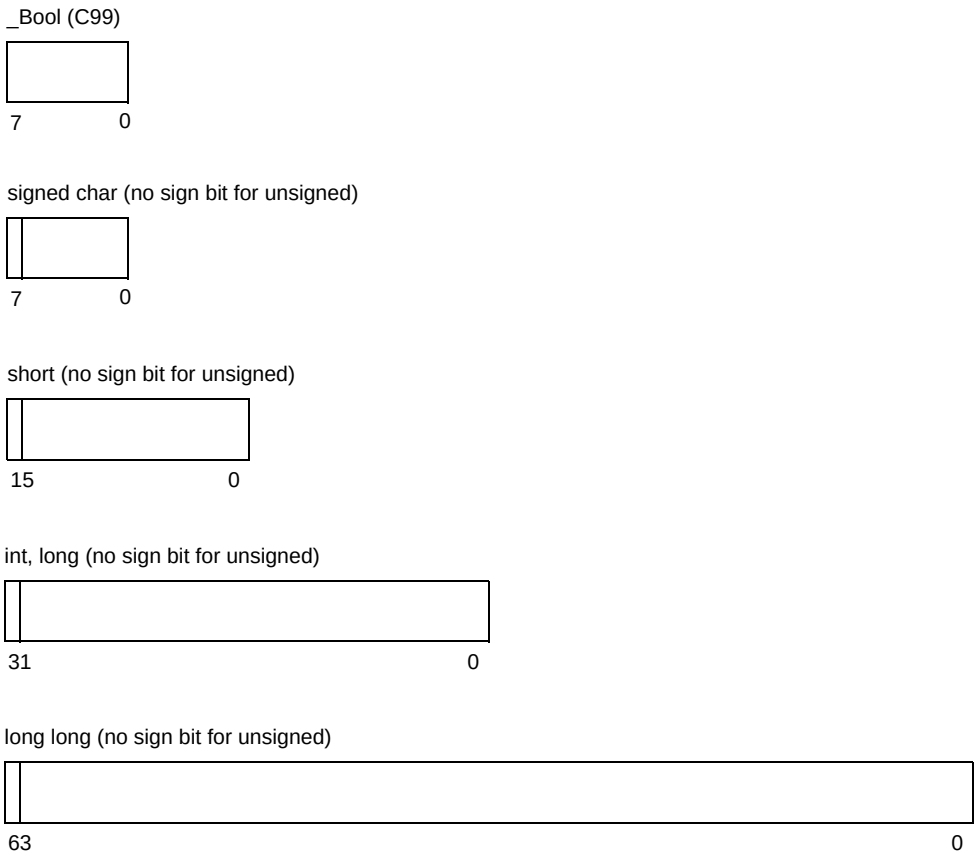
(1) Integer type

(a) Internal representation

The leftmost bit in an area is a sign bit with a signed type (type declared without "unsigned"). The value of a signed type is expressed as 2' s complement.

If the -Xchar=unsigned option is specified, however, a char type specified without "signed" or "unsigned" is assumed to be unsigned.

Figure 3-1. Internal Representation of Integer Type



(b) Value area

Table 3-7. Value Area of Integer Type

Type	Value Area
char[Note]	-128 to +127
short	-32768 to +32767
int	-2147483648 to +2147483647
long	-2147483648 to +2147483647
long long	-9223372036854775807 to +9223372036854775807

Type	Value Area
unsigned char	0 to 255
unsigned short	0 to 65535
unsigned int	0 to 4294967295
unsigned long	0 to 4294967295
unsigned long long	0 to 18446744073709551615

Note The value area is 0 to 255, if the -Xchar=unsigned option is specified by the CX.

(2) Floating-point type

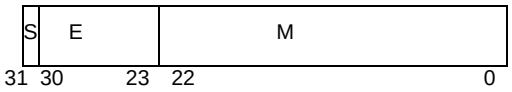
(a) Internal representation

Internal Representation of floating-point data type conforms to IEEE754[Note]. The leftmost bit in an area of a sign bit. If the value of this sign bit is 0, the data is a positive value; if it is 1, the data is a negative value.

Note IEEE**:** Institute of Electrical and Electronics Engineers
IEEE754 is a standard to unify specifications such as the data format and numeric range in systems that handle floating-point operations.

Figure 3-2. Internal Representation of Floating-Point Type

float

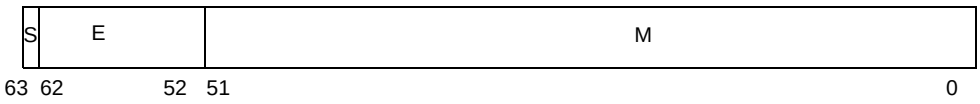


S: Sign bit of mantissa

E: Exponent (8 bits)

M: Mantissa (23 bits)

double



S: Sign bit of mantissa

E: Exponent (11 bits)

M: Mantissa (51 bits)

(b) Value area

Table 3-8. Value Area of Floating-Point Type

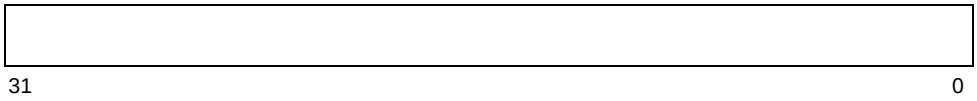
Type	Value Area
float	1.18×10^{-38} to 3.40×10^{38}
double	2.23×10^{-308} to 1.80×10^{308}

(3) Pointer type

(a) Internal representation

The internal representation of a pointer type is the same as that of an unsigned int type.

Figure 3-3. Internal Representation of Pointer Type

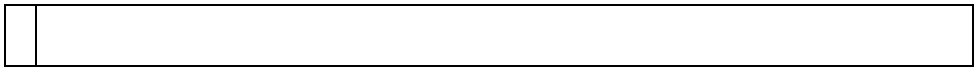


(4) Enumerate type

(a) Internal representation

The internal representation of an enumerate type is the same as that of a signed int type. The leftmost bit in an area of a sign bit.

Figure 3-4. Internal Representation of Enumerate Type



When the -Xenum_type=string option is specified, see "(30) Enumerate type specifier".

(5) Array type

(a) Internal representation

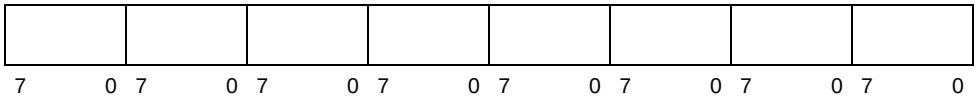
The internal representation of an array type arranges the elements of an array in the form that satisfies the alignment condition (alignment) of the elements

Example

```
char    a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

The internal representation of the array shown above is as follows.

Figure 3-5. Internal Representation of Array Type



(6) Structure type

(a) Internal representation

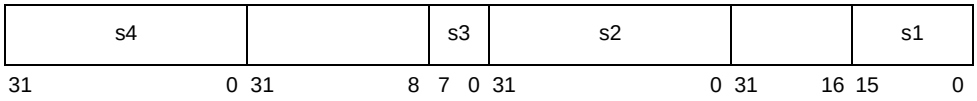
The internal representation of a structure type arranges the elements of a structure in a form that satisfies the alignment condition of the elements.

Example

```
struct {
        short   s1;
        int     s2;
        char    s3;
        long    s4;
} tag;
```

The internal representation of the structure shown above is as follows.

Figure 3-6. Internal Representation of Structure Type



For the internal representation when the structure type packing function is used, see "(12) Structure type packing".

(7) Union type

(a) Internal representation

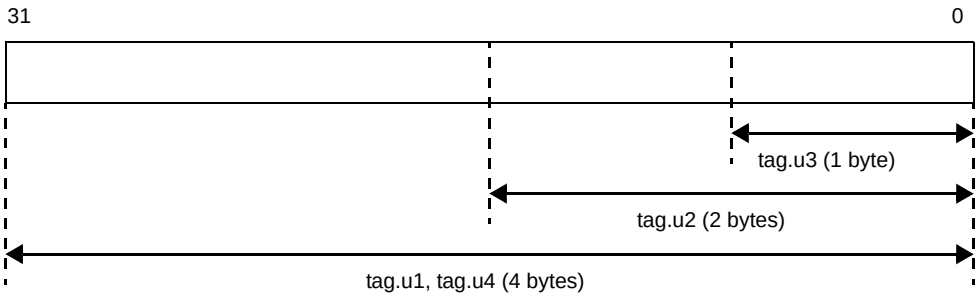
A union is considered as a structure whose members all start with offset 0 and that has sufficient size to accommodate any of its members. The internal representation of a union type is like each element of the union is placed separately at the same address.

Example

```
union {
        int     u1;
        short   u2;
        char    u3;
        long    u4;
} tag;
```

The internal representation of the union shown in the above example is as follows.

Figure 3-7. Internal Representation of Union Type



(8) Bit field

(a) Internal representation

An area including the declared number of bits is reserved for a bit field. The most significant bit of the area for a bit field declared to be of signed type is a sign bit.

The bit field declared first is allocated from the least significant bit of 4-byte area. If the alignment condition of the type specified in the declaration of a bit field is exceeded as a result of allocating an area that immediately follows the area of the preceding bit field to the bit field, the area is allocated starting from a boundary that satisfies the alignment condition.

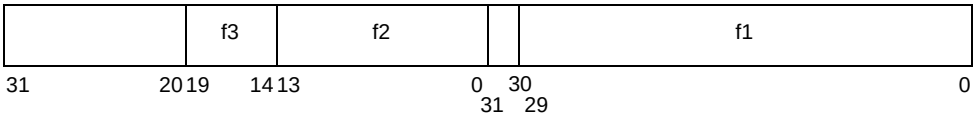
Note, however, that in the case of a bit field of type long long, then if the alignment conditions exceed the 64-bit boundary of the long long type, rather than the 4-byte boundary, then it will be allocated from the next boundary.

Example

```
struct {
        unsigned int    f1:30;
        int             f2:14;
        unsigned int    f3:6;
} flag;
```

The internal representation for the bit field in the above example is as follows.

Figure 3-8. Internal Representation of Bit Field



The ANSI standards do not allow char and short types to be specified for a bit field, but the CX allows char, short, long, long long and those unsigned types.

For the internal representation of bit field when the structure type packing function is used, see "(12) Structure type packing".

(9) Alignment condition

(a) Alignment condition for basic type

Alignment condition for basic type is as follows.

If the -Xinline_strcpy option of the CX is specified, however, all the arrey types are 4-byte boundaries.

Table 3-9. Alignment Condition for Basic Type

Basic Type	Alignment Conditions
(unsigned) char and its array type	Byte boundary
(unsigned) short and its array type	2-byte boundary
Other basic types (including pointer)	4-byte boundary

(b) Alignment condition for union type

The alignment condition for the union type varies as shown in Table 3-12, depending on the maximum member size.

Table 3-10. Alignment Condition for Union Type

Maximum Member Size	Alignment Conditions
2 bytes < size	4-byte boundary
Size <= 2 bytes	Maximum member size boundary

Here are examples of the respective cases:

Examples 1.

```
union  tug1 {
  unsigned short i; /*2 bytes member*/
  unsigned char  c; /*1 bytes member*/
};  /*The union is aligned with 2-byte.*/
```

2.

```
union  tug2 {
  unsigned int  i;  /*4 bytes member*/
  unsigned char  c; /*1 byte member*/
};  /*The union is aligned with 4-byte.*/
```

(c) Alignment condition for structure type

The alignment conditions for a structure type are the same as those of the structure's member whose type has the largest alignment condition.

If the -Xinline_strcpy option of the CX is specified, however, all the structure types are 4-byte boundaries.

Here are examples of the respective cases:

Examples 1.

```
struct  ST {
        char    c;      /*1 byte member*/
};  /*Structure is aligned with 1-byte.*/
```

2.

```
struct  ST {
        char    c;      /*1 byte member*/
        short   s;      /*2 bytes member*/
};  /*Structure is aligned with 2-byte.*/
```

3.

```
struct  ST {
        char    c;      /*1 byte member*/
        short   s;      /*2 bytes member*/
        short   s2;     /*2 bytes member*/
};  /*Structure is aligned with 2-byte.*/
```

4.

```
struct  ST {
        char    c;      /*1 byte member*/
        short   s;      /*2 bytes member*/
        int     i;      /*4 bytes member*/
};  /*Structure is aligned with 4-byte.*/
```

5.

```
struct  ST {
        char        c;      /*1 byte member*/
        short       s;      /*2 bytes member*/
        int         i;      /*4 bytes member*/
        long long   ll;     /*4 bytes member*/
};  /*Structure is aligned with 4-byte.*/
```

(d) Alignment condition for function argument

The alignment condition for a function argument is a 4-byte boundary.

(e) Alignment condition for executable program

The alignment condition when an executable object module file is created by linking object module files is 2-byte boundary.

3.1.7 General-purpose registers

How the CX uses the general-purpose registers are as follows.

The general-purpose registers includes the following functions.

(1) Software register bank

The number of the work registers (r10 through r19) and register variable registers (r20 through r29) used can be reduced by the -Xreg_mode option of the CX (see "3.1.9 Software register bank").

Table 3-11. Using General-Purpose Registers

Register		Usage
r0	Zero register	Used for operation as value of 0. Also used to reference data located at const section (read-only section placed in ROM area)Note.
r1	Assembler-reserved register	Used for instruction expansion by assembler.
r2 (hp)	Handler stack pointer	Reserved for system.
r3 (sp)	Stack pointer	Used to indicate beginning of stack frame.
r4 (gp)	Global pointer	Used to reference external variable.
r5 (tp)	Text pointer	Used to indicate beginning of text section (.text section)
r6 to r9	Argument registers	Used to pass argument.
r10 to r19	Work register	Used as work area during operation (r10 is also used to pass return value of function).
r20 to r29	Register variable registers	Used as an area for register variables.
r30 (ep)	Element pointer	Used to reference external variable specified to be located in internal RAM or external RAM sectionNote.
r31 (lp)	Link pointer	Used to pass return address of function.

Note For the allocation of data to a section, see "(2) Allocation of data to section".

3.1.8 Referencing data

How the CX references data are as follows.

Table 3-12. Referencing Data

Type	Referencing Method
Numeric constant	Immediate
Character string constant	Offset from global pointer (gp) Offset from element pointer (ep) Offset from r0
Automatic variable, Argument	Offset from stack pointer (sp)
External variable, Static variable in function	Offset from global pointer (gp) Offset from element pointer (ep) Offset from r0
Function address	Operated during execution by using offset from text pointer (tp)

3.1.9 Software register bank

Because the CX implements a register bank function by software, three register modes are provided. By specifying these register modes efficiently, the contents of some registers do not need to be saved or restored when an interrupt occurs or the task is switched. As a result, the processing speed can be improved. The register modes are specified by using the register mode specification option (-Xreg_mode) of the CX. This function reduces the number of registers internally used by the CX on a step-by-step basis. As a result, the following effects can be expected:

- The registers not used can be used for the application program (that is, a source program in assembly language).
- The overhead required for saving and restoring registers can be reduced.

Caution In an application program that has many variables to be allocated to registers by the CX, the variables so far allocated to a register are accessed from memory when a register mode has been specified. As a result, the processing speed may drop.

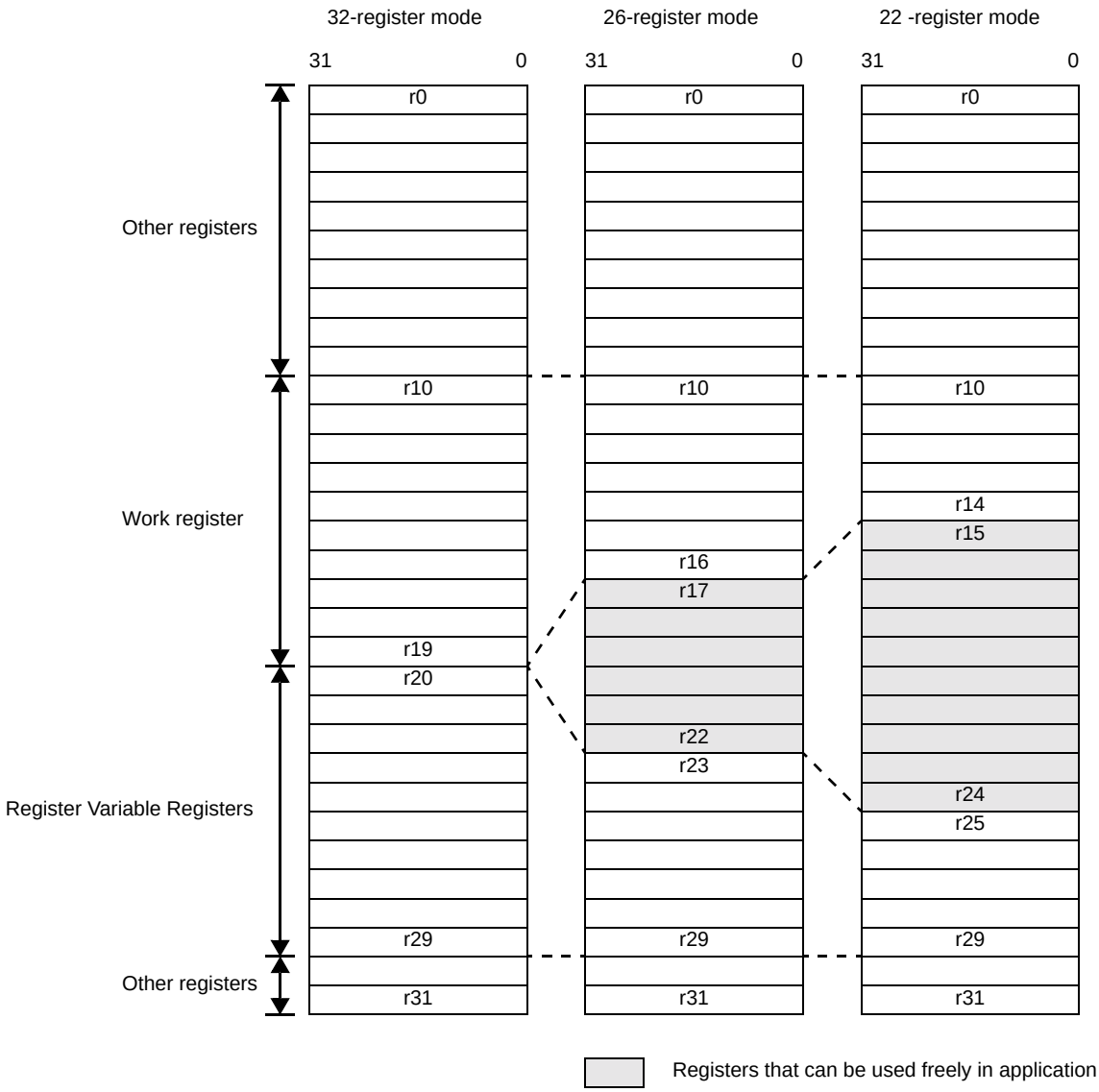
(1) Register mode

Next table and next Figure show the three register modes supplied by the CX.

Table 3-13. Register Modes Supplied by CX

Register modes	Work Register	Register Variable Registers
32-register mode (Default)	r10 to r19	r20 to r29
26-register mode	r10 to r16	r23 to r29
22-register mode	r10 to r14	r25 to r29

Figure 3-9. Register Modes and Usable Registers



Specification example on command line

```
> cx -Cf3507 -Xreg_mode=26 file.c <- compiled in 26-register mode
```

3.1.10 Device file

A device file is a binary file that contains information dependent upon the device type. One device file is available for each device or group of target devices as a package. The compiler referred a device file to generate object codes corresponding to the target system that is used in the application system. Therefore, place the device file to be used under the standard folder for the device file. If the device is placed under any other folder, specify the folder using a compiler option; otherwise an error occurs during compilation because the device file is not found.

(1) Specifying device file

A device file that is referenced by a program in C language can be specified in the following way.

(a) Specifying device name using compiler option (-C*device-name*)

Example

```
> cx -Cf3507 file.c
```

When building a program with CubeSuite, specifying a device has an effect equivalent to specifying this option.

In this example, the device name is "f3507" (V850E2/PJ4). The character strings that can be specified as "device name" dose not distinguish uppercase and lowercase characters.
For the character strings that can be specified as a device name, see the User's Manual of each device.

(2) Notes on specifying device file

(a) If no device name is specified

If a device name is not specified by the -C option, and if neither the -Xcommon=v850e option, nor the -Xcommon=v850e2 option, -Xcommon=v850e2v3[Note] is specified, the compiler outputs the error message and stops compiling. Note, however, that specifying the -V/-h/-P option will cause an error.

Note A device file is necessary during linking even if the -Xcommon=v850e, -Xcommon=v850e2 option or -Xcommon=v850e2v3 option is specified.

(b) Program described in assembler instructions

In this case also, a device must be specified by an assembler option or the PROCESSOR control instruction when an object module file that can be linked is created.

3.2 Extended Language Specifications

This section explains the extended language specifications supported by the CX.

The expanded specifications inclue how to specify section location of data and access the internal peripheral function registers of the device, how to insert assembler code in a C source program, how to specify inline expansion for each function, how to define a handler when an interrupt or exception occurs, how to disable interrupts at the C language level, the valid RTOS functions when a real-time OS is used for the target environment, and how to embed instructions in a C source program.

3.2.1 Macro name

All the following macro names are supported.

Macros not ending with "__" are supplied for the sake of former C language specifications (K&R specifications). To perform processing strictly conforming to the ANSI standards, use macros with "__" before and after.

Table 3-14. List of Supported Macros

Macro Name	Definition
__LINE__	Line number of source line at that point (decimal).
__FILE__	Name of assumed source file (character string constant).
__DATE__	Date of translating source file (character string constant in the form of "Mmm dd yyyy".) Here, the name of the month is the same as that created by the asctime function stipulated by ANSI standards (3 alphabetic characters with only the first character is capital letter) (The first character of dd is blank if its value is less than 10).
__TIME__	Translation time of source file (character string constant having format "hh:mm:ss" similar to the time created by the asctime function).
__STDC__	Decimal constant 1 (defined when the -Xansi option is specified[Note])
__v850 __v850__	Decimal constant 1.
__v850e __v850e__	Decimal constant 1 (defined by the CX, if V850Ex is specified as a target device).
__v850e2 __v850e2__	Decimal constant 1 (defined by the CX, if V850E2 is specified as a target device).
__v850e2v3 __v850e2v3__	Decimal constant 1 (defined by the CX, if device with an instruction set of V850E2V3 is specified as a target device).
__K0R __K0R__	Decimal constant 1 (defined by the CX, if 78K0R is specified as a target device).
__CX __CX__	Decimal constant 1.
__CHAR_SIGNED__	Decimal constant 1 (defined if signed is specified by the -Xchar option and when the -Xchar option is not specified).
__CHAR_UNSIGNED__	Decimal constant 1 (defined when unsigned is specified by the -Xchar option).
__DOUBLE_IS_64BITS__	Decimal constant 1.
CPUmacro	Decimal constant 1 of a macro indicating the target CPU. A character string indicated by "product type specification" in the device file with "_ _" prefixed and suffixed is defined.

Macro Name	Definition
Register mode macro	Decimal constant 1 of a macro indicating the target CPU. Macro defined with register mode is as follows. 32 register mode: __reg32__ 26 register mode: __reg26__ 22 register mode: __reg22__ Universal register mode: __reg_common__
_MULTI_CORE__	Decimal constant 1 (defined when the -Xmulti option is specified)
__MULTI_CMN__	Decimal constant 1 (defined when the -Xmulti=cmn option is specified)
__MULTI_PE*n*__	Decimal constant 1 (defined when the -Xmulti=pe*n* option is specified)

Note For the processing to be performed when the -Xansi option is specified, see "3.1.5 ANSI option".

3.2.2 Keyword

The CX adds the following characters as a keyword to implement the expanded function. These words are similar to the ANSI C keywords, and cannot be used as a label or variable name.

Keywords that are added by the CX are listed below.

_bsh, _bsw, __caxi, data, __DI, __EI, _halt, _hsw, __ldgr, __ldsr, __mul, __mulu, _mul32, _mul32ut, _nop, _sasf, _satadd, _satsub, __sch0l, __sch0r, __sch1l, __sch1r, __set_il, __stgr, __stsr, _sxb, _sxh

3.2.3 #pragma directive

The CX can specify the following #pragma directives.

(1) Description with assembler instruction

Assembler directives can be described in a C source program.

For the details of description, see "(5) Describing assembler instruction".

```
#pragma asm
        assembler instruction
#pragma endasm
```

(2) Inline expansion specification

A function that is expanded inline can be specified.

For the details of expansion specification, see "(9) Inline expansion".

```
#pragma inline  function-name[, function-name ...]
```

(3) Data or program memory allocation

(a) section

Allocates variables to an arbitrary section.

For details about the allocation method, see "(2) Allocation of data to section".

(b) text

A function to be allocated in a text section with an arbitrary name can be specified.

For details about the allocation specification, see "(3) Allocating functions to sections".

```
#pragma section section-type    ["section-name"]
#pragma text    ["section-name"] [Function-name[, Function-name]...]
```

(4) Peripheral I/O register name validation specification

The peripheral I/O registers of a device are accessed by using peripheral function register names. #pragma directive should be specified, when the program is executed by using the Peripheral I/O register name as it is.

```
#pragma ioreg
```

(5) Interrupt/exception handler specification

Interrupt/Exception handlers are described in C language.

For details, see "(c) Describing interrupt/exception handler".

```
#pragma interrupt   interrupt-request-name function-name [allocation-method] [Option
[Option]...]
```

(6) Interrupt disable function specification

Interrupts are disabled for the entire function.

```
#pragma block_interrupt function-name
```

(7) Task specification

Task that runs on an RTOS is described in the C language.

For details, see "(a) Description of task".

```
#pragma rtos_task   [Function-name]
```

(8) Structure type packing specification

Specifies the packing of a structure type. The packing value, which is an alignment value of the member, is specified as the numeric value. A value of 1, 2, 4, or 8 can be specified. When the numeric value is not specified, the setting is the default 8[Note] assumption.

```
#pragma pack([1248])
```

Note Alignment values "4" and "8" are treated as the same in this version.

(9) Smart correction specification

Specifies the function of smart correction.

For the details of description, see "(13) Smart correction function".

```
#pragma smart_correct Function-name Function-name
```

(10) Position independent access

Specify position independent access. When this is specified, accesses subsequently declared and defined variables and functions will use relative addresses.

For the details of description, see "(14) Position independent operations".

```
#pragma pic
```

(11) Fixed address access

Specify fixed address access. When this is specified, accesses to subsequently declared and defined variables and functions will use absolute addresses.

For the details of description, see "(14) Position independent operations".

```
#pragma nopic
```

3.2.4 Using expanded specifications

This section explains using expanded specifications.

- Constant notation
- Allocation of data to section
- Allocating functions to sections
- Peripheral I/O register
- Describing assembler instruction
- Controlling interrupt level
- Disabling interrupts
- Interrupt/Exception processing handler
- Inline expansion
- Real-time OS support function
- Embedded functions
- Structure type packing
- Smart correction function
- Position independent operations

(1) Constant notation

The CX allows constants to be written in binary or octal notation. Binary constants must consist of an "0b" or "0B", followed by a string of "1"s and "0"s. Octal constants must consist of an "0o", followed by a string of numbers between "0" and "7".

Example

```
0b00010110111110101011111010010111
0o001726354
```

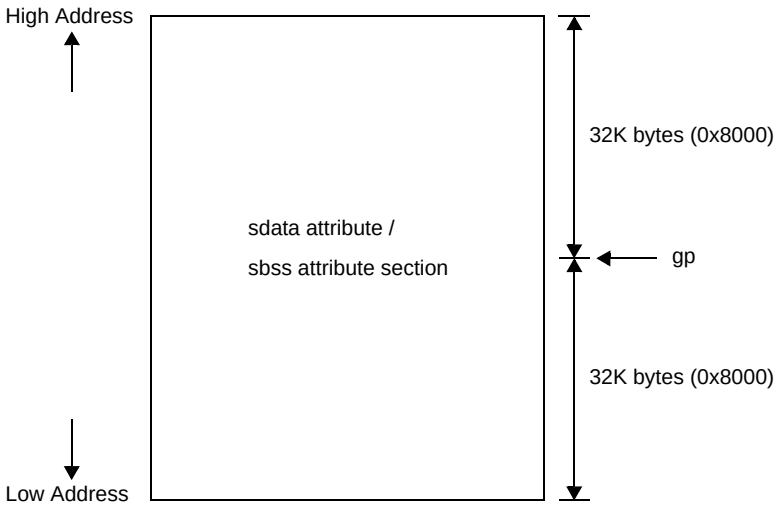
If a binary or octal constant is used, specifying the -Xansi option will cause an error message to be output.

(2) Allocation of data to section

When external variables and data are defined in a C source, the CX allocates them to memory. The memory location to which the variables and data are allocated is, basically, an area that can be referenced by an offset from the address pointed to by the global pointer (gp). If the variables or data are accessed in the program, therefore, the CX attempts to output a code that accesses the area using gp, by default.

At this time, the CX attempts to output a code that allocates data to an area that can be referenced from gp by one instruction, in order to enhance the object efficiency and execution efficiency as much as possible. Since the range that can be referenced by one instruction from gp must be within ±32 K bytes (a total of 64 K bytes) from gp according to the V850 architecture, the CX has dedicated sections in the ±32 K bytes area from gp. These sections are called the sdata and sbss attribute sections.

Figure 3-10. sdata and sbss Attribute Sections



In many cases, however, variables exceed in this range when using an application that uses many variables. In this case, the variables must be allocated to other sections. The CX has many sections to which variables and data can be allocated, in addition to the sdata and sbss attribute sections. Each of these sections has its own feature and sections to which variables that must be accessed quickly can be allocated are also available, so that the sections can be selected depending on the usage.

The sections that can be used in the CX including sdata and sbss attribute sections are explained below.

- sdata and sbss attribute sections

  These sections can be referenced from gp with one instruction and must be allocated within ± 32 K bytes from gp. Data with initial values is allocated to the sdata attribute section, and data without initial values is allocated to sbss attribute section.

  The CX first attempts to generate a code that is to be allocated to these sections.

  If the code exceeds the upper limit of the section of these attributes, the compiler generates a code that allocates data to a data or bss attribute section.

  To increase the amount of data to be allocated to the sdata or sbss attribute sections, the upper size limit for the data to be allocated can be specified with the "-G" option of the CX so that data in excess of this upper limit is not allocated to the sdata or sbss attribute sections (see "CubeSuite Build for CX Compiler" for details of this option).

  Use the #pragma section directive to specify a variable to be allocated to the sdata or sbss attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section sdata
int a = 1;  /*Allocated to sdata attribute section*/
int b;      /*Allocated to sbss attribute section*/
#pragma section default
```

- data and bss attribute sections

These sections can be referenced from gp with two instructions. Since access is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed.

In other words, these sections can be allocated anywhere as long as they are in RAM.

Use the #pragma section directive to specify a variable to be allocated to the data or bss attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section data
int a = 1;  /*Allocated to data attribute section*/
int b;      /*Allocated to bss attribute section*/
#pragma section default
```

- sconst-attribute section

This is a section that can be referenced from r0, in other words from address 0, with 1 instruction, and must be allocated within +32K bytes from address 0. Basically, data that can be fixed to ROM is allocated to this section. In the case of V850 devices with internal ROM, in many cases the internal ROM is assigned from address 0, and data that should be referenced with 1 instruction and that can be fixed to ROM is allocated to the sconst attribute section. Variables/data declared by adding the const qualifier are subject to allocation to the sconst attribute section. If the data exceeds the upper limit of this attribute section, it is allocated to the const attribute section.

To increase the amount of data to be allocated to the sconst attribute section, the upper size limit for the data to be allocated can be specified with the -Xsconst option of the CX so that data in excess of this upper limit is not allocated to the sconst attribute section (see "CubeSuite Build for CX Compiler" for details of this option).

Use the #pragma section directive to specify a variable to be allocated to the sconst attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section sconst
const int   a = 1;  /*Allocated to sconst attribute section*/
#pragma section default
```

- const-attribute section

This is a section that can be referenced from r0, in other words from address 0, with two instructions. Data that can be fixed to ROM that exceeds the upper limit of the sconst attribute section, or data that should be allocated to external ROM in the case of ROMless devices of the V850 microcontrollers, is allocated to the const attribute section. Variables/data declared by adding the const qualifier are subject to allocation to the const attribute section.

The variables declared by adding the const qualifier are allocated to the const attribute section, string literal even if allocation to the .const section is not specified by the #pragma section directive. Since access is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed. In other words, these sections can be allocated anywhere as long as they are in 32-bit space.

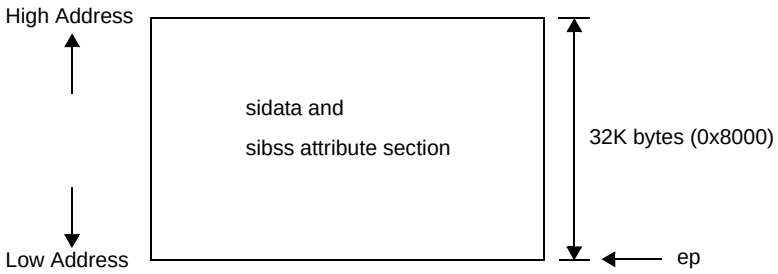
Use the #pragma section directive to specify a variable to be allocated to the const attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section const
const int   a = 1;  /*Allocated to const attribute section*/
#pragma section default
```

- sidata and sibss attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward higher addresses. In other words, these sections are allocated in the 32 K bytes space toward higher addresses from ep.

Figure 3-11. sidata and sibss Attribute Sections



Data with initial values is allocated to the sidata attribute section, and data without initial values is allocated to sibss attribute section. If variables that exceed the upper limit of the sdata and sbss attribute sections that can be accessed from gp with 1 instruction, but which need to be accessed with 1 instruction still exist, they canbe allocated in the range that can be accessed with 1 instruction using ep.

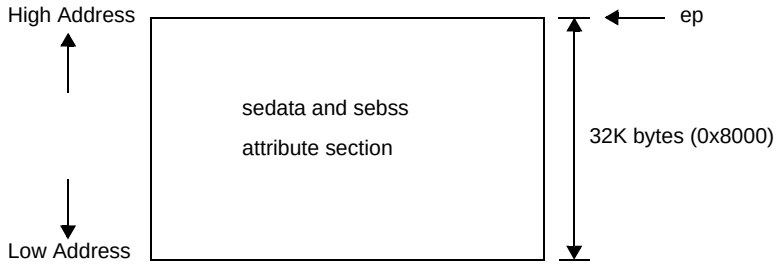
Use the #pragma section directive to specify a variable to be allocated to the sidata or sibss attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section sidata
int a = 1;  /*Allocated to sidata section*/
int b;      /*Allocated to sibss section*/
#pragma section default
```

- sedata and sebss attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward lower addresses from ep. In other words, these sections are allocated in the 32 K bytes space toward lower addresses from ep.

Figure 3-12. sdata and sbss Attribute Sections



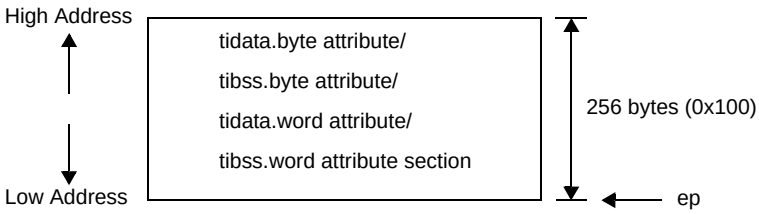
Data with initial values is allocated to the sedata attribute section, and data without initial values is allocated to sebss attribute section. If variables that exceed the upper limit of the sdata and sbss attribute sections that can be accessed from gp with 1 instruction, but which need to be accessed with 1 instruction still exist, they can be allocated in the range that can be accessed with 1 instruction using ep.
Use the #pragma section directive to specify a variable to be allocated to the sedata or sebss attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section sedata
int a = 1;  /*Allocated to sedata section*/
int b;      /*Allocated to sebss section*/
#pragma section default
```

- tidata (tidata.byte, tidata.word) and tibss (tibss.byte, tibss.word) attribute sections

These sections can be referenced from ep (element pointer) with 1 instruction toward higher addresses.
These sections are accessed with 1 instruction in the same manner as the sidata and sibss attribute sections, but differ in terms of the assembler instruction to be used.
The sidata and sibss attribute sections use the 4-byte st/ld instruction for store/reference, whereas the tidata and tibss attribute sections use the 2-byte sst/sld instruction to perform access. In other words, the code efficiency of the tidata and tibss attribute sections is better than that of the sidata and sibss attribute sections.
However, the range in which sst/sld instructions can be applied is small, so it is not possible to allocate a large number of variables.

Figure 3-13. tidata and tibss Attribute Sections



Data with initial values is allocated to the tidata (tidata.byte, tidata.word) attribute section, and data without initial values is allocated to the tibss (tibss.byte, tibss.word) attribute section. Specify the tidata.byte/tibss.byte attribute to allocate byte data, and specify the tidata.word/tibss.word attribute to allocate word data. To select automatic byte/word judgment by the CX, specify the tidata/tibss attribute.
The tidata and tibss attribute sections are used to allocate data that must be accessed the fastest in the system.
However, the data to be allocated to these sections must be carefully selected because the quantity of data that can be allocated to these sections is limited. Use the #pragma section directive to specify variables to be allocated to the tidata.byte/tibss.byte or tidata.word/tibss.word attribute section in the program (see "(a) #pragma section directive" for details).

```
#pragma section tidata_byte
char            a = 1;  /*Allocated to tidata.byte attribute section*/
unsigned char   b;      /*Allocated to tibss.byte attribute section*/
#pragma section default
```

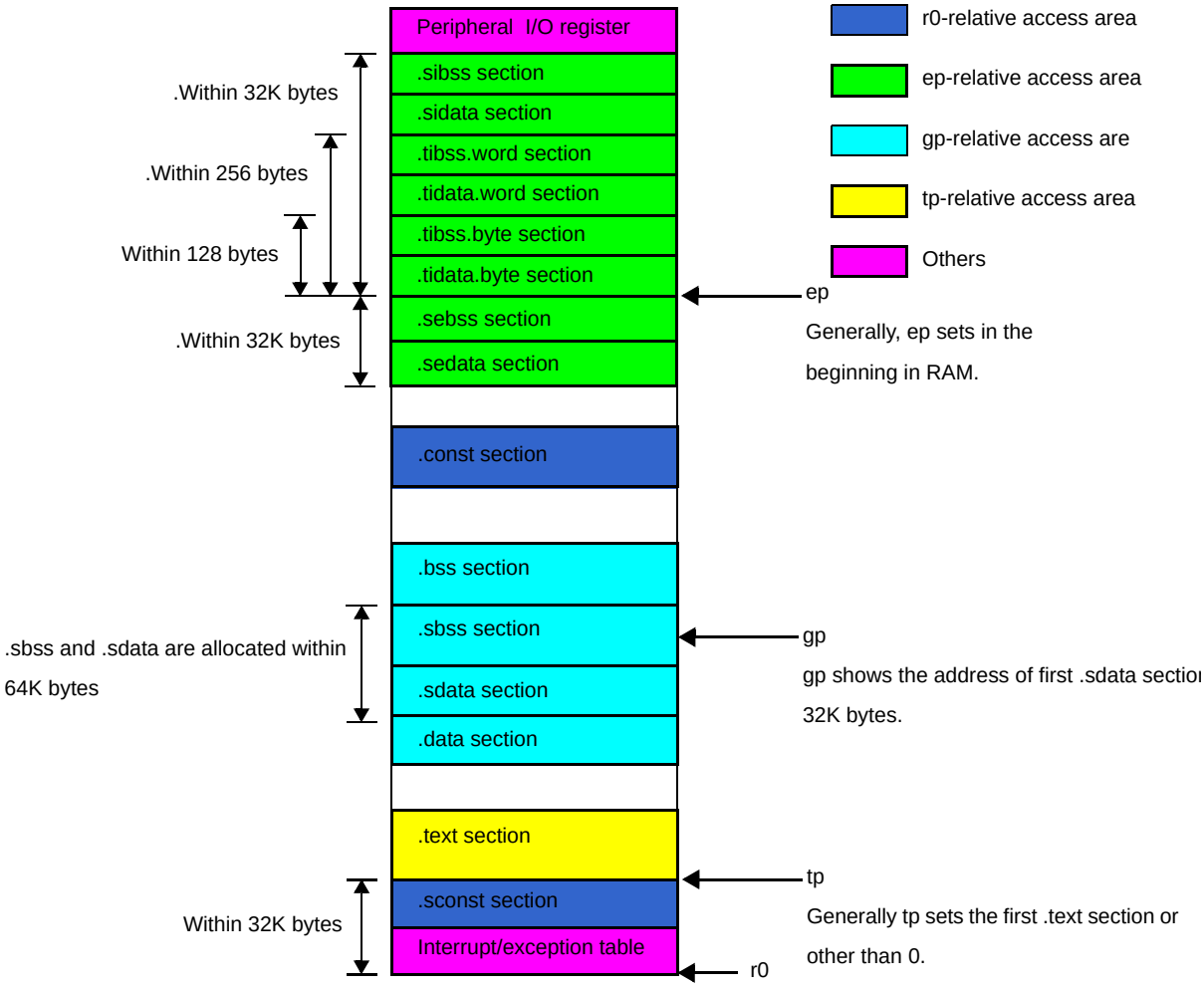
```
#pragma section tidata_word
int     a = 1;          /*Allocated to tidata.word attribute section*/
short   b;              /*Allocated to tibss.word attribute section*/
#pragma section default
```

```
#pragma section tidata
int     a = 1;          /*Allocated to tidata.word attribute section*/
char    b;              /*Allocated to tibss.byte attribute section*/
#pragma section default
```

The efficiency can be enhanced in terms of execution speed if variables or data that are especially frequently used in the system are selected and allocated to the tidata (tidata.byte, tidata.word) or tibss (tibss.byte or tibss.word) attribute section. The CX has a function that investigates the frequency of reference. The code that allocates data to the tidata (tidata.byte, tidata.word) or tibss (tibss.byte, tibss.word) attribute section is output based on this information.

Following figure shows an example of memory allocation of each section

Figure 3-14. tidata and tibss Attribute Sections



(a) #pragma section directive

How to allocate data to a section using the #pragma section directive is explained below.

<1> By default, when the section name is used as it is

Describe the #pragma section directive in the following format when using the section name defined by the CX.

```
#pragma section section-type
Variable declaration/definition
#pragma section default
```

The following can be specified as the section-type.

data, sdata, sedata, sidata, tidata, tidata_word, tidata_byte, sconst, const

The name of the bss attribute section must not be specified as the section type. The CX automatically allocates declared or defined data with initial values to the data attribute section, and data without initial values to the bss attribute section.

```
#pragma section sdata
int a = 1;  /*allocated to sdata attribute section*/
int b;      /*allocated to sbss attribute section*/
#pragma section default
```

In the above case, "variable a" is allocated to the data-attribute .sdata section because it has an initial value, and "variable b" is allocated to the bss-attribute .sbss section because it does not have an initial value.

Two or more variable declarations or definitions can be described between "#pragma section *section-type*" and "#pragma section default". Enumerate variables to be allocated for each section type.

Use "_" (underscore) instead of "." (period) to specify tidata.word or tidata.byte as the section type, as shown below.

tidata_word, tidata_byte

<2> To assign original section name

The user can assign a specific name to the sections, and can allocate variables and data to those sections.

In this case, describe the #pragma section directive in the following format.

```
#pragma section section-type    "created-section-name"
Variable declaration / Definition
#pragma section default
```

However, ".*section-type*" is appended to a section name actually generated by this method as follows.

```
created-section-name.section-type
```

This is to prevent a section with another attribute and having the same name from being created because the section attribute is classified into data or bss attribute depending on whether the data has an initial value or not. Specify a generated section name when specifying a section in a link directive file. See "(b) Specifying link directive of specific data section" for an example of specification in a link directive file. The following table shows specific examples of section names specified by the user and generated section names.

Table 3-15. Specified Section Names and Generated Section Names

Section Name Specified by User	Section Type	Character String Appended	Generated Section Name
mydata	data attribute	.data/.bss	mydata.data/mydata.bss
mysdata	sdata attribute	.sdata/.sbss	mysdata.sdata/mysdata.sbss
myconst	const attribute	.const	myconst.const

If the name is specified as follows, "variable a" is allocated to the mysdata.sdata section because it has an initial value, and "variable b" is allocated to the mysdata.sbss section because it does not have an initial value.

```
#pragma section sdata   "mysdata"
int a = 1;  /*allocated to mysdata.sdata attribute section*/
int b;      /*allocated to mysdata.sbss attribute section*/
#pragma section default
```

(b) Specifying link directive of specific data section

Specifying link directive of specific data section when a specific section is created using the #pragma section directive, describe that section in a link directive file as explained below.
If "variable a" and "variable b" are specified as follows in a C source, "variable a" is allocated to the mysdata.sdata section because it has an initial value, and "variable b" is allocated to the mysdata.sbss section because it does not have an initial value.

```
#pragma section sdata   "mysdata"
int a = 1;  /*allocated to mysdata.sdata attribute section*/
int b;      /*allocated to mysdata.sbss attribute section*/
#pragma section default
```

At this time, the variable can be allocated to a specific section if the mapping directive in the link directive file is described as follows.

```
.sdata          = $PROGBITS     ?AWG .sdata;
.sbss           = $NOBITS       ?AWG .sbss;
mysdata.sdata   = $PROGBITS     ?AWG mysdata.sdata;
mysdata.sbss    = $NOBITS       ?AWG mysdata.sbss;
```

Since the variables are allocated in the order in which they are described, change the description order to change the allocation order. It is also possible to specify the start address of the section directly (generally, a segment is created first and a mapping directive, which specifies the start address of a section in segment units, is then described in that segment).

It must be noted here that mysdata.sdata's "$PROGBITS ?AWG" attribute and input section, and mysdata.sbss's "$NOBITS ?AWG" attribute and input section (above, "$PROGBITS ?AWG mysdata.sdata" and "$NOBITS ?AWG mysdata.sbss" at the far right of the mapping directive) must not be omitted.

(c) Notes on section allocation

Notes below must be noted when sections are allocated by the #progma section directive, the const qualifier, or the section file.

<1> If a section is specified for an automatic variable, the specification is ignored. Section specification is a function for external variables, character string and static variable.

<2> A variable declaration that is not set with an initial value is usually treated as a tentative definition. When a section is specified, however, it is treated as a "definition". Do not allow variable declarations, which do not have their initial values, set to get mixed in with definitions.

[Variable declaration not using #pragma section]	[Variable declaration using #pragma section]
int i; /*tentative definition*/	#pragma section sedata
int i = 10; /*definition*/	int i; /*definition*/
	int i = 10; /*definition*/
[Error does not occur.]	#pragma section default
	[Duplicated definition error.]

Be sure to make extern declaration in files that reference an external variable. In the example below, a duplicated definition error occurs if extern is missing in the tentative definition of the variable in file1.c.

[file1.c]	[file2.c]
#pragma section sedata	#pragma section sedata
extern int i;	int i;
#pragma section default	#pragma section default
[Duplicated definition error occurs if extern is not declared]	

<3> When a variable specified by a section is referenced by another file, the section must be specified with the same section type for the extern declaration of that variable. An error occurs if a type of section different from that of the section specified when a variable is defined is specified.
For example, if "#pragma section data begin - #pragma section default" is specified on the definition side and "#pragma section data begin - #pragma section default" is not specified on the tentative definition side (extern declaration), it is assumed on the tentative definition side that the variable is allocated to the sdata section. This means that a code that accesses the variable from gp with two instructions is output on the definition side and that a code that accesses the variable from gp with one instruction is output on the tentative definition side. In other words, a contradiction occurs. Consequently, the error message is output during linking.

Example Correct specification

[file1.c]	[file2.c]
#pragma section sedata	#pragma section sedata
int i = 1;	extern int i;
#pragma section default	#pragma section default

Example Incorrect specification 1

[file1.c]	[file2.c]
int i = 1;	#pragma section sedata
	extern int i;
	#pragma section default

"variable i" defined by file1.c is allocated to the sbss or bss attribute section. However, file2.c outputs a code that accesses the sebss attribute section for "variable i". As a result, the linker outputs the error message.

Example Incorrect specification 2

[file1.c]	[file2.c]
#pragma section sedata	extern int i;
int i = 1;	
#pragma section default	

"variable i" defined by file1.c is allocated to the sbss or bss attribute section. However, file2.c outputs a code that accesses the sebss attribute section for "variable i". As a result, the linker outputs the error message.

<4> Although a variable specified as const is allocated to the const section, if a #pragma section directive specifies other than const/sconst, then it will be allocated to the specified section.

<5> When defining a variable with the sconst or const attribute using the #pragma section directive, be sure to make a const specification for the variable. A const specification is also necessary at the location of the tentative definition made by extern declaration.

If the const declaration is missing when a variable is declared, the variable is not allocated to the sconst section or const section (the #pragma section directive is ignored) even if "#pragma section sconst" or "#pragma section const" is specified, but to a gp-relative section such as the sdata section or data section. In other words, allocation is not performed as intended.

[file1.c]	[file2.c]
#pragma section sconst	#pragma section sconst
const int i = 1;	int i;
#pragma section default	#pragma section default

A code that allocates "variable i" to the sconst section is output in file1.c. In file2.c, however, the #pragma section specification is ignored because the const specification is missing from "variable i", and therefore the variable is treated as a gp-relative variable. In other words, a code that allocates the variable to the sdata or data section is output. Consequently, "variable i" is not allocated to the sconst section during linking.
A const specification is also necessary at the location of the tentative definition with extern declaration, as shown below.

[file1.c]	[file2.c]
#pragma section sconst	#pragma section sconst
const int i = 10;	extern const int i;
#pragma section default	#pragma section default

<6> In #pragma section, it is not possible specify variables with unknown sizes, arrays with unknown numbers of elements, undefined structures, or structures including any of these.

<7> If the -Xsdata and -Xsconst options are specified, and a #pragma section is specified, then the specification of the #pragma section is effective. If there is no #pragma section, or if "default" was specified in the relocation attribute, then the option specification is effective.

(d) Example of #pragma section directive

Here are some examples of using the #pragma section directive.

<1> Allocating "variable a" to tidata.word section and "variable b" to tibss.word section

```
#pragma section tidata_word
int     a = 1;      /*allocated to tidata.word attribute section*/
short   b;          /*allocated to tibss.word attribute section*/
#pragma section default
```

<2> Allocating "variable c" to tidata.byte section and "variable d" to tibss.byte section

```
#pragma section tidata_byte
char    c = 0x10;   /*allocated to tidata.byte section*/
char    d;          /*allocated to  tibss.byte section*/
#pragma section default
```

In the tidata attribute section, word data or halfword data is allocated to the tidata_word or tibss_word section, and byte data is allocated to the tidata_byte or tibss_byte section.

<3> Allocating "variable e" specified by const to the sconst section and character string constant data indicated by pointer p to sconst section.

```
#pragma section sconst
const   int     e = 0x10;
const   char    *p = "Hello, World";
#pragma section default
```

In the above description, "Hello World" indicated by pointer p is allocated to the sconst section, and pointer variable "p" itself is allocated to the sdata section or data section. The allocation position of the pointer variable and the contents indicated by the pointer vary depending on how const is specified.

Examples 1.

```
const   char    *p = "Hello, World";
```

If this declaration is made, the pointer variable and character sting constant indicated by the pointer are

Pointer variable "p"	Can be rewritten ("p = 0" can be compiled).
Character string constant "Hello World"	Cannot be rewritten ("p = 0" cannot be compiled).

Describe as shown below to allocate what the pointer variable indicates to a section with the const attribute.

```
#pragma section sconst
const   char    *p = "Hello, World";
#pragma section default
```

The above definition allocates the pointer variable and constant to the following sections.

Pointer variable "p"	sdata/data section
Character string constant "Hello World"	sconst section

2.

```
char    *const  p;
```

Pointer variable "p"	Cannot be rewritten ("p = 0" cannot be compiled).

Describe as shown below to allocate the pointer variable to a section with the const attribute.

```
char    *const  p = "Hello World";
```

The above description allocates both the pointer variable and character string constant "Hello World" to a section with the const attribute.

```
#pragma section sconsts
char    *const  p = "Hello World";
#pragma section default
```

The above definition allocates the pointer variable and constant to the following sections.

Pointer variable "p"	sconst section

Character string constant "Hello World"	sconst section

3.

```
const   char    *const  p;
```

Pointer variable "p"	Cannot be rewritten ("p = 0" cannot be compiled).

Describe as shown below to allocate what the pointer variable indicates to a section with the const attribute. This description is used when the pointer itself is fixed to ROM.

```
const   char    *const  p = "Hello World";
```

The above description allocates both the pointer variable and character string constant "Hello World" to a section with the const attribute.

```
#pragma section sconst
const   char    *const  p = "Hello World";
#pragma section default
```

The above definition allocates the pointer variable and constant to the following sections.

Pointer variable "p"	sconst section
Character string constant "Hello World"	sconst section

In addition to the #pragma section directive, the compiler option "-Xconst" can be used to allocate a variable specified by const to the sconst section.

<4> Make the extern declaration of the #pragma section directive in a commonly used header file and include it in the C source.

```
[header.h]
#pragma section sidata
extern  int k;
#pragma section default
```

```
[file1.c]
#include    "header.h"
#pragma section sidata
int k;
#pragma section default
```

```
[file2.c]
#include    "header.h"
void func1(void) {
        k = 0x10;
}
```

If the extern declaration of the #pragma section directive is made in a header file as shown above, the errors decrease and the source is visually simplified.

(3) Allocating functions to sections

The CX allocates the functions of a C source program, i.e., program codes, to the .text section by default. When the text section allocation address is specified in the link directive file, the program is allocated from that address. However, it may be necessary to change the allocation address for each function or distribute the allocation address because of the layout of the memory. To satisfy these necessities, the CX has the #pragma text directive. Using this directive, any name can be given to a section with the text attribute, and the allocation address can be changed in the link directive file.

(a) #pragma text directive

Using the #pragma text directive, any name can be given to a section with the text attribute. The #pragma text directive can be used in the following two ways

<1> Specifying the function name to be allocated to a section to be created using the #pragma text directive.

```
#pragma text    "created section name"     function-name[, unction-name]...
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1". The created section name can be omitted. In this case, a function specified by "function name" is allocated to the default .text section.

<2> Describing the #pragma text directive before the main body of a function (function definition) but not specifying a function name.

```
#pragma text    "created section name"
```

The created section name can be omitted. In this case, specification of the name of section to be created by "#pragma text" specified immediately before is canceled, and the subsequent functions are allocated to the default .text section.
However, ".section-type" is appended to a section name actually generated by this method as follows.

```
section-name.text
```

Specify a generated section name when specifying a section in a link directive file. See "(b) Specifying link directive of specific data section" for an example of specification in a link directive file.
The following table shows specific examples of section names specified by the user and generated section names.

Table 3-16. Specified Section Names and Generated Section Names

Section Name Specified by User	Section Type	Character String Appended	Generated Section Name
mytext	text attribute	.text	mytext.text

If the name is specified as follows, "func1" is allocated to the mytext1.text section, and "func2" is allocated to the .text section by default, because the #pragma text directive is not used.

```
#pragma text    "mytext1"       func1
void func1(void) {
  :
}

void func2(void) {
  :
}
```

If the name is specified as follows, "func1" and "func2" are allocated to the mytext2.text section, "func3" to the "mytext3.text section", and "func4" to the default .text section because the #pragma text "mytext3" immediately before is cancelled.

```
#pragma text    "mytext2"
void func1(void) {
  :
}

void func2(void) {
  :
}

#pragma text    "mytext3"
void func3(void) {
  :
}

#pragma text
void func4(void) {
  :
}
```

(b) Specifying link directive of specific data section

When a specific section is created using the #pragma section directive, describe that section in a link directive file as explained below.

```
#pragma text    "mytext2"
void func1(void) {
  :
}

void func2(void) {
  :
}

#pragma text    "mytext3"
void func3(void) {
  :
}

#pragma text
void func4(void) {
  :
}
```

If the name is specified as follows, "func1" and "func2" are allocated to the mytext2.text section, "func3" to the "mytext3.text section", and "func4" to the default .text section because the #pragma text "mytext3" immediately before is cancelled.

```
text = $PROGBITS    ?AX .text;
mytext2 = $PROGBITS ?AX mytext2.text;
mytext3 = $PROGBITS ?AX mytext3.text;
```

Since the variables are allocated in the order in which they are described, change the description order to change the allocation order. It is also possible to specify the start address of the section directly (generally, a segment is created first and a mapping directive, which specifies the start address of a section in segment units, is then described in that segment).
Because the attribute of mytext2.text and mytext3.text is "$PROGBITS ?AX", do not omit the input section (".text", "mytext2.text", and "mytext3.text" on the rightmost side of the mapping directive in the above example) from mapping directives that have the same attribute as these.

Example If an input section is omitted from a mapping directive having the same "$PROGBITS?AX" attribute, the linker links and locates all the sections having that attribute. Consequently, data is not allocated to the specific section created by the user.
This means that the program that should be allocated to the mytext2.text or mytext3.text section is allocated to the .text section.

```
.text = $PROGBITS       ?AX;
```

(c) Notes on #pragma text directive.

Note the following points when using the #pragma text directive

- Describe the #pragma text directive before the function definition in the same file; otherwise a warning message is output and the directive is ignored. However, the order of prototype declaration of a function is not affected.
- After a #pragma text that specifies a function name, if a #pragma text is written that does not specify a function, then the specified function is allocated to the specified section, and the non-specified function will be allocated in accordance with a subsequent #pragma text.
- If a function specified by the #pragma text directive is an interrupt handler specified as direct allocation, a warning message is output and the #pragma text directive is ignored. See "(8) Interrupt/Exception processing handler" for details of direct allocation specification.
- If a function specified in a #pragma text becomes unnecessary due to a #pragma inline specification, or inline expansion via optimization options, the function will still be output to the specified section.
- If the name of the section being created was omitted, this specification will be allocated to the default text attribute section, so it will have not meaning, but if a named section had already been specified, then it will revert to the default.
- When specifying a section name, keep the length of the name to within 256 characters.

(4) Peripheral I/O register

Peripheral I/O registers are used to control the internal peripheral functions of a device. By using the peripheral I/O register name defined by the device, the internal I/O can be accessed at C language level. The peripheral I/O register names can be treated in the C source program as if they were normal unsigned external variables.
For the register names and attributes that can be specified, see the Relevant Device 's Hardware User' s Manual of each device.

(a) Accessing

A peripheral function register name is validated by describing the following #pragma directive.

```
#pragma ioreg
```

In a C source file in which "#pragma ioreg" directive is described, the peripheral function register name described after the #pragma directive can be used.
If this directive is not used or if a peripheral function register name is used without specifying an applicable device name, an "undefined variable" error occurs.
An error also occurs if the access attribute peculiar to the specified register is violated.
Of the examples as follows, Example 1 is correct, but Examples 2 and 3 cause an error.
P0, P1, P2, RXB0, and OVF0 in the following examples indicate the peripheral I/O registers of the V850 microcontrollers. In this way, symbols defined by the device file are specified as "register names".

Examples 1.

```
#pragma ioreg
void func1(void) {
        int i;
        P0 = 1;     /*Writes to P0*/
        i = RXB0;   /*Reads from RXB0*/
}

void func2(void) {
        P1 = 0;     /*Writes to P1*/
}
```

2.

```
void func(void) {
        P1 = 0;     /*Undefined error*/
}
```

3.

```
#pragma ioreg
void func(void) {
        RXB0 = 1;   /*Error that occurs if attribute of RXB0 is read-only*/
}
```

(5) Describing assembler instruction

With the CX, assembler instruction can be described in the functions of a C source program in the following format.

- asm declaration
- #pragma directives

To use registers with an inserted assembler, save or restore the contents of the registers in the program because they are not saved or restored by the CX.
Insert assembler instruction code inside a function. If the instructions are described outside a function,an error occurs. t

(a) asm declaration

```
__asm(character string constant);
```

<1> If the asm declaration is specified, the compiler suffixes a new-line character (\n) to the specified character string constant[Note] and passes it to the assembler.

Note The backslash ("\") is an escape character. (Example:\0->NULL, \r->Carriage return, \"->", \\->\)

Example

```
__asm("nop");
__asm (".str    \"string\\0\"");
```

- __asm is a declaration and is not treated as a statement. Therefore, because of the syntax of the C source, an error occurs in a structure that does not allow the use of a declaration only, as shown in Example 1 below.
  Therefor, enclose the statement in "{ }" as shown in Example 2 to make it a compound statement.

Examples 1.

```
if(i == 0)
__asm("mov     r11, r10");  /*Error occurs because only declaration is made.*/
```

2.

```
if(i == 0) {
        __asm("mov     r11, r10");  /*Can be used because this is compound
                                      statement.*/
}
```

(b) #pragma directives

In the range enclosed by the above #pragma directives, assembler instructions can be described as is. This is useful for using two or more assembler instructions.

```
#pragma asm
        assembler instruction
#pragma endasm
```

A description of example 1 to show next is same to a description of example 2.

Examples 1.

```
int i;

void f() {
#pragma asm
        mov     r0, r10
        st.w    r10, $_i
          :
#pragma endasm
}
```

2.

```
int i;

void f() {
        __asm("mov     r0, r10");
        __asm("st.w    r10, $_i");
          :
}
```

The description from "#pragma asm" to "#pragma endasm" is passed to the assembler as it is.
In other words, the CX internally creates an assembler instruction and starts the assembler.
Therefore, a directive of the assembler can be used after the #pragma asm declaration. A local variable in a C source must not be used with the assembler. Because the local variable is allocated to the stack or a register by the CX, it cannot be used with an inline assembler.
A symbol defined using #define in the C source file cannot be used in the description from "#pragma asm" to "#pragma endasm". Therefore expand a macro defined by #define in a file by an assembler instruction, as follows.

- Define the macro by using the .macro instruction in the #pragma asm - #pragma endasm directives.
- Call an assembler instruction from the C source program by means of a function call.

Another method is to write an assembler instruction without making a macro definition.

(6) Controlling interrupt level

(a) __set_il function

The CX can manipulate the interrupts of the V850 microcontrollers as follows in a C source.

- By controlling interrupt level
- By enabling or disabling acknowledgment of maskable interrupts (by masking interrupts)

In other words, the interrupt control register can be manipulated.
For this purpose, the "__ set_il" function is used. Specify this function as follows to manipulate the interrupt priority level.

```
__set_il(interrupt-priority-level, "interrupt-request-name");
```

Integer values 1 to 16 can be specified as the interrupt priority level. With devices with V850E2V3 instruction set architecture, sixteen steps, from 0 to 15, can be specified as the interrupt priority level. To set the interrupt priority level to "5", therefore, specify the interrupt priority level as "6" by this function.

Example

```
__set_il(2, "INTP0");
```

This specification sets the interrupt priority level of interrupt INTP0 to 1.
Specify the __ set_il function as follows to enable or disable acknowledgment of a maskable interrupt.

```
__set_il(enables/disables maskable interrupt, "interrupt request name");
```

"-1" or "0" can be specified to enable or disable the maskable interrupt.

Table 3-17. Enabling or Disabling Maskable Interrupt

Set Value	Operation
-1	Disables acknowledgment of maskable interrupt (masks interrupt).
0	Enables acknowledgement of maskable interrupt (unmasks interrupt).

Example

```
__set_il(-1, "INTP0");
```

If the function is specified as shown above, acknowledging maskable interrupt INTP0 is disabled (INTP0 is masked).
Note that the __ set_il function does not manipulate the EP flag (that indicates that exception processing is in progress) in the program status word (PSW).

(b) __set_il function and interrupt control register

If the __ set_il function is used, either "priority level" or "interrupt mask flag" is set. This means that the __ set_il function cannot set an interrupt request flag.

(7) Disabling interrupts

The CX can disable the maskable interrupts in a C source.
This can be done in the following two ways.

- Locally disabling interrupt in function
- Disabling interrupts in entire function

(a) Locally disabling interrupt in function

The "di instruction" and "ei instruction" of the assembler instruction can be used to disable an interrupt locally in a function described in C language. However, the CX has functions that can control the interrupts in a C.language source.

Table 3-18. Interrupt Control Function

Interrupt Control Function	Operation	Processing by CX
__DI	Disables the acceptance of all maskable interrupts.	Generates di instruction.
__EI	Enables the acceptance of all maskable interrupts.	Generates ei instruction.

Example How to use the __ DI() and __ EI() functions and the codes to be output are shown below.

```
[C source]
void func1(void) {
          :
        __DI();
        /*describe processing to be performed with interrupt disabled*/
        __EI();
          :
}
```

```
[Output codes]
_func1:
        -- prologue code
          :
        di
        -- processing to be performed with interrupt disabled
        ei
          :
        -- epilogue code
        jmp     [lp]
```

(b) Disabling interrupts in entire function

The CX has a "#pragma block_interrupt" directive that disables the interrupts of an entire function.
This directive is described as follows.

```
#pragma block_interrupt function-name
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1".
The interrupt to the function specified by "function-name" above is disabled. As explained in "(a) Locally disabling interrupt in function", __ DI()" can be described at the beginning of a function and "__ EI()", at the end. In this case, however, an interrupt to the prologue code and epilogue code output by the CX cannot be disabled or enabled, and therefore, interrupts in the entire function cannot be disabled.
Using the #pragma block_interrupt directive, interrupts are disabled immediately before execution of the prologue code, and enabled immediately after execution of the epilogue code. As a result, interrupts in the entire function can be disabled.

Example How to use the #pragma block_interrupt directive and the code that is output are shown below.

```
[C source]
#pragma block_interrupt func1
void func1(void) {
          :
        /*describe processing to be performed with interrupt disabled*/
          :
}
```

```
[Output codes]
_func1:
        di
        -- prologue code
          :
        -- processing to be performed with interrupt disabled
          :
        -- epilogue code
        ei
        jmp     [lp]
```

(c) Notes on disabling interrupts in entire function

Note the following points when disabling interrupts in an entire function.

- If an interrupt handler and a #pragma block_interrupt directive are specified for the same interrupt, the interrupt handler takes precedence, and the setting of disabling interrupts is ignored.
- If the following functions are called in a function in which an interrupt is disabled, the interrupt is enabled when execution has returned from the call.
- Function specified by #pragma block_interrupt.
- Function that disables interrupt at the beginning and enables interrupt at the end.
- Describe the #pragma block_interrupt directive before the function definition in the same file; otherwise an error occurs during compilation.
- However, the order of prototype declaration of a function is not affected.
- Neither #pragma inline nor inline expansion can be specified by an optimization option for the function specified by a #pragma block_interrupt directive. The inline expansion specification is ignored.
- A code that manipulates the ep flag (that indicates exception processing is in progress) in the program status word (PSW) is not output even if #pragma block_interrupt is specified.

(8) Interrupt/Exception processing handler

The CX can describe an "Interrupt handler" or "Exception handler" that is called if an interrupt or exception occurs. This section explains how to describe these handlers.

(a) Occurrence of interrupt/exception

If an interrupt or exception occurs in the V850 microcontrollers, the program jumps to a handler address corresponding to the interrupt or exception. An interrupt source and a handler address correspond one by one. A collection of handler addresses is called an interrupt/exception table.

For example, the interrupt/exception table of the V850E2/MN4 is as shown below (only the part is shown).

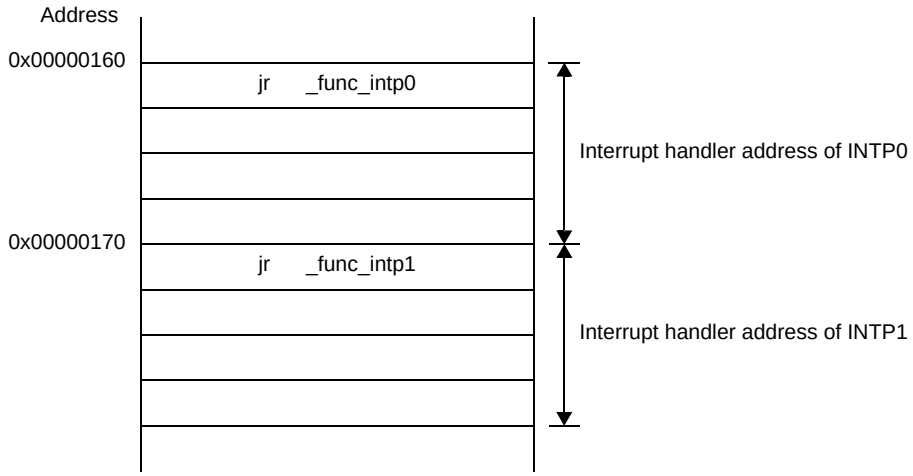
Table 3-19. Interrupt/Exception Table (V850E2/MN4)

Address	Interrupt Name	Interrupt Trigger
0x0000	RESET	RESET input
0x0010	FEINT	External NMI input
0x0020	FENMI	WDT0ATNMI/WDT1ATNMI
:	:	:
0x0120	INTWDTA0	WDT0 Interval timer interrupt
0x0130	INTWDTA1	WDT1 Interval timer interrupt

Address	Interrupt Name	Interrupt Trigger
0x0140	INTOSTM0	OS timer underflow interrupt
0x0150	INTOSTM1	OS timer underflow interrupt
0x0160	INTP0	External Interrupt
0x0170	INTP1	External Interrupt
0x0180	INTP2	External Interrupt

The arrangement of the handler addresses and the available interrupts vary depending on the device of the V850. See the Relevant Device 's User' s Manual of each device for details.
Each handler address has a 16-byte area. If an interrupt occurs, an instruction written in that 16-byte area is executed. This means that, if the processing code does not exceed 16 bytes, it is performed only in the handler address. If not, an instruction that branches to a function (interrupt handler) where the processing is written is described.

Figure 3-15. Image of Interrupt Handler Address



If the INTP0 interrupt occurs in the V850E2/MN4, the program jumps to address 0x160 and executes the code written at that address. In this example, the program jumps to the func_intp0 function because a code that branches to that function is written. This means that func_intp0 is the interrupt handler of INTP0.

The above description is at an assembler source level. With the CX, users do not have to pay much attention to this when describing interrupt servicing in C language source. How to describe interrupt servicing is explained specifically in "(c) Describing interrupt/exception handler".

(b) Processing necessary in case of interrupt/exception

If an interrupt/exception occurs while a function or a task is being executed, interrupt/exception processing must be immediately executed. When the interrupt/exception processing is completed, execution must return to the function or task that was interrupted [Note].
Therefore, the register information at that time must be saved when an interrupt/exception occurs, and the register information must be restored when interrupt/exception processing is complete.

Note When a real-time OS is used, execution may not return to a task that is interrupted if a system call is issued during interrupt servicing. See the User's Manual of each real-time OS for details.

The prologue and epilogue codes of an ordinary function save and restore the registers for register variables. The registers for register variables are shown below. Those that must be saved and restored are saved and restored.

Table 3-20. Registers for Register Variables

Register Modes	Register Variable Registers
22-register mode	r25, r26, r27, r28, r29
26 -register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

When execution shifts to an interrupt/exception handler, the following registers that must be saved, in addition to the registers shown in the above table, are also saved as a stack frame for the interrupt/exception handler.

Table 3-21. Stack Frame for Interrupt/Exception Handler

Register Modes	Registers Saved/Restored in Case of Interrupt/Exception
22-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r31 (lp), CTPC, CTPSW, BSEL [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]
26-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r31 (lp), CTPC, CTPSW, BSEL [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]
32-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r31 (lp), CTPC, CTPSW, BSEL [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]

When multiple interrupt/exception occurs, the following registers that must be saved, in addition to the registers for register variables, are also saved as a stack frame for the multiple interrupt/exception handler.

Table 3-22. Stack Frame for Multiple Interrupt/Exception Handler

Register Modes	Registers Saved/Restored in Case of Multiple Interrupts/Exceptions
22-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r31 (lp), EIPC, EIPSW, CTPC, CTPSW, BSEL [V850E2V3], EIIC [V850E2V3], EIWR [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]
26-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r31 (lp), EIPC, EIPSW, CTPC, CTPSW, BSEL [V850E2V3], EIIC [V850E2V3], EIWR [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]
32-register mode	r1, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r31 (lp), EIPC, EIPSW, CTPC, CTPSW, BSEL [V850E2V3], EIIC [V850E2V3], EIWR [V850E2V3], FPSR/FPEPC(with FPU) [V850E2V3]

The usage of the above registers is as follows.

Table 3-23. Usage of Registers

Register	Usage
r1	Assembler-reserved register
r6-r9	Registers for arguments (registers to set arguments of function)

Register	Usage
r10-r19	Work registers (registers used by the CX to generate codes)
r31	Link pointer
CTPC	Program counter (PC) when CALLT instruction is executed.
CTPSW	Program status word (PSW) when CALLT instruction is executed.
EIPC	Program counter (PC) during interrupt/exception processing
EIPSW	Program status word (PSW) during EIPSW interrupt/exception processing.
BSEL [V850E2V3]	Register bank selection register
EIIC [V850E2V3]	Register that stores the EI level exception cause
EIWR [V850E2V3]	EI level exception working register
FPSR [V850E2V3]	Floating-point operation setting/status storage register
FPEPC [V850E2V3]	Floating-point operation exception program counter

When interrupt/exception processing is completed, the code which restores saved registers is output, the eiret instruction is output. This instruction notifies the V850 that the interrupt/exception servicing is completed.
If codes for saving/restoring registers or outputting the reti instruction are described as explained in "(c) Describing interrupt/exception handler", the CX automatically outputs the relevant code. The code for saving/restoring registers is output as explained in "Table 3-24. Processing for Saving/Restoring Registers During Interrupt". The user therefore does not have to pay much attention to this and can concentrate on describing the processing of the main body of the interrupt handler.

Table 3-24. Processing for Saving/Restoring Registers During Interrupt

Register Name		Register	Explanation
Assembler-reserved register		r1	Always saved/restored at interrupt.
Argument registers		r6 to r9	r6 is always saved/restored when the interrupt source is TRAP0/ TRAP1. Saved/restored when a function call (including runtime functions) exists. Saved/restored if a function call does not exist but is used with an interrupt function.
Work Registers	22-register mode	r10 to r14	Saved/restored when a function call exists. Saved/restored if a function call does not exist but is used with an interrupt function.
	26-register mode	r10 to r16	
	32-register mode	r10 to r19	
Register Variable Registers	22-register mode	r25 to r29	Saved/restored as necessary, as with ordinary functions.
	26-register mode	r23 to r29	
	32-register mode	r20 to r29	
Link pointer		r31(lp)	Saved/restored when a function call (including runtime functions) exists Does not save/restore if a function call does not exist.
Interrupt-related system registers		EIPC, EIPSW	Always saved/restored with functions using the multiple interrupt (The interrupt function with multi option). Not saved/restored without the multi option.

Register Name	Register	Explanation
callt instruction-related system registers	CTPC, CTPSW	Always saved/restored with interrupt functions without the nopush option. Not saved/restored with the nopush option.
Register bank selection register [V850E2V3]	BSEL	Always saved/restored with interrupt functions being compiled with a device with an instruction set of V850E2V3 specified.
Register that stores the EI level exception cause [V850E2V3]	EIIC	Always saved/restored with multiple interrupt functions being compiled with a device with an instruction set of V850E2V3 specified.
EI level exception working register [V850E2V3]	EIWR	Always saved/restored with multiple interrupt functions being compiled with a device with an instruction set of V850E2V3 specified.
Floating-point operation setting/status storage register [V850E2V3]	FPSR	Always saved/restored with interrupt functions being compiled with a device with an instruction set of V850E2V3 with FPU device specified.
Floating-point operation exception program counter [V850E2V3]】	FPEPC	Always saved/restored with interrupt functions being compiled with a device with an instruction set of V850E2V3 with FPU device specified.

(c) Describing interrupt/exception handler

The format in which an interrupt/exception handler is described does not differ from ordinary C functions, but the functions described in C must be recognized as an interrupt/exception handler by the CX. With the CX, an interrupt/exception handler is specified using the #pragma interrupt directive.

```
#pragma interrupt Interrupt-request-name Function-name [Allocation-method]   [Option [Option]...]
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1".

- *Inerrupt request name*

  Interrupt request names registered in the device file can be specified. Refer to the interrupt request names described in the Relevant Device 's Architecture User' s Manual of each device; they are the interrupt request names registered in the device file.

  A non-maskable interrupt (NMI) can also be specified in this way, but a reset interrupt (RESET) cannot be specified. Processing after reset must be described with assembler instructions. Processing after reset is generally described in the startup routine, so see "CHAPTER 7 STARTUP" for details.

  If the interrupt request name is set to "NO_VECT", then it will not be set in the interrupt handler address, and the function will only be output as an interrupt function.

- *Function Name*

  Specify the names of functions that are used as an Interrupt/Exception handler. Describe the function name in C source. When specifying the function "void func1(void)", specify the function name as "func1".

- *Allocation method*

Specify whether the main body of the function is directly allocated to the handler address, or only the instruction that branches to the interrupt handler function is allocated. Specify "direct" when the main body of the function is directly allocated; otherwise describe nothing as "allocation method". By specifying "direct", all functions are allocated from the handler address of the specified interrupt source. Note, however, that the areas for the subsequent handler address are also used.

When specifying "direct", be sure to describe the #pragma interrupt directive before the function definition; otherwise an error occurs during compilation.

If the interrupt request name is set to "NO_VECT", then "direct" cannot be specified (it will cause an error).

- *Option*

The following options can be specified.

multi	Use a multi interrupt handler. Output EIPC/EIPSW save/restore code. Code to enable interrupts is also output, so there is no need to enable interrupts via __EI(). This disables interrupts when terminating a function. Perform the synce instruction immediately prior to disabling.
nopush	Do not output CTPC/CTPSW save/restore code. This option can reduce the code size, if you are using single interrupts and the function call doesn't exist in the interruption function.
push_ei	Output EIPC/EIPSW save code.
nopush_fpu	Do not output FPSR/FPEPC save code.

The multi interrupt handler specification specifies a function that enables multiple interrupts. It does not specify a function that makes multiple interrupts.

Next, the function type that can be specified as an interrupt handler is explained.

- Function type

The type of a handler that handles a maskable interrupt or NMI is as follows.

void func(void) type

The argument and return value of this function are void type.

The type of a software exception processing (trap) handler is as follows.

void func(unsigned int) type

The exception cause code for the EI level exception cause register (EIIC) is set in the parameter. Unless the function is specified by this type, an error occurs during compilation. Refer to the next paragraph for the software exception processing function.

- Software exception processing (trap processing) handler

When software exception processing (trap processing) is used, two entry points, TRAP0 (address 0x40) and TRAP1 (address 0x50), are used according to the specifications of the V850 microcontrollers. When the software exception "trap 0x00 to trap 0x0F" occurs, execution branches to TRAP0 (address 0x40); if "trap 0x10 to trap0x1F" occurs, it branches to TRAP1 (address 0x50). At this time, the value "0x40 to 0x4F" is set to the interrupt source register (ECR) as a software exception code in the case of TRAP0. In the case of TRAP1, the value "0x50 to 0x5F" is set to the EIIC.

Table 3-25. Trap Instructions and Software Exception Codes

Trap Instruction	Software Exception Code
`trap 0x00`	`0x40`
`trap 0x01`	`0x41`

Trap Instruction	Software Exception Code
trap 0x02	0x42
:	:
trap 0x0A	0x4A
trap 0x0B	0x4B
:	:
trap 0x10	0x50
trap 0x11	0x51
trap 0x12	0x52
:	:
trap 0x1E	0x5E
trap 0x1F	0x5F

When software exception processing for TRAP0 or TRAP1 is described, that function has one argument and the type of the variable is "unsigned int". The software exception code set to the EI level exception cause register (EIIC) is set as the argument. In the case of TRAP0, the value is "0x40 to 0x4F". In the case of TRAP1, it is "0x50 to 0x5F". Processing must be described in the handler depending on these values.

```
#pragma interrupt   TRAP0   trap0_func
void trap0_func(unsigned int codenum) {
          :
        describe processing of each exception code
          :
}
```

(d) Notes on describing interrupt/exception handler

- "Specifying multiple-interrupt handler" with the multi option means to "specify a function that can be interrupted more than once" and does not mean "to specify a function that interrupts more than once".
- The reset interrupt cannot be specified by the #pragma interrupt directive.

```
#pragma interrupt   RESET   reset_func  /*error*/
```

If the above description is made, an error occurs during compilation. Processing after reset must be described with assembler instructions.

Processing after reset is generally described in the startup routine, so see "CHAPTER 7 STARTUP" for details.

- Specify multi option in the function specified as a handler that processes multiple interruptions. In such case, code which saves, restores the EIPC and EIPSW is output. Interrupt handler where multi option is not specified, the code which saves, restores the EIPC and EIPSW is not output.
- The #pragma interrupt directive do not support multiple exceptions and multiple NMIs. To use multiple exceptions or multiple NMI, add a code that saves or restores the necessary system registers (such as FEPC and FEPSW). See the Relevant Device's User's Manual of each device for the necessary system registers.
- The user is not required to additionally describe an interrupt handler address in the link directive file. It is output internally by the CX.
- The same interrupt request name must not be specified for two or more functions.

- A function specified as an interrupt/exception handler cannot be expanded inline. The #pragma inline directive is ignored even if specified.
- An interrupt to a function specified as an interrupt/exception handler is disabled. Therefore, the #pragma block_interrupt directive is ignored even if specified.
- A function specified as an interrupt/exception handler cannot be called by an ordinary function call. If it is called from another file, the compiler cannot check it.
- When an assembler instruction is called from an interrupt/exception handler and the registers shown in "Table 3-20. Registers for Register Variables" and "Table 3-21. Stack Frame for Interrupt/Exception Handler" are used, processing to save/restore the register contents must be described. Processing to save/restore the register contents must also be described when sp (r3), gp (r4), tp (r5), and ep (r30) are rewritten.
- The #pragma interrupt directive do not issue a processing end report (EOI command) to the external interrupt controller. The user should therefore execute this directive, if necessary.
- Disable interrupts at the end of multiple interrupts because a code that restores EIPC and EIPSW must be described.
- If "direct" is not specified, an instruction to branch to the interrupt/exception handler is allocated to the handler address. In this case, the CX outputs the jr instruction to enhance the code efficiency. However, the range in which the jr instruction can branch execution is limited to ±21 bits from the jr instruction. If the main body of the interrupt handler is not within the range in which the jr instruction can branch execution, an error occurs during linking. In this case, specify the compilation option "-Xfar_jump" to replace the jr instruction with the jmp instruction.
- The FE level interrupt is not supported.
- If the "multi" option is specified, then code to save EIPC/EIPSW will be output due to the device specifications, regardless of whether there is a "push_ei". An error will not be output.
- "nopush_fpu" has no meaning on devices without an FPU, and will be assumed to have been specified implicitly. Even if it is not specified, code to save FPSR/FPEPC will not be output (devices without an FPU do not have FPSR/FPEPC).

(e) Description example of interrupt/exception handler

Examples of describing interrupt/exception handlers are shown below.
Note that the interrupt request name differs depending on the device. See the Relevant Device 's User' s Manual of each device.

Examples 1. Non-maskable interrupt

```
#pragma interrupt   NMI     func1   /*non-maskable interrupt*/
void func1(void) {
  :
}
```

2. Trap

```
#pragma interrupt   TRAP0   func2   /*Trap*/
void func2(unsigned int num) {
        switch(num) {   /*for every exception cod*/
          :
        }
}
```

3. Multiple interrupts

```
#pragma interrupt   INTP0   func1   /*multiple-interrupt*/
void func1(void) {
  :
}
```

(9) Inline expansion

The CX allows inline expansion of each function. This section explains how to specify inline expansion.

(a) Inline Expansion

Inline expansion is used to expand the main body of a function at a location where the function is called. This decreases the overhead of function call and increases the possibility of optimization. As a result, the execution speed can be increased.

If inline expansion is executed, however, the object size increases.

Specify the function to be expanded inline using the #pragma inline directive.

```
#pragma inline  function-name[, function-name, ...]
```

Describe functions that are described in the C language. In the case of a function, "void func1() {}", specify "func1". Two or more function names can be specified with each delimited by "," (comma).

```
#pragma inline  func1, func2
void    func1() {...}
void    func2() {...}
void func(void) {
        func1();    /*function subject to inline expansion*/
        func2();    /*function subject to inline expansion*/
}
```

(b) Conditions of inline expansion

At least the following conditions must be satisfied for inline expansion of a function specified using the #pragma inline directive.

Inline expansion may not be executed even if the following conditions are satisfied, because of the internal processing of the CX.

<1> A function that expands inline and a function that is expanded inline are described in the same file

A function that expands inline and a function that is expanded inline, i.e., a function call and a function definition must be in the same file. This means that a function described in another C source cannot be expanded inline. If it is specified that a function described in another C source is expanded inline, the CX does not output a warning message and ignores the specification.

<2> The #pragma inline directive is described before function definition.

If the #pragma inline directive is described after function definition, the CX outputs a warning message and ignores the specification. However, prototype declaration of the function may be described in any order. Here is an example.

Example

```
[Valid Inline Expansion Specification]
#pragma inline  func1, func2
void    func1();        /*prototype declaration*/
void    func2();        /*prototype declaration*/
void    func1() {...}   /*function definition*/
void    func2() {...}   /*function definition*/
```

```
[Invalid Inline Expansion Specification]
void    func1();        /*prototype declaration*/
void    func2();        /*prototype declaration*/
void    func1() {...}   /*function definition*/
void    func1() {...}   /*function definition*/
#pragma inline  func1, func2
```

<3> The number of arguments is the same between "call" and "definition" of the function to be expanded inline.

If the number of arguments is different between "call" and "definition" of the function to be expanded inline, the CX ignores the specification.

<4> The types of return value and argument are the same between "call" and "definition" of the function to be expanded inline.

If the number of arguments is different between "call" and "definition" of the function to be expanded inline, the CX ignores the specification. If the type of the argument is the integer type (including enum) or pointer-type, and in the same size, however, inline expansion is executed.

<5> The number of arguments of the function to be expanded inline is not variable.

If inline expansion is specified for a function with a variable arguments, the CX outputs neither an error nor warning message and ignores the specification.

<6> Recursive function is not specified to be expanded inline.

If a recursive function that calls itself is specified for inline expansion, the CX outputs neither an error nor warning message and ignores the specification. If two or more function calls are nested and if a code that calls itself exists, however, inline expansion may be executed.

<7> An interrupt handler is not specified to be expanded inline.

A function specified by the #pragma interrupt is recognized as an interrupt handler. If inline expansion is specified for this function, the CX outputs a warning message and ignores the specification.

<8> A task of a real-time OS is not specified to be expanded inline.

A function specified by the #pragma rtos_task directive is recognized as a task of a real-time OS. If inline expansion is specified for this function, the CX outputs a warning message and ignores the specification.

<9> Interrupts are not disabled in a function by the #pragma block_interrupt directive.

#If inline expansion is specified for a function in which interrupts are declared by the #pragma block_interrupt directive to be disabled, the CX outputs a warning message and ignores the specification.

(c) Execution speed priority optimization and inline expansion

If the "execution speed priority optimization (-Ospeed)" option of the CX is specified, the CX uses inline expansion as one of the means of optimization.

If the -Ospeed option is specified, the CX selects an appropriate function and expands it inline as long as the inline expansion conditions in "(b) Conditions of inline expansion" are satisfied, even if the function is not specified for inline expansion by the #pragma inline directive.

(d) Examples of differences in inline expansion operation depending on option specification

Here are examples of differences in inline expansion operation depending on whether the #pragma inline directive or an option is specified.

- When the -Osize (size priority optimization) option is specified (other than -Ospeed)

```
#pragma inline  func0
void    func0() {...}    /*expanded if inline expansion conditions are satisfied because,
                           #pragma inline is specified*/
void    func1() {...}    /*Not expanded*/
void    func2() {...}    /*Not expanded*/
```

- When the -Ospeed (execution speed priority optimization) option is specified

```
#pragma inline  func0
void    func0() {...}   /*expanded if inline expansion conditions are satisfied
                          because -Ospeed is specified*/
void    func1() {...}   /*expanded if inline expansion conditions are satisfied
                          because -Ospeed is specified*/
void    func2() {...}   /*expanded if inline expansion conditions are satisfied
                          because -Ospeed is specified*/
```

Remarks 1. The CX does not treat a function specified for inline expansion by the #pragma inline directive as a static function. To use such a function as a static function, static must be explicitly specified.

2. When executing debugging, a breakpoint cannot be specified for a function specified for inline expansion in the C source.

(10) Real-time OS support function

The CX has functions to improve programming description and to reduce the number of codes, making allowances for organizing a system using the V850 microcontrollers real-time OS RX850V4.

(a) Description of task

An application using a real-time OS performs processing in task units. The real-time OS schedules a task using a system call issued in that task or interrupt servicing. Register contents are saved and restored by the real-time OS when the task is switched (when the context is switched). Therefore, prologue and epilogue processing are different from those of an ordinary function.

In other words, the prologue and epilogue processing generated by the CX when a function is called are not executed by a task.

To use a function described as a task, the code can be reduced by deleting the prologue and epilogue processing that are executed when a function is called. However, ordinary functions and tasks are not distin-

guished according to the description method of C language Therefore, the CX has the following #pragma directive so that a function can be recognized as a task of a real-time OS.

```
#pragma rtos_task   [function-name]
```

Consequently, the function specified by "function-name" can be recognized as a task of a real-time OS. A function name described in C is specified as "function-name". The following description is made, for example, to use the function "void func1(int inicode){}" as a task.

Example

```
#pragma rtos_task   func1
```

Specifying the #pragma rtos_task directive has the following effect.

<1> The prologue/epilogue processing output by an ordinary function is not performed. Specifically, the following codes are not output.

- Saving/restoring of register contents for register variables
- Saving/restoring of link pointer (lp)
- Jump to return address

<2> The system call "ext_tsk" can be used as a defined function.

This system call can be used even if a prototype declaration is not made in the application. Functions other than the one specified as a task can be called in the same manner as long as they are described after the #pragma rtos_task directive.

When this system call is called, a code using the jr instruction is output to reduce the code size. If the main body of system call "ext_tsk" is not in the range in which the jr instruction can branch execution, the linker outputs an error. In this case, take the following actions

- Change the memory allocation by the link directive
- Replace the jr instruction with the jmp instruction in the assembler source
- Specify far jump

Note the following points when the #pragma rtos_task directive is specified.

- A task cannot be called in the same manner as calling a function. A task called from a separate file is not checked. A task cannot be expanded inline because it cannot be called as a function. That is, even if the #pragma inline directive is specified for a function specified by the #pragma rtos_task directive, the #pragma inline specification is ignored.
- An error occurs if "#pragma rtos_task function-name" is described after the function definition in the same file.
- If the function is not defined after "#pragma rtos_task function-name" is described in the file, the #pragma directive for that function is ignored. Note, however that "#pragma rtos_task" code is valid, and it is possible to use the ext_tsk() system call in functions called after that.
- A function specified by the #pragma rtos_task directive cannot be specified as an ordinary interrupt/exception handler (see "(8) Interrupt/Exception processing handler").

See the User's Manual of each real-time OS for the real-time OS functions.

(11) Embedded functions

In the CX, some of the assembler instructions can be described in C source as "Embedded Functions". However, it is not described "as assembler instruction", but as a function format set in the CX. When these functions are used, output code outputs the compatible assembler instructions without calling the ordinary function.

If a parameter is specified whose type cannot be implicitly converted to that of the parameter of the embedded function, then an warning is output, and it is treated as an ordinary function. A warning is also output if a register number that does not exist in the hardware is specified for ldsr()/stsr()/ldgr()/stgr(), and it will be treated as an ordinary function.

The instructions that can be described as functions are as follows.

Table 3-26. Embedded Functions

Assembler Instruction	Function	Embedded Function
di	Interrupt control	`__DI();`
ei		`__EI();`
nop	No operation	`__nop();`
halt	Stops the processor	`__halt();`
satadd	Saturated addition	`long a, b;` `long __satadd(a, b);`
satsub	Saturated subtraction	`long a, b;` `long __satsub(a, b);`
bsh	Halfword data byte swap	`long a;` `long __bsh(a);`
bsw	Word data byte swap	`long a;` `long __bsw(a);`
hsw	Word data halfword swap	`long a;` `long __hsw(a);`
sxb	Byte data sign extension	`char a;` `long __sxb(a);`
sxh	Halfword data sign extension	`short a;` `long __sxh(a);`
mul	Instruction that applies result of 32-bit x 32-bit signed multiplication to variable using mul instruction	`long a, b;` `long long __mul(a, b);`
mulu	Instruction that applies result of 32-bit x 32-bit signed multiplication to variable using mulu instruction	`unsigned long a, b;` `Unsigned long long __mulu(a, b);`
mul32	Instruction that assigns higher 32 bits of multiplication result to variable using mul32 instruction	`long a, b;` `long __mul32(a, b);`
mul32u	Instruction that assigns higher 32 bits of unsigned multiplication result to variable using mul32u instruction	`unsigned long a, b;` `unsigned long __mul32u(a, b);`

sasf	Flag condition setting with logical left shift	`long a;`
		`unsigned int b;`
		`long __sasf(a, b);`

Assembler Instruction	Function	Embedded Function
sch0l	Bit (0) search from MSB side [V850E2V3]	long a; long __sch0l(a);
sch0r	Bit (0) search from LSB side [V850E2V3]	long a; long __sch0r(a);
sch1l	Bit (1) search from MSB side [V850E2V3]	long a; long __sch1l(a);
sch1r	Bit (1) search from LSB side [V850E2V3]	long a; long __sch1r(a);
ldsr	Loads to system register [V850E2V3]	long a; void __ldsr(regID[Note], a);
stsr	Stores contents of system register [V850E2V3]	unsigned long __stsr(regID[Note]);
ldgr	Loads to general-purpose register [V850E2V3]	long a; void __ldgr(regID[Note], a);
stgr	Stores contents of general-purpose register [V850E2V3]	unsigned long __stgr(regID[Note]);
caxi	Compare and Exchange [V850E2V3]	long *a; long b, c; void __caxi(a, b, c);

Note Specified the system register number (0 to 31) in regID.
But, don't specify 0 as regID in ldsr.

Caution Even if a function is defined with the same name as an embedded function, it cannot be used. If an att isempt made to call such a function, processing for the embedded function provided by the compiler takes precedence.

(12) Structure type packing

In the CX, the alignment of structure members can be specified at the C language level. This function is equivalent to the -Xpack option, however, the structure type packing directive can be used to specify the alignment value in any location in the C source.

Caution The data area can be reduced by packing a structure type, but the program size increases and the execution speed is degraded.

(a) Format of structure type packing

The structure type packing function is specified in the following format.

```
#pragma pack([1248])
```

#pragma pack changes to an alignment value of the structure member upon the occurrence of this directive. The numeric value is called the packing value and the specifiable numeric values are 1, 2, 4, and 8. Specification of the packing value cannot be omitted. If there is no packing value, the CX outputs the following message.

E0521605: Illegal #pragma *character string* syntax.

Since this directive becomes valid upon occurrence, several directives can be described in the C source.

Example

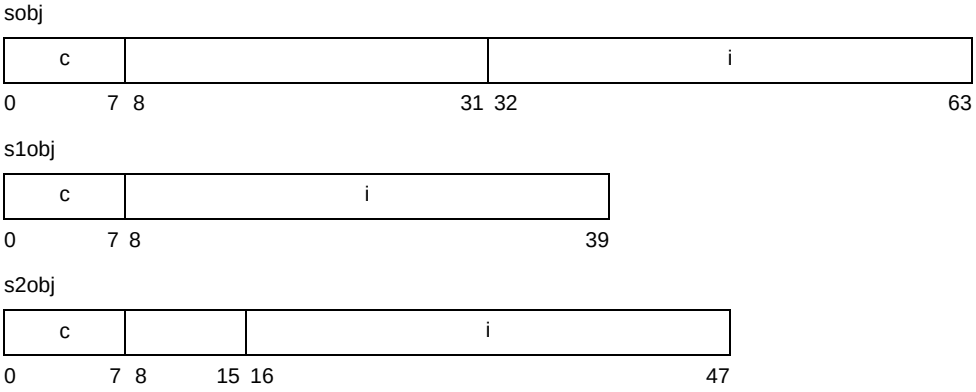
```
#pragma pack(1) /*Structure member aligned using 1-byte alignment*/
struct  TAG {
        char    c;
        int     i;
        short   s;
};
```

(b) Rules of structure type packing

The structure members are aligned in a form that satisfies the condition whereby members are aligned according to whichever is the smaller value: the structure type packing value or the member's alignment value.
For example, if the structure type packing value is 2 and member type is int type, the structure members are aligned in 2-byte alignment.

Example

```
struct  S {
        char    c;  /*Satisfies 1-byte alignment condition*/
        int     i;  /*Satisfies 4-byte alignment condition*/
};

#pragma pack(1)
struct  S1 {
        char    c;  /*Satisfies 1-byte alignment condition*/
        int     i;  /*Satisfies 1-byte alignment condition*/
};

#pragma pack(2)
struct  S2 {
        char    c;  /*Satisfies 1-byte alignment condition*/
        int     i;  /*Satisfies 2-byte alignment condition*/
};

struct  S   sobj;   /*Size of 8 bytes*/
struct  S1  s1obj;  /*Size of 5 bytes*/
struct  S2  s2obj;  /*Size of 6 bytes*/
```



(c) Union

A union is treated as subject to packing and is handled in the same manner as structure type packing.

Examples 1.

```
union   U {
        struct  S {
                char    c;
                int     i;
        } sobj;
};

#pragma pack(1)
union   U1 {
        struct  S1 {
                char    c;
                int     i;
        } s1obj;
};

#pragma pack(2)
union   U2 {
        struct  S2 {
                char    c;
                int     i;
        } s2obj;
};

union   U   uobj;   /*Size of 8 bytes*/
union   U1  u1obj;  /*Size of 5 bytes*/
union   U2  u2obj;  /*Size of 6 bytes*/
```

2.

```
union   U {
        int i:7;
};

#pragma pack(1)
union   U1 {
        int i:7;
};

#pragma pack(2)
union   U2 {
        int i:7;
};

union   U   uobj;   /*Size of 4 bytes*/
union   U1  u1obj;  /*Size of 1 byte*/
union   U2  u2obj;  /*Size of 2 bytes*/
```

(d) Bit field

Data is allocated to the area of the bit field element as follows.

<1> When the structure type packing value is equal to or larger than the alignment condition value of the member type

Data is allocated in the same manner as when the structure type packing function is not used. That is, if the data is allocated consecutively and the resulting area exceeds the boundary that satisfies the alignment condition of the element type, data is allocated from the area satisfying the alignment condition.

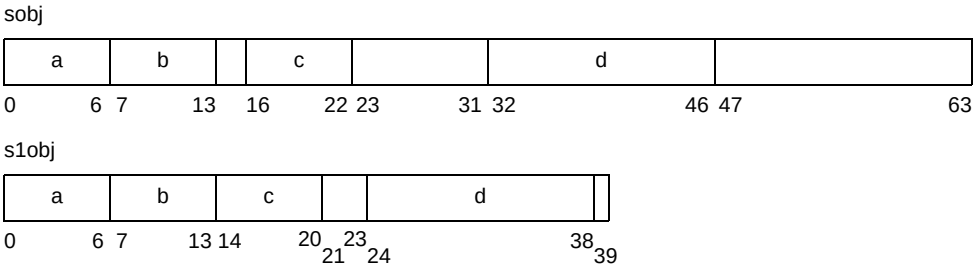
<2> When the structure type packing value is smaller than the alignment condition value of the element type

- If data is allocated consecutively and results in the number of bytes including the area becoming larger than the element type
  The data is allocated in a form that satisfies the alignment condition of the structure type packing value.
- Other conditions
  The data is allocated consecutively.

Example

```
struct  S {
        short   a:7;    /*0 to 6th bit*/
        short   b:7;    /*7 to 13th bit*/
        short   c:7;    /*16 to 22nd bit (aligned to 2-byte boundary)*/
        short   d:15;   /*32 to 46th bit (aligned to 2-byte boundary)*/
} sobj;
```

```
#pragma pack(1)
struct  S1 {
        short   a:7;    /*0 to 6th bit*/
        short   b:7;    /*7 to 13th bit*/
        short   c:7;    /*14 to 20th bit*/
        short   d:15;   /*24 to 38th bit  (aligned to byte boundary)*/
} s1obj;
```



(e) Alignment condition of top structure object

The alignment condition of the top structure object is the same as when the structure packing function is not used.

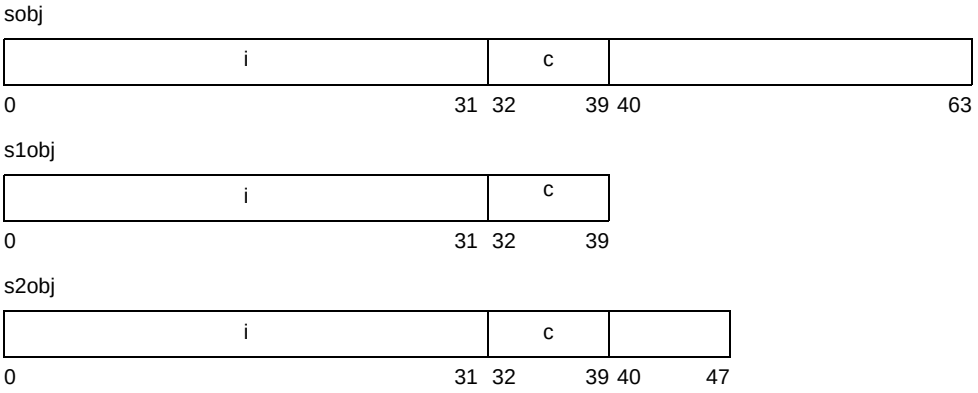
(f) Size of structure objects

Perform packing so that the size of structure objects becomes a multiple value of whichever is the smaller value: the structure alignment condition value or the structure packing value. The alignment condition of the top structure object is the same as when the structure packing function is not used.

Examples 1.

```
struct  S {
        int     i;
        char    c;
};

#pragma pack(1)
struct  S1 {
        int     i;
        char    c;
};

#pragma pack(2)
struct  S2 {
        int     i;
        char    c;
};

struct  S   sobj;   /*Size of 8 bytes*/
```

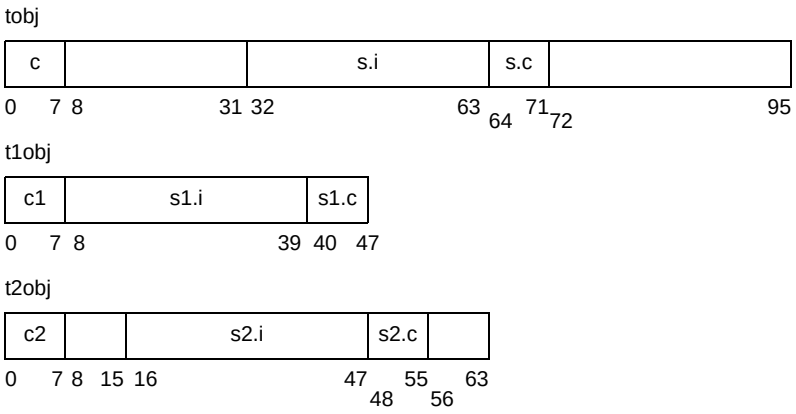
```
struct  S1  s1obj;  /*Size of 5 bytes*/
struct  S2  s2obj;  /*Size of 6 bytes*/
```



2.

```
struct  S {
        int     i;
        char    c;
};
struct  T {
        char    c;
        struct  S   s;
};

#pragma pack(1)
struct  S1 {
        int     i;
        char    c;
};
struct  T1 {
        char    c;
        struct  S1  s1;
};

#pragma pack(2)
struct  S2 {
        int     i;
        char    c;
};
struct  T2 {
        char    c;
        struct  S2  s2;
};
```

```
struct  T   tobj;   /*Size of 12 bytes*/
struct  T1  t1obj;  /*Size of 6 bytes*/
struct  T2  t2obj;  /*Size of 8 bytes*/
```

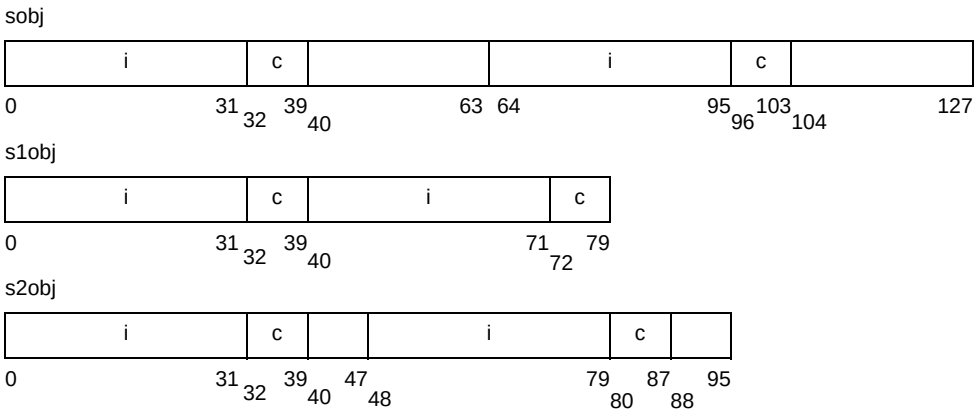


(g) Size of structure array

The size of the structure object array is a value that is the sum of the number of elements multiplied to the size of structure object.

Example

```
struct  S {
        int     i;
        char    c;
};

#pragma pack(1)
struct  S1 {
        int     i;
        char    c;
};

#pragma pack(2)
struct  S2 {
        int     i;
        char    c;
};

struct  S   sobj[2];    /*Size of 16 bytes*/
struct  S1  s1obj[2];   /*Size of 10 bytes*/
struct  S2  s2obj[2];   /*Size of 12 bytes*/
```

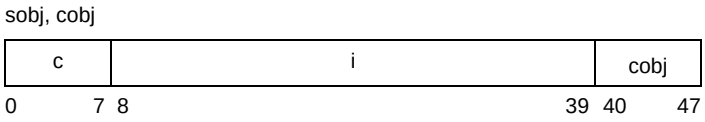


(h) Area between objects

For example, sobj.c, sobj.i, and cobj may be allocated consecutively without a gap in the following source program (the allocation order of sobj and cobj is not guaranteed).

Example

```
#pragma pack(1)
struct  S {
        char    c;
        int     i;
} sobj;
char    cobj;
```



(i) Notes concerning structure packing function

<1> Specification of the -Xpack option and #pragma pack directive at the same time

If the -Xpack option is specified when structure packing is specified with the #pragma pack directive in the C source, the specified option value is applied to all the structures until the first #pragma pack directive appears. After this, the value of the #pragma pack directive is applied.

Even after the #pragma pack directive appears, however, the specified option value is applied to the area specified by default.

Example When -Xpack=2 is specified

```
struct  S2 {...};   /*Packing value is specified as 2 in option
                      Option -Xpack = 2 is valid: packing value is 2*/
#pragma pack(1)     /*Packing is specified as 1 in #pragma directive
struct  S1 {...};     pragma pack(1) is valid: packing value is 1*/
#pragma pack()      /*Packing value is specified by default in #pragma directive
struct  S2_2 {...};   Option -Xpack = 2 is valid: packing value is 2*/
```

<2> Restrictions

When using the V850 microcontrollers and a CPU that is set to disable misalign access for V850Ex products, the following restrictions apply.

- Access using the structure member address cannot be executed correctly.
  As shown in the following example, the structure member address is acquired, and the access to that address is then performed with the address masked in accordance with the data alignment of the device. Therefore, some data may disappear or be rounded off.

Example

```
struct  test {
        char    c;      /*offset 0*/
        int     i;      /*offset 1-4*/
} test;
int *ip, i;

void func(void) {
        i = *ip;        /*Accessed with address masked*/
}

void func2(void) {
        ip = &(test.i); /*Acquire structure member address*/
}
```

- In bit field access, an area with no data to be read using the member 's type is also accessed.
  If the width of the bit field is smaller than the member' s type as shown in the following example, access occurs outside the object because reading is performed using the member' s type. Generally, there is no problem with the function, but if I/O are mapped, an illegal access may occur.

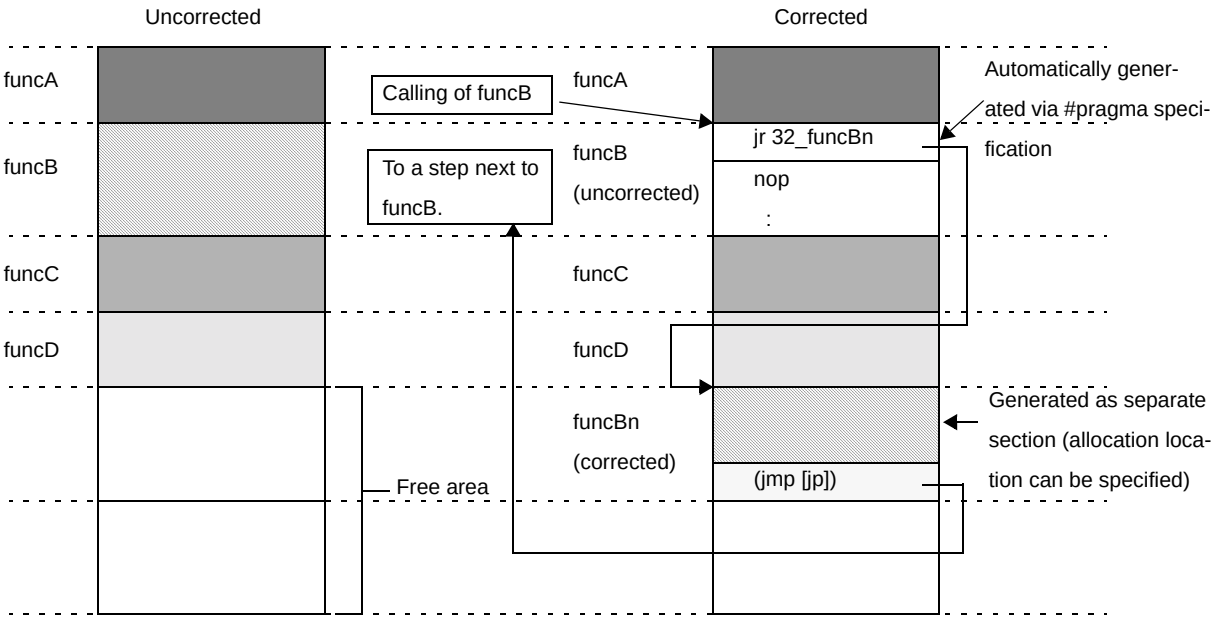
Example

```
struct  S {
        int x:21;
} sobj;  /*3 bytes*/
sobj.x = 1;
```

(13) Smart correction function

The smart correction feature enables you to correct a specific function without changing the other functions at all (code or addresses), by replacing the execution of that function only.
Specifically, the body of the function is replaced with a jump instruction (generated automatically via a "#pragma" specification) to the corrected function, without changing the size of the function to be corrected.
The corrected function is allocated to a separate section that does not impact the original program.
Doing this keeps all parts of the program except for the corrected function identical to before correction.

Figure 3-16. Image of Smart Correction Memory



The smart correction feature is useful when a bug is found in a specific function after the program has been written to the flash area.

This has the following benefits compared to recompiling the program after making the correction as normal.

- If a normal recompile is performed, then the contents (allocated addresses and branch addresses) of functions other than the corrected function will change, making it necessary to evaluate the entire program.
  But if smart correction is used, locations other than the corrected function do not change, making it possible to minimize the amount of reevaluation.
- Self overwriting is also possible, because it is sufficient to overwrite only the location to change, without the need for completely overwriting the flash area.

(a) Smart correction format

The smart collection function is designated by the next format.

```
#pragma smart_correct   uncorrected-function-name    corrected-function-name
```

The function to be corrected remains as-is; a new copy is created with a different name, and that function is corrected. The CX outputs code to call the corrected function in the location of the uncorrected function.

(b) Smart correction procedure

- Upon first compilation, the compiler checks for the use of options requiring attention, such as function-optimization options.
- Prepare a pre-correction link directive file.
- Copy the function to correct, and add it to the end of the C source file containing that function. Correct the added function, and rename it.
- Add a #pragma smart_correct directive in front of the uncorrected function.
  Do not make any changes to the C source file other than adding the #pragma directive, and adding the corrected function to the end of the file.

The #pragma smart_correct directive causes a jump instruction from the uncorrected function to the corrected function to be generated automatically.

- Specify the allocation section name of the added function via a #pragma text directive.

```
#pragma text "section" corrected-function-name
```

If a static variable was added, also specify the allocation section of that variable using a #pragma section directive. Specify a new name for this section, which does not depend on the original program.

- Specify the allocation address of the corrected function name in the link directive file.
- Specify the same compiler options as the first compilation, and rebuild. A jump instruction from the pre-correction function to the corrected function is generated, with the same code as a function specifying a far jump.
- Make sure that the difference between the pre-correction hex file and the post-correction hex file is the corrected portion.

(c) Sample smart correction code

- Assume a program "prog" (prog.lmf/prog.hex) consists of three C source files: "file1.c", "file2.c", and "file3.c". Of these, there was a bug in function "funcB", defined in "file1.c". First, copy "funcB" and add it to the end of "file1.c", and change the function name to "funcBn". Next, correct "funcBn".
- Add a #pragma smart_correct directive (a) before the definition of function "funcB".
- Add a #pragma text directive (b) before the definition of corrected function "funcBn". This specifies that "funcBn" is to be allocated to a section called "text.rc".

```
[Uncorrected file1.c]
void funcA()
{
        :
        funcB();
        :
}


void funcB()
{
        :
}

void funcC()
{
        :
}
```

```
[Corrected file1.c]
void funcA()
{
        :
        funcB();
        :
}

#pragma smart_correct funcB funcBn  <- (a)
void funcB()
{
        :
}

void funcC()
{
        :
}

#pragma text "text.rc" funcBn       <- (b)
void funcBn()
{
        :
}
```

- Add a specification to allocate to the "text.rc" section to the link directives.

Example Allocate the "text.rc" section to address 0x2000000.

```
TEXT.RC: !LOAD ?RX V0x2000000 {
    text.rc = $PROGBITS ?AX text.rc.text;
};
```

- Set the options absolutely identically to those of the original program compile/assemble/link, and re-compile/assemble/link.
- Compare the original "prog.hex" file to the newly generated "prog.hex" file, and make sure that there are no differences other than "funcB" and "funcBn".

```
[Corrected file1.asm]
_funcA:
        :
        jarl funcB
        :
$smart_correct _funcB, _funcB.End, _funcBn
_funcB:
        :
_funcB.End:
_funcC:
        :
text.rc.text .cseg text
_funcBn:
        jmp [lp]
_funcBn.end:
```

```
[Assembler image of corrected file1.obj]
_funcA:
        :
        jarl funcB
        :
_funcB:                 ┐
        jr32 funcBn     │
        nop             ├─The same size as funcB which is originally
        :               │
                        ┘
_funcB.End:
_funcC:
        :
text.rc.text .cseg text
_funcBn:
        jmp [lp]
_funcBn.end:
```

(d) Cautions for the smart correction function

- You can only make additions, deletions, and modifications inside the uncorrected function.
- You cannot delete or modify variables defined outside the function. Variables can be added by defining them in a different section.
- To add a variable, explicitly specify a section and allocation location, taking care not to change already existing data areas.
- Do not add variables with initial values, because it could change the ROMization copy size.
- When copying individual items, the function in question must also be taken into consideration as a correction target.
- Make the size of the pre-correction function at least as large as the code size necessary for the call of the corrected function.
- Only optimizations closed within the function are applied to uncorrected functions.
- If a uncorrected function is a target for inline expansion (optimization), a message will be output asking whether to make it a target for smart correction.
- The compiler automatically appends the string ".text" to section names specified via "#pragma". If the target section name is specified in the input section of a link directive, then if there is nothing written before the automatically added ".text", it will be determined to be an unnecessary section, because the linker will not be able to identify the target section name.

(14) Position independent operations

Normally, when accessing variables and functions in CX, relative addressing is output, and position-independent code is output. This feature can be used to change whether addressing for accessing variables and functions uses position-independent output or fixed-address output.

For example, in multi-core programming the sections in each core are relative from the base register of that core, but when they are accessed from other cores or the common module, then the absolute address must be specified, because the base registers are different. This feature can be used to control this.

(a) Position independent operation format

Use the following format to specify position-independent operations on variables and functions.

```
#pragma pic
#pragma nopic
```

(b) Sample position independent operation code

When a "#pragma pic" directive is specified, then access to subsequently declared/defined variables and functions will use relative addresses. When a "#pragma nopic" directive is specified, then access to subsequently declared/defined variables and functions will use absolute addresses.

Example

```
#pramga nopic
extern int i;   /* "i" is accessed via the absolute address. */
#pragma pic
extetern int j; /* "j" is accessed via relative address. */
```

If the same specification is made repeatedly, then it will not cause an error:

```
#pragma nopic
extern int i;
#pragma nopic   /* Not error */
extern int j;
```

But if different directives are specified for the same variable, then it will cause an error:

```
#pragma nopic
extern int i;
#pragma pic   /* Error */
int i;
```

When performing multi-core programming, declare variables defined in another core module that you want to access after a "#pragma nopic" directive. If the "-Xmult=cmn" option was specified, then it is not necessary to specify "#pragma nopic", because it is the default.

Example

Each PE (Processing Element) program (-Xmulti=pen)

```
#pragma nopic
/* Common module declaration */
extern int cmn_var;
extern int cmn_func();
#pragma pic
/* PE local module declaration */
int pe_var;
int pe_func();
```

Common module (-Xmulti=cmn)

```
#pramga nopic   /* Does not matter whether it is included or not */
int cmn_var = 0;
int cmn_func(){
        return 1;
}
```

(c) Important information for position independent operations

- If the "-Xmulti" option is not specified, or if the "-Xmulti=pen" option is specified, then it will be assumed that "#pragma pic" is written at the beginning of the file. In this case, it will be the same as ordinary output code.
- If the "-Xmulti=cmn" option is specified, then it will be assumed that "#pragma nopic" is written implicitly at the beginning of the file. If the "-Xmulti=cmn" option is specified, then writing "#pragma pic" will cause an error.
- If the "-Xmulti=cmn" option is specified, then specifying sdata/sidata/sedata/tidata/tidata_byte/tidata_word in a "#pragma section" will cause an error.
- If "-Xmulti" is not specified, or the "-Xmulti=pen" option is specified, then when "#pragma nopic" is specified it is possible to use sdata/sidata/sedata/tidata/tidata_byte/tidata_word simultaneously in a "#pragma section", but it will hurt code efficiency.
- Making different specifications for multiple declarations will cause an error. Care is needed when coding header files.
- Features relating to symbol files (-Xsfg*, -Xsymbol_file) cannot be specified simultaneously. The behavior when they are so specified is undefined.

3.2.5 Modification of C source

By using expanded function object with high efficiency can be created. However, as expanded function is adapted in V850 microcontrollers, C source needs to be modified so as to use in other than V850 microcontrollers.

Here, 2 methods are described for shifting to the CX from other C compiler and shifting to C compiler from the CX.

<From other C compiler to the CX>

- #pragma[Note]

  C source needs to be modified, when C compiler supports the #pragma. Modification methods are examined according to the C compiler specifications.

- Expanded Specifications

  It should be modified when other C compilers are expanding the specifications such as adding keywords etc. Modified methods are examined according to the C compiler specifications.

Note #pragma is one of the pre-processing directives supported by ANSI. The character string next to #pragma is made to be recognized as directives to C compiler. If that directive does not supported by the compiler, #pragma directive is ignored and the compiler continues the process and ends normally.

<From the CX to other C compiler>

- The CX, either deletes key word or divides # fdef in order shift to other C compiler as key word has been added as expanded function.

Examples 1. Disable the keywords

```
#ifndef __CA850__
#define interrupt       /*Considered interrupt function as normal function*/
#endif
```

2. Change to other type

```
#ifdef __V850__
#define bit char    /*Change bit type variable to char type variable*/
#endif
```

3.3 Function Call Interface

This section describes how to handle arguments when a program is called by the CX.

3.3.1 Calling between C functions

- Normal function call
  --> jarl instruction
- Function call using a pointer indicating a function (and returning from function call)
  --> jmp instruction

When a C function is called from another C function, a 4-word argument is stored in the argument registers (r6 to r9). An argument in excess of 4 words is stored in the stack frame of the calling function. As with structs and parameters of type double/long long, it is stored in r6, from the least significant byte. Control is then transferred (jumps) to the called function and the value in the argument registers stored when the function was called is stored in the stack frame of the calling function.

For a function that returns a structure, create memory for the return value in the calling function, and pass the address of this memory area to the function as the first argument. In this case, the first, second, ... argument specified in the source will be treated as the second, third, ... arguments.

The CX uses r10 for function return values. If the function is of type double or long long, it uses r10 and r11, storing the lower 32 bits in r10, and the higher 32 bits in r11. For functions that return structures, the structure is stored in the address passed via the first argument; there is no explicit return value.

The stack frame is generated when the prologue code of the function, i.e., the code that is executed before the code of the main body of the function is called (processing (4) to (7) in "Figure 3-19. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Center of Stack)", "Figure 3-21. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Beginning of Stack)" is the prologue code), is executed and the stack pointer (sp) is shifted by the necessary size. The stack frame disappears when the epilogue code of the function, i.e., the code that is executed after the code of the main body of the function is executed and until control returns to the calling function (processing (i) to (iv) in "Figure 3-19. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Center of Stack)", "Figure 3-21. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Beginning of Stack)" is the epilogue code), is executed and the stack pointer (sp) is returned.

(1) Stack frame/Function call

This section explains the stack frame format and how the stack frame is generated and disappears when a function is called.

(a) Stack frame format

The CX allocates the argument register area to either the beginning of the stack or center of the stack in the stack frame, according to the argument condition. The argument conditions are as follows.

<1> When the argument register area is allocated to the beginning of the stack

The argument register area is allocated to the beginning of the stack when the area is accessed successively, exceeding the area for the 4-word argument, in the following two cases.

- If the number of arguments is variable.
- If the argument is the entity of a structure and its area extends over a 4-word area.

<2> When the argument register area is allocated to the center of the stack

In such case, it is other than the conditions mentioned above.

"Figure 3-17. Stack Frame (When Argument Register Area Is Located at Center of Stack)" shows stack frame when the argument register area is at the center of the stack and "Figure 3-18. Stack Frame (When Argument Register Area Is Located at Beginning of Stack)" shows stack frame when the argument register area is at the beginning of the stack.

Figure 3-17. Stack Frame (When Argument Register Area Is Located at Center of Stack)

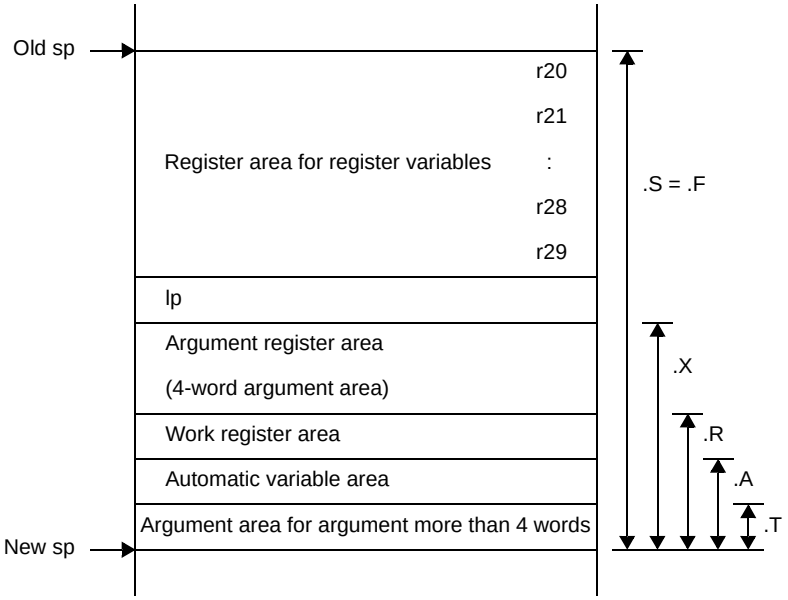
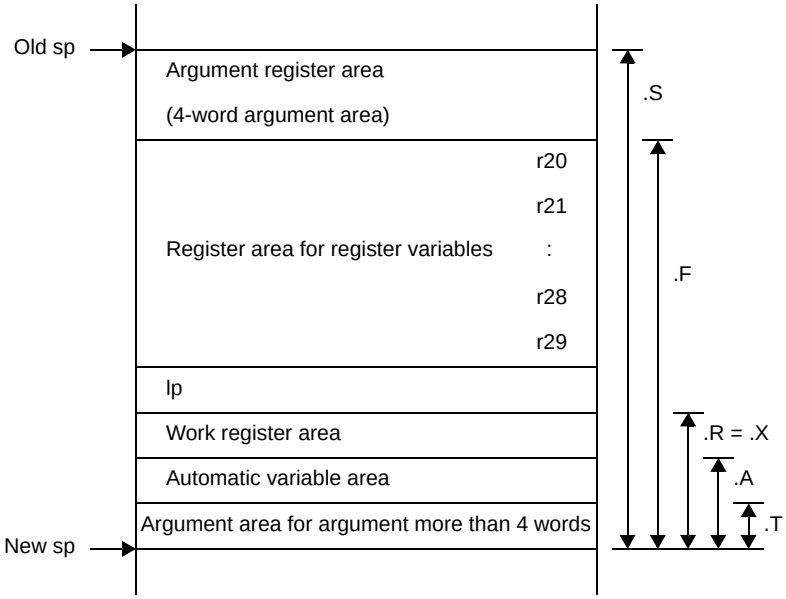


Figure 3-18. Stack Frame (When Argument Register Area Is Located at Beginning of Stack)



".S, .F, .X, .R, .A, and .T" in the figure are macros for functions output by the compiler internally. macros are used for a specific purpose, as shown in the following table.

Table 3-27. Macros for Functions

Macro Name	Meaning
.S	Stack size
.F	Stack size - Size of argument register area (if at the beginning of the stack)
.X	Size of argument register area (if at the center of the stack) + .R
.R	Size of work register area + .A
.A	Size of automatic variable area + .T
.T	Size of area for arguments of function to be called in excess of 4 words
.P	Always 0 (macro for code generation)[Note]

Note .P is not shown in "Figure 3-17. Stack Frame (When Argument Register Area Is Located at Center of Stack)" and "Figure 3-18. Stack Frame (When Argument Register Area Is Located at Beginning of Stack)" because it is always 0.

These macros are used to access the stack area. The following table shows specific access methods (access codes.

Table 3-28. Method of Accessing Stack Area

Stack Area	Access Method (Displacement [sp])
Register area for register variables (including lp)	-offset + .F*xx*[sp]
Work register area	-offset + .R*xx*[sp]
Automatic variable area	-offset + .A*xx*[sp]
Area for arguments in excess of 4 words	offset + .P*xx*[sp]
Argument register area	offset + .F*xx*[sp]
Argument register area (if at the center of the stack)	offset + .R*xx*[sp]

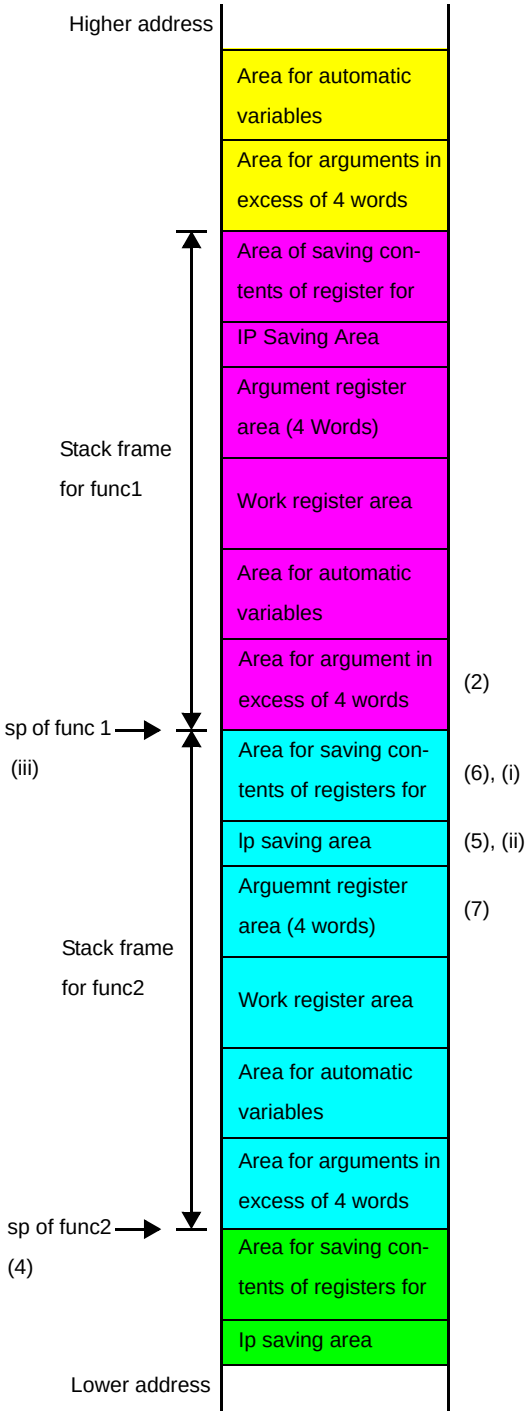
"offset" in this table is a positive integer and means the offset in each area. "*xx*" after a macro is a positive integer and indicates the frame number of the function.

(b) Generation/disappearance of stack frame when function is called (when argument register area is at center of stack)

The following explains the generation and disappearance of the stack frame when a function is called if the argument register area is at the center of the stack. This case applies to most function calls.

The following figure shows an example of the generation/disappearance of the stack frame when the function "func2" is called from the function "func1" and then execution returns to "func1".

Figure 3-19. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Center of Stack)



[Processing on func1 side when func2 is called]

(1) The arguments are stored in the argument registers.
The arguments of func2 to be called are stored in r6 to r9.

(2) The arguments in excess of 4 words are stored in the stack.
The excess arguments that cannot be stored in r6 to r9 are stored in the stack.

(3) Execution branches to func2() by the jarl instruction.

[Processing on func2() side when called by func1]

(4) sp is shifted.
The stack pointer moves to the stack to be used by func2.

(5) lp is saved.
The return address of func1 is stored.

(6) Register variable registers are saved.
These registers are saved because the register values used by func1 must be retained when func2 also uses the register variable registers.

(7) Arguments in argument register area are stored.
The values of r6 to r9 are stored. The current argument fvalues are stored in the stack because when another function is called from func2, the arguments at that time are stored in registers r6 to r9.

Since the V850Ex can perform processing (4) to (6) with the prepare instruction, the CX outputs the prepare instruction.

[Processing on func2 side when execution returns from func2 to func1]

(i) The contents of the registers for register variables are restored.
The values of the register variable registers of func1() is restored to registers.

(ii) lp is restored.
The return address of func1() is restored.

(iii) ssp is returned. The stack pointer moves back to the stack to be used by func1().

(iv) Execution is returned by the jmp [lp] instruction.

(v) Since the V850Ex can perform processing (i) to (iv) with the dispose instruction, the CX outputs the dispose instruction.

The items that are saved to the stack frame and the stack frame to be used are summarized below.

<1> Calling side - func1

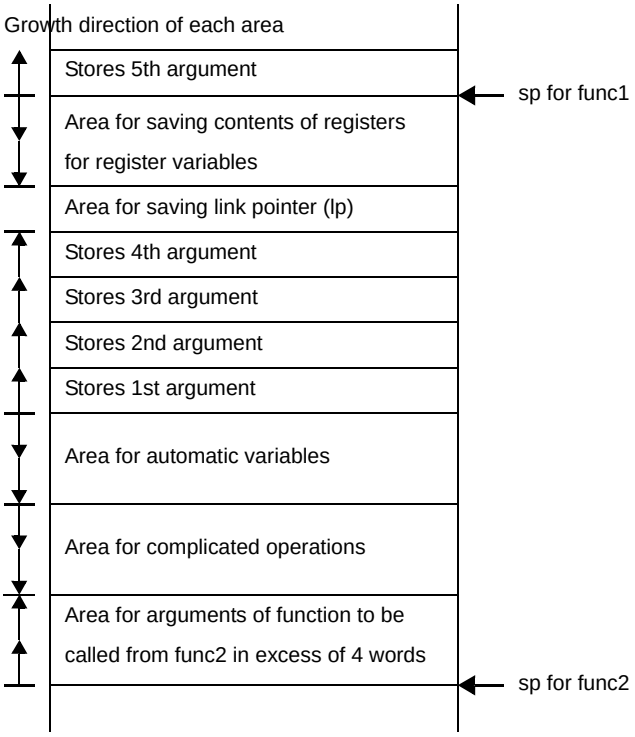
- The values of the excess arguments are called if the arguments of func2 to be called exceed 4 words.

<2> Called side - func2

- The arguments which are entered in the argument registed are passed (To enter into the argument register means to call a function (Function (fun 1)).
- Saving the link pointer (lp) (= return address of func1) of the calling side (func1) Saving the contents of the register variable registers.
- Saving the contents of the register variable registers
  The register variable registers are allocated as follows.
  In 22-register mode: "r25, r26, r27, r28, r29"
  In 26-register mode: "r23, r24, r25, r26, r27, r28, r29"
  In 32-register mode: "r20, r21, r22, r23, r24, r25, r26, r27, r28, r29"
  Of these registers, those that are used are saved.
- Area for automatic variables
- Allocating an area used for operation if a very complicated expression is used in a function Although this area is not is allocated at the lower address of the area for automatic variables if it is necessary.

If the function has a return value, that value is stored in r10.
The location of each area of the stack frame and the image of the stack growth direction of each area are illustrated below (it is assumed that func2() to be called has five arguments).

Figure 3-20. Stack Growth Direction of Each Area of Stack Frame



An example of a source calling a C function from a C function and an assembly source when that source is compiled is shown below.

Example

```
void func1(void) {
        int a, b, c, d, e;
        func2(a, b, c, d, e);
          :
}
int func2(int a, int b, int c, int d, int e) {
        register int    i;
          :
        return(i);
}
```

Assembler instructions generated when func2 is called in the above example.

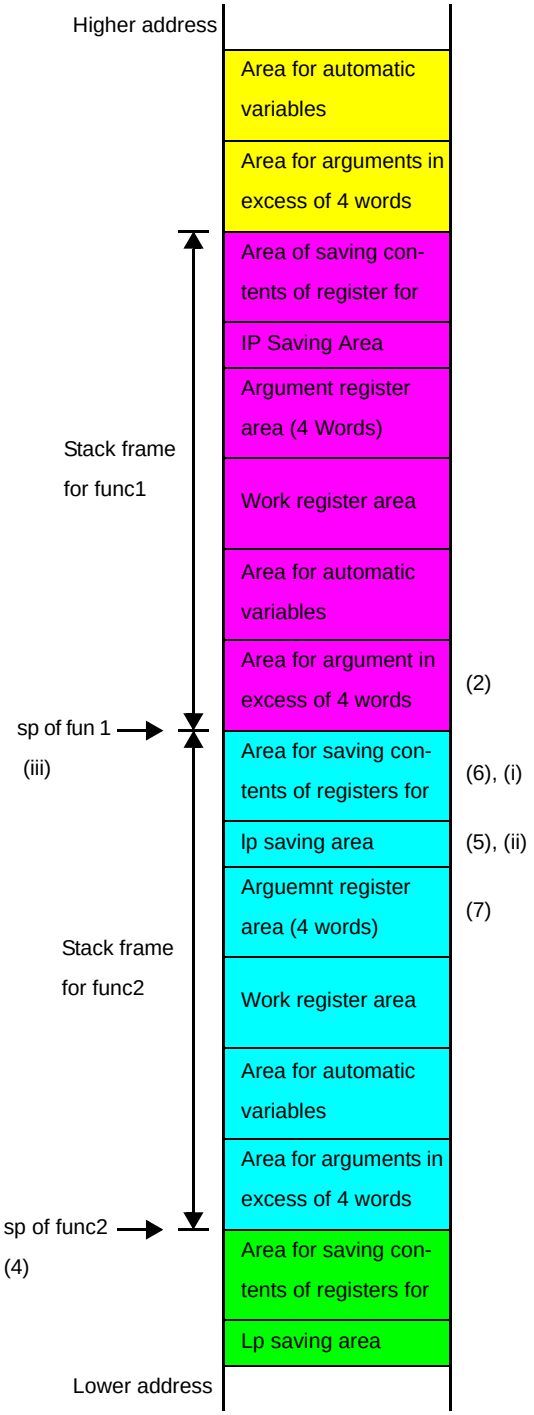
```
_func1:
        jbr     .L3
.L4:
        ld.w    -8 + .A3[sp], r6
        ld.w    -12 + .A3[sp], r7
        ld.w    -16 + .A3[sp], r8       -- (1)
        ld.w    -20 + .A3[sp], r9
        ld.w    -24 + .A3[sp], r10
        st.w    r10, [sp]               -- (2)
        jarl    _func2, lp              -- (3)
          :
        -- epilogue for func1
        -- Processing from (ii) to (iv)
.L3:
        -- prolog  for func1
        -- processing (4) and (5)
          :
        jbr     .L4
_func2:
        jbr     .L5
.L6:
        st.w    r6, .R2[sp]
        st.w    r7, 4 + .R2[sp]
        st.w    r8, 8 + .R2[sp]         -- (7)
        st.w    r9, 12 + .R2[sp]
        st.w    r29, -4 + .A2[sp]
          :
        jbr     .L2
.L2:
        ld.w    -4 + .A2[sp], r10
        dispose .X2, 0x3, [lp]
        -- (i), (ii), (iii), (iv)
.L5:
        prepare 0x3, .X2
        -- (4), (5), (6)
        jbr     .L6
```

(c) Generation/disappearance of stack frame when function is called (when argument register area is at beginning of stack)

The following explains the generation and disappearance of the stack frame when a function is called if the argument register area is at the beginning of the stack.

The following figure shows an example of the generation/disappearance of the stack frame when the function"func2" is called from the function"func1" and then execution returns to "func1".

Figure 3-21. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Beginning of Stack)



[Processing on func1 side when func2 is called]

(1) The arguments are stored in the argument registers.
The arguments of func2 to be called are stored in r6 to r9.

(2) The arguments in excess of 4 words are stored in the stack.
The excess arguments that cannot be stored in r6 to r9 are stored in the stack. This processing is performed if the number of arguments is five or more.

(3) Execution branches to func2 by the jarl instruction.

[Processing on func2 side when called by func1]

(4) sp is shifted.
The stack pointer moves to the stack to be used by func2.

(5) lp is saved.
The return address of func1 is stored.

(6) Register variable registers are saved.
These registers are saved because the register values used by func1 must be retained when func2 also uses the register variable registers.

(7) Arguments in argument register area are stored.
The values of r6 to r9 are stored. The current argument fvalues are stored in the stack because when another function is called from func2, the arguments at that time are stored in registers r6 to r9.

Since the V850Ex can perform processing (4) to (6) with the prepare instruction, the CX outputs the prepare instruction.

[Processing on func2 side when execution returns from func2 to func1]

(i) The contents of the registers for register variables are restored.
The values of the register variable registers of func1 is restored to registers.

(ii) lp is restored.
The return address of func1 is restored.

(iii) sp is returned. The stack pointer moves back to the stack to be used by func1.

(iv) Execution is returned by the jmp [lp] instruction.

Since the V850Ex can perform processing (i) to (iv) with the dispose instruction, the CX outputs the dispose instruction.

The items that are saved to the stack frame and the stack frame to be used are summarized below.

<1> Calling side - func1

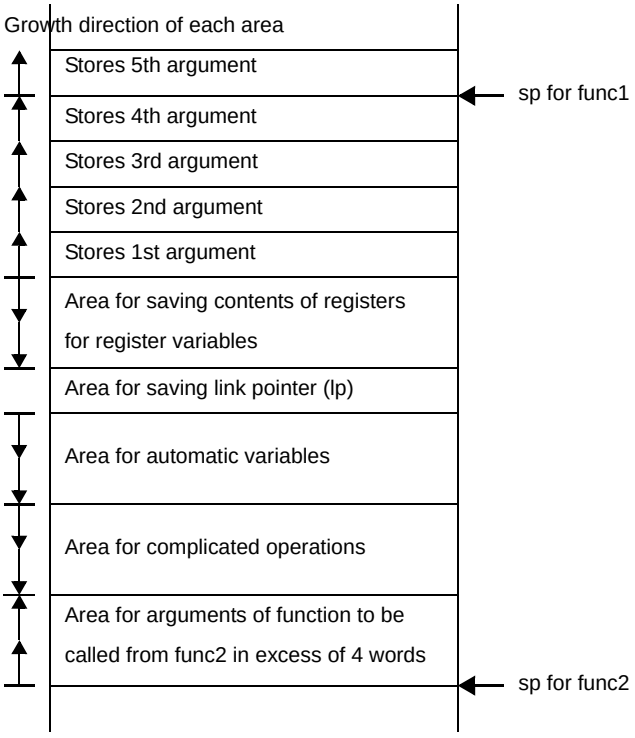
- The values of the excess arguments are called if the arguments of func2() to be called exceed 4 words.

<2> Called side - func2

- The arguments which are entered in the argument registed are passed (To enter into the argument register means to call a function (Function (fun 1)).
- Saving the link pointer (lp) (= return address of func1) of the calling side (func1) Saving the contents of the register variable registers.
- Saving the register variable registers.
- Area for automatic variables
- Allocating an area used for operation if a very complicated expression is used in a function Although this area is not is allocated at the lower address of the area for automatic variables if it is necessary.

If the function has a return value, that value is stored in r10.
The location of each area of the stack frame and the image of the stack growth direction of each area are illustrated below (it is assumed that func2 to be called has five arguments).

Figure 3-22. Stack Growth Direction of Each Area of Stack Frame



An example of a source calling a C function from a C function and an assembly source when that source is compiled is shown below.

Example

```
void func1(void) {
        int a, b, c, d, e;
        func2(a, b, c, d, e);
          :
}
int func2(int a, int b, int c, int d, int e) {
        register int    i;
          :
        return(i);
}
```

Assembler instructions generated when func2 is called in the above example.

```
_func1:
        jbr     .L3
.L4:
        ld.w    -8 + .A3[sp], r6
        ld.w    -12 + .A3[sp], r7
        ld.w    -16 + .A3[sp], r8       -- (1)
        ld.w    -20 + .A3[sp], r9
        ld.w    -24 + .A3[sp], r10
        st.w    r10, [sp]               -- (2)
        jarl    _func2, lp              -- (3)
          :
        -- epilogue for func1
        -- Processing from (ii) to (iv)
.L3:
        -- Prolog  for func1
        -- Processing (4) and (5)
          :
        jbr     .L4
_func2:
        jbr     .L5
.L6:
        st.w    r6, .F2[sp]
        st.w    r7, 4 + .F2[sp]
        st.w    r8, 8 + .F2[sp]         -- (7)
        st.w    r9, 12 + .F2[sp]
          :
        st.w    r29, -4 + .A2[sp]
        jbr     .L2
.L2:
        ld.w    -4 + .A2[sp], r10
        dispose .X2, 0x3
        -- (i), (ii), (iii)
        add     .S2 - .F2, sp           -- (iii)
        jmp     [lp]                    -- (iv)
.L5:
        add     .F2 - .S2, sp           -- (4)
        prepare 0x3, .X2
        -- (4), (5), (6)
        jbr     .L6
```

3.3.2 Prologue/Epilogue processing function

The CX can reduce the object size in part of the prologue/epilogue processing of a function by calling a runtime library. It is called as "Prologue/Epilogue Runtime" process. Because the prologue/epilogue processing of a function is predetermined, it is prepared as runtime library functions and these functions are called when a function is called or execution returns to a function.

An example of the assembler code of the prologue/epilogue processing of a function is shown below.

Numbers in parentheses in this example correspond to those in "Figure 3-19. Generation/Disappearance of Stack Frame (When Argument Register Area Is Located at Center of Stack)".

Example

```
int func(int a, int b, int c, int d, int e) {
        register int    i;
          :
        return(i);
}
```

Assembler instruction in prologue/epilogue processing of function "func" in above example

[Code when runtime library function is not used]

```
_func:
.BB.LABEL.0:
        prepare 3, 16           --(4)(5)(6)
        st.w    r6, 0[sp]
        st.w    r7, 4[sp]
        st.w    r8, 8[sp]       --(7)
        st.w    r9, 12[sp]
              :
        mov     r29, r11
.BB.LABEL.1:
        mov     r11, r10
        dispose 16, 3, [lp]     --(i),(ii),(iii),(iv)
_func.end:
```

[Code when runtime library function is used]

```
_func:
.BB.LABEL.0:
        callt   9               --(4)(5)(6)
        st.w    r6, 0[sp]
        st.w    r7, 4[sp]
        st.w    r8, 8[sp]       --(7)
        st.w    r9, 12[sp]
              :
        mov     r29, r11
.BB.LABEL.1:
```

```
        mov     r11, r10
        callt   39              --(i),(ii),(iii),(iv)
_func.end:
```

(1) Specifying use of runtime library function for prologue/epilogue of function

Specify the compiler option "-Xpro_epi_runtime=on" to call the runtime library for prologue/epilogue. Specify the -Xpro_epi_runtime=off option if the runtime library is not called.

When an optimization option other than "-Ospeed (execution speed priority optimization)" is specified, however, the runtime library is automatically called for the prologue/epilogue of a function. That is, the compiler option "-pro_epi_runtime=on" is automatically specified.

If an option other than "-Ospeed" is specified and if a runtime library should not be called, specify the -Xpro_epi_runtime=off option.

The -Xpro_epi_runtime option can be specified in each source file, so a file for which the runtime library is called and a file for which the runtime library is not called can be used together.

When a runtime library is called for the prologue/epilogue of a function by specifying the -Xpro_epi_runtime=on option, a dedicated section ".pro_epi_runtime" is necessary.

Consequently, the following definition must be described by a link directive.

```
.pro_epi_runtime = $PROGBITS    ?AX     .pro_epi_runtime;
```

Table information of the prologue/epilogue runtime function is allocated to this section.

(2) Calling runtime library for prologue/epilogue

The following instruction is used to call the prologue/epilogue runtime function of a function.

The CALLT instruction is a 2-byte instruction. The code size can be reduced by using this instruction for calling a function. The CALLT instruction requires a pointer that indicates that the table of the function subject to the CALLT instruction is set to the CTBP (Callt Base Pointer) register. If processing of the setting is missing from the program, the error message is output during linking.

If processing of the setting is missing from the program, the following error message is output during linking.

Add the following instruction to the startup routine.

```
mov     #___PROLOG_TABLE, r12   --three underscores "_" before "PROLOG"
ldsr    r12, 20
```

At this time, _ _ _ PROLOG_TABLE is the first symbol of the function table of the runtime function of the prologue/epilogue of a function, and the function table itself is allocated to the ".pro_epi_runtime" section by setting it to CTEB. The r12 register is used in the above example, but it is not always necessary to use r12.

If the CALLT instruction provided in the CX is used for any purpose other than calling a runtime library for the prologue/epilogue of a function, the CTBP register contents must be saved/restored If the CALLT instruction is used by another object, such as middleware or a user-created library, and if a code that saves/restores the CTBP register contents is missing or cannot be inserted in that object, a runtime library for the prologue/epilogue of a function cannot be called In this case, suppress calling the runtime library by specifying the -Xpro_epi_runtime=off option.

See the Relevant Device 's Architecture User' s Manual of each device for details of the CALLT instruction and CTEB register.

(3) Notes on calling runtime library for prologue/epilogue of function

Note the following points when calling a runtime library for the prologue/epilogue of a function.

- Calling a runtime library for the prologue/epilogue of a function degrades the execution speed because a function is called. Specify the -Xpro_epi_runtime=off option to avoid this. Specifying this option in file units is effective.
- In the case of a program in which few functions are called, the code size may not be reduced even if a runtime library is called for the prologue/epilogue. If no real effect can be expected, specify the -Xpro_epi_runtime=off option.
- Note the following points when calling a runtime library for the prologue/epilogue of a function. Calling a runtime library for the prologue/epilogue of a function degrades the execution speed because a function is called.

3.3.3 far jump function

The CX outputs a code using the jarl instruction when a function is called.

```
jarl    _func1, lp
```

The architecture allows only a sign-extended value of up to 22 bits (22-bit displacement) to be specified as the first operand of the jarl instruction.

This means that, if the branch destination is not within ± 2MB range from the branch point, branching cannot take place and the linker outputs the error message.

This can be solved easily by allocating as shown below, however, the branch destination may not be able to be located within this range depending on target system. The "far jump" function solves this.

- The branch destination within ± 2MB range from the branch point.

When the far jump function is used, a code that uses the jmp instruction is output when a function is called. As a result, execution can branch to the entire 32-bit space of the V850. However, one of the general purpose register is used.

Function call using far jump function is called "far jump calling".

(1) Specifying far jump

When calling a function using the far jump function, prepare a file in which functions to be called by the far jump function are enumerated (file listing functions to be called by the far jump function), and use the compiler option "-Xfar_jump".

```
-Xfar_jump=file listing functions to be called by far jump function
```

See the next section for the format of the file listing the functions to be called by the far jump function.

(2) File listing functions to be called by far jump function

This section explains the format of the file that enumerates the functions to be called by using the far jump function. Describe one function to which the far jump function is applied in one line. Describe a C function name with "_" (underscore) prefixed.

[Sample of file listing functions to be called by far jump]

```
_func_led
_func_beep
_func_motor
  :
_func_switch
```

If the following description is made instead of "_function-name", all the functions are called using the far jump function.

```
{all_function}
```

If {all_function} is specified, all the functions are called by the far jump function, even if "_function-name" is specified.
The far jump function can also be applied to the following functions, as well as to user functions.

- Standard library functions
- Runtime library functions
- System calls of real-time OS

If the following is coded instead of "_function-name", then all interrupt functions will be called via far jump.

```
{all_interrupt}
```

Note the following points when describing the file listing the functions to be called by the far jump function.

- Only ASCII characters can be used.
- Comments must not be inserted.
- Describe only one function in one line.
- A blank and tab may be inserted before and after a function name.
- Up to 1,023 characters can be described in one line. A blank or tab is also counted as one character.
- Describe a C function name with "_" (underscore) prefixed to the function name.
- The far jump function cannot be used together with the re-link function of the flash memory/external ROM.

(3) Examples of using far jump function

Examples of using the far jump function are shown below.

(a) User function (same applies to standard functions)

[C source file]

```
extern  void    func3(void);

void func(void)
{
        func3();
}
```

[File listing functions to be called by far jump]

```
_func3
```

[Normal calling code]

```
#@CALL_ARG
jarl    _func3, lp
```

[Far jump calling code]

```
#@CALL_ARG
        movea   _func3, tp, r10
        movea   .L18, tp, lp
        jmp     [r10]
.L18:
```

(b) Runtime function (when calling a macro)

[File listing functions to be called by far jump]

```
___mul
```

[Normal calling code]

```
.macro  mul     arg1, arg2
        add     -8, sp
        st.w    r6, [sp]
        st.w    r7, 4[sp]
        mov     arg1, r6
        mov     arg2, r7
        jarl    ___mul, lp
        ld.w    4[sp], r7
        mov     r6, arg2
        ld.w    [sp], r6
        add     8, sp
.endm
```

[Far jump calling code]

```
.macro  mul     arg1, arg2
        .local  macro_ret
        add     -8, sp
        st.w    r6, [sp]
        st.w    r7, 4[sp]
        mov     arg1, r6
        mov     arg2, r7
        movea   macro_ret, tp, r31
        .option nowarning
        movea   #___mul, tp, r1
        jmp     [r1]
        .option warning
macro_ret:
        ld.w    4[sp], r7
        mov     r6, arg2
        ld.w    [sp], r6
        add     8, sp
.endm
```

(c) Runtime function (when direct calling)

[File listing functions to be called by far jump]

```
___mul
```

[Normal calling code]

```
mov     r12, r6
mov     r13, r7
#@CALL_ARG      r6, r7
#@CALL_USE      r6, r7
jarl    ___mul, lp
mov     r6, r13
```

[Far jump calling code]

```
        mov     r12, r6
        mov     r13, r7
        #@CALL_ARG      r6, r7
        #@CALL_USE      r6, r7
        movea   #___mul, tp, r14
        movea   .L13, tp, lp
        jmp     [r14]
.L13:
        mov     r6, r13
```

The compiler automatically selects whether a runtime macro is called or a runtime function is directly called by judging the register efficiency in the process of optimization.

(d) System calls of real-time OS

[File listing functions to be called by far jump]

```
_ext_tsk
```

[Normal calling code]

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
jr      _ext_tsk    --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

[Far jump calling code]

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
movea   #_ext_tsk, tp, r10
jmp     [r10]       --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

3.4 Section Name List

The following table lists the names, section types, and section attributes of these reserved sections.

Table 3-29. Reserved Sections

Name[Note 1]	Description	Section Type	Section Attribute
.bss	.bss section	NOBITS	AW
.const	.const section	PROGBITS	A
.data	.data section	PROGBITS	AW
.ext_info .ext_info_boot	Information section for flash/external ROM re-link function	PROGBITS	None
.ext_table	Branch table section for flash/external ROM re-link function	PROGBITS	AX
.ext_tgsym	Information section for flash/external ROM re-link function	PROGBITS	None
.gptabname	Global pointer table[Note 2]	GPTAB	None
.pro_epi_runtime	Prologue/epilogue run-time call section	PROGBITS	AX
.regmode	Register mode information	REGMODE	None
.relname	Relocation information	REL	None
.relaname	Relocation information	RELA	None
.sbss	.sbss section	NOBITS	AWG
.sconst	.sconst section	PROGBITS	A
.sdata	.sdata section	PROGBITS	AWG
.sebss	.sebss section	NOBITS	AW
.sedata	.sedata section	PROGBITS	AW
.shstrtab	String table where the section name is saved	STRTAB	None
.sibss	.sibss section	NOBITS	AW
.sidata	.sidata section	PROGBITS	AW
.strtab	String table	STRTAB	None
.symtab	Symbol table	SYMTAB	None
.text	.text section	PROGBITS	AX
.tibss	.tibss section	NOBITS	AW
.tibss.byte	.tibss.byte section	NOBITS	AW
.tibss.word	.tibss.word section	NOBITS	AW
.tidata	.tidata section	PROGBITS	AW
.tidata.byte	.tidata.byte section	PROGBITS	AW
.tidata.word	.tidata.word section	PROGBITS	AW
.debug_info	Debug information	PROGBITS	None
.debug_line	Line and column information	PROGBITS	None
.debug_loc	Location list information	PROGBITS	None
.version	Version information	PROGBITS	None
.float_info	Floating-point operation information	FLOATINFO	None

Name[Note 1]	Description	Section Type	Section Attribute
.multi	Multi-core information	MULTI	None

Notes 1. The name part of .gptabname, .relname, and .relaname indicates the name of the section corresponding to each respective section.

2. This is information that is used when processing the linker' s -Xsdata_info option.

Remark ".cmn/.pen (n=1...N)" is added to the ends of (default) section names reserved for multi-core.
"_CMN/_PE*n* (n=1...N)" is added to the ends of (default) segment names reserved for multi-core.
The section names and segment names reserved for multi-core are shown below.

- Reserved section names

.sconst.cmn, .pro_epi_runtime, .const.cmn, .text.cmn, .data.cmn, .bss.cmn,
.sconst.pe1, .const.pe1, .text.pe1, .data.pe1, .sdata.pe1, .sbss.pe1, .bss.pe1 ,.sedata.pe1, .sebss.pe1,
.tidata.byte.pe1, .tibss.byte.pe1, .tidata.word.pe1, .tibss.word.pe1, .sidata.pe1, .sibss.pe1
:
.sconst.pe*n*, .const.pe*n*, .text.pe*n*, .data.pe*n*, .sdata.pe*n*, .sbss.pe*n*, .bss.pe*n*,.sedata.pe*n*, .sebss.pe*n*,
.tidata.byte.pe*n*, .tibss.byte.pe*n*, .tidata.word.pe*n*,.tibss.word.pe*n*,.sidata.pe*n*, .sibss.pe*n*

- Reserved segment names

SCONST_CMN, CONST_CMN, TEXT_CMN, DATA_CMN,
SCONST_PE1, CONST_PE1, TEXT_PE1, DATA_PE1, SEDATA_PE1, SIDATA_PE1,
:
SCONST_PE*n*, CONST_PE*n*, TEXT_PE*n*, DATA_PE*n*, SEDATA_PE*n*, SIDATA_PE*n*

CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICATIONS

This chapter explains the assembly language specifications supported by the CX assembler.

4.1 Description of Source

This section explains description of source, expressio, and operators.

4.1.1 Description

An assembly language statement consists of a "symbol", a "mnemonic", "operands", and a "comment".

```
[symbol][:Δ]       [mnemonic]     [operand], [operand]    ;[comment]
```

Separate labels by colons or one or more whitespace characters. Whether colons or spaces are used, however, depends on the instruction coded by the mnemonic.

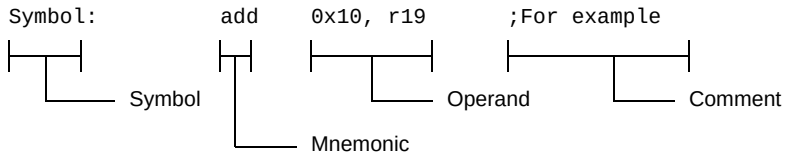
It is irrelevant whether blanks are inserted in the following location.

- Between the symbol name and colon
- Between the colon and mnemonic
- Before the second and subsequent operands
- Before semicolon that indicates the beginning of a comment

One or more blank is necessary in the following location.

- Between the mnemonic and the operand

Figure 4-1. Organization of Assembly Language Statement



One assembly language statement is described on one line. There is a line feed (return) at the end of the statement.

(1) Character set

The characters that can be used in a source program (assembly language) supported by the asembler are the following 3 types of characters.

- Language characters
- Character data
- Comment characters

(a) Language characters

These characters are used to code instructions in the source.

Table 4-1. Language Characters and Usage of Characters

Character	Usage
Lowercase letter (a-z)	Constitutes a mnemonic, identifier, and constant
Uppercase letter (A-Z)	Constitutes an identifier and constant
_ (underscore)	Constitutes an identifier
.(period)	Constitutes an identifier and constant
@	Constitutes an identifier and constant
~	Constitutes an identifier and constant
Numerals	Constitutes an identifier and constant
, (comma)	Delimits an operand
: (colon)	Delimits a label
; (semicolon)	Beginning of comment
*	Multiplication operator
/	Division operator
+	Positive sign and addition operator
- (hyphen)	Negative sign and subtraction operator
' (single quotation)	Character constant and symbol indicating a complete macro parameter
<	Relational operator
>	Relational operator
( )	Specifies an operation sequence
$	Symbol indicating the location counter Symbol indicating the start of a control instruction equivalent to an assembler option Symbol specifying relative addressing gp offset reference of label
=	Relational operator
!	Beginning immediate addressing and negation operator
Δ (blank)	Field delimiter
~	Concatenation symbol (in macro body)
&	Logical product operator
#	Beginning indicates and comment
[ ]	Indirect indication symbol
"(double quotation)	Start and end of character string constant
%	ep offset referring of a label and remainder operator
<<	Left shift operator
>>	Right shift operator
|	Logical sum operator
^	Exclusive OR operator

(b) Character data

Character data refers to characters used to write character string constant, character constant, and the quote-enclosed operands of some control instructions.

Caution Character data can use all characters except 0x00 (including multibyte kanji, although the encoding depends on the OS). If 0x00 is encountered, an error occurs and all characters from the 0x00 to the closing single quote (') are ignored.

(c) Comment characters

Comment characters are used to write comments.

Caution Comment characters and character data have the same character set.

(2) Symbol

The symbol field is for symbols, which are names given to addresses and data objects. Symbols make programs easier to understand.

(a) Symbol types

Symbols can be classified as shown below, depending on their purpose and how they are defined.

Symbol Type	Purpose	Definition Method
Label	Used as labels for addresses and data objects in source programs.	Write a symbol followed by a colon (:).
External reference name	Used to reference symbols defined by other source modules.	Write in the operand field of an .extern directive.
Section name	Used at link time.	Write in the symbol field of a .cseg, .dseg or .org directive.
Macro name	Use to name macros in source programs.	Write in the symbol field of macro directive.

(b) Conventions of symbol description

Observe the following conventions when writing symbols.

- The characters which can be used in symbols are the alphanumeric characters and special characters (?, @, _).

  The first character in a symbol cannot be a digit (0 to 9).
- The maximum number of characters for a symbol is 4,294,967,294 (=0xFFFFFFFE) (theoretical value). The actual number that can be used depends on the amount of memory, however.
- Reserved words cannot be used as symbols.

  See "4.5 Reserved Words" for a list of reserved words.
- The same symbol cannot be defined more than once.

  However, a symbol defined with the .set directive can be redefined with the .set directive.
- The assembler distinguishes between lowercase and uppercase characters.
- When a label is written in a symbol field, the colon (:) must appear immediately after the label name.

Example Correct symbols

```
CODE01  .cseg   text        ; "CODE01" is a segment name.
VAR01   .set    0x10        ; "VAR01" is a symbol.
LAB01:  .dw     0           ; "LAB01" is a label.
```

Example Incorrect symbols

```
1ABC    .set    3           ; The first character is a digit.s
LAB     mov     r10, r11    ; "LAB"is a label and must be separated from the mnemonic
                            ; field by a colon ( : ).
FLAG:   .set    0x10        ; The colon ( : ) is not needed for symbols.
```

Example A statement composed of a symbol only

```
ABCD:                       ; ABCD is defined as a label.
```

(c) Points to note about symbols

The assembler generates a name automatically when a section definition directive does not specify a name. These section names are listed below.

Duplicate section name definitions are errors.

Section Name	Directive	Relocation Attribute
.text	.cseg directive	TEXT
.const		CONST
.sconst		SCONST
.bss	.dseg directive	BSS
.data		DATA
.sbss		SBSS
.sdata		SDATA
.sebss		SEBSS
.sedata		SEDATA
.sibss		SIBSS
.sidata		SIDATA
.tibss		TIBSS
.tibss.byte		TIBSS.BYTE
.tibss.word		TIBSS.WORD
.tidata		TIDATA
.tidata.byte		TIDATA.BYTE
.tidata.word		TIDATA.WORD
SECUR_ID	.cseg directive	SECUR_ID

Section Name	Directive	Relocation Attribute
OPT_BYTE	.dseg directive	OPT_BYTE

(d) Symbol attributes

Every symbol and label has both a value and an attribute.
The value is the value of the defined data object, for example a numerical value, or the value of the address itself.
Section names, module names, and macro names do not have values.
The following table lists symbol attributes.

Attribute Type	Classification	Value
BIT	- Symbols defined as bit values - Symbols defined with the EXTBIT directive	Decimal notation: -2147483648 to 2147483647 Hexadecimal notation: 0x80000000 to 0x7FFFFFFF (signed)
CSEG	Section names defined with the .cseg directive	These attribute types have no values.
DSEG	Section names defined with the .dseg directive	
MACRO	Macro names defined with the Macro directive	These attribute types have no values.
FNUMBER	Symbols defined with the FLOAT directive (Single precision floating point)	1.40129846e-45 to 3.40282347e+38
DFNUMBER	Symbols defined with theDFLOAT directive (Double-precision floating point)	4.9406564584124654e-324 to 1.7976931348623157e+308

Example

```
BIT1    .set    0xFFE20.0   ; The symbol BIT1 has the BIT attribute and a value of 0xFFE20.0.
```

(3) Mnemonic field

Write instruction mnemonics, directives, and macro references in the mnemonic field.
If the instruction or directive or macro reference requires an operand or operands, the mnemonic field must be separated from the operand field with one or more blanks or tabs.
However, if the first operand begins with "#", "$","!", or "[", the statement will be assembled properly even if nothing exists between the mnemonic field and the first operand field.

Example Correct mnemonics

```
add     r11, r12
reti
di
```

Example Incorrect mnemonics

```
addr11, r12     ; There is no blank between the mnemonic and operand fields.
r eti           ; The mnemonic field contains a blank.
HLT             ; This is an instruction that cannot be coded in the mnemonic field.
```

(4) Operand field

In the operand field, write operands (data) for the instructions, directives, or macro references that require them.
Some instructions and directives require no operands, while others require two or more.
When you provide two or more operands, delimit them with a comma (,).
The following types of data can appear in the operand field:

- Constants (numeric constants, character constants, character string constants)
- Register names
- Relocation attributes of section definition directives
- Symbols
- Expressions

See the user's manual of the target device for the format and notational conventions of instruction set operands.
The following sections explain the types of data that can appear in the operand field.

(a) Constants

A constant is a fixed value or data item and is also referred to as immediate data.
There are numeric constants, character constants and character string constants.

- Numeric constants

Integer constants can be written in binary, octal, decimal, or hexadecimal notation.
Integer constants has a width of 32 bits. A negative value is expressed as a 2's complement. If an integer value that exceeds the range of the values that can be expressed by 32 bits is specified, the assembler uses the value of the lower 32 bits of that integer value and continues processing (it does not output any message).

Type	Notation	Example
Binary	Append a "B" or "Y" suffix to the number. Append an "0b" suffix to the number.	1101B 1101Y 0b1101
Octal	Append an "0" suffix to the number.	074
Decimal	Simply write the number.	128
Hexadecimal	Append an "0x" suffix to the number.	0xA6

Floating constants consist of the following elements. Specify the exponent and mantissa as decimal constants. Do not use (3), (4), or (5) if an exponent expression cannot be used.

(1) sign of mantissa part ("+" is optional)
(2) mantissa part
(3) 'e' or 'E' indicating the exponent part
(4) sign of exponent part ("+" is optional)
(5) exponent part

Example

```
123.4
-100.
10e-2
-100.2E+5
```

You can indicate that the number is a floating constant by appending "0f" or "0F" to the front of the mantissa.

Example

```
0f10
```

- Character constants

A character constant consists of a single character enclosed by a pair of single quotation marks (' ') and indicates the value of the enclosed character[Note].

If any of the escape sequences listed below is specified in " ' " and " ' ", the assembler regards the sequence as being a single character.

Example

```
"ab"            ; 0x6162
"A"             ; 0x41
"A\""           ; 0x4122
" "             ; 0x20 (1 blank)
""              ;
```

Note If a character constant is specified, the assembler assumes that an integer having the value of that character constant is specified.

Table 4-2. Value and Meaning of Escape Sequence

Escape Sequence	Value	Meaning
\0	0x00	null character
\a	0x07	Alert
\b	0x08	Backspace
\f	0x0C	Form feed
\n	0x0A	Line feed
\r	0x0D	Carriage return
\t	0x09	Horizontal tab
\v	0x0B	Vertical tab
\\	0x5C	Back slash
\'	0x27	Single quotation marks
\"	0x22	Double quotation mark
\?	0x3F	Question mark
\ddd	0 to 0377	Octal number of up to 3 digits (0 < d < 7) [Note]
\xhh	0 to 0xFF	Hexadecimal number of up to 2 digits (0 < h < 9, a < h < f, or A < h < F)

Note If a value exceeding "\377" is sp value of the escape sequence becomes the lower 1 byte. Cannot be of value more than 0377. For example value of"\777"is 0377.

- Character string constants

  A character-string constant is expressed by enclosing a string of characters from those shown in "(1) Character set", in a pair of single quotation marks (").

  The string constant is assembled with the character-code values specified via the -Xcharacter_set option.

  A "\0" is appended to the end of the result.

  To include the single quote character in the string, write it twice in succession.

Example

```
"ab"            ; 0x616200
"A"             ; 0x4100
"A\""           ; 0x412200
" "             ; 0x2000 (1 blank)
""              ; 0x00
```

(b) Register names

The following registers can be named in the operand field:

- General registers
- General register pairs
- Special function registers
- Others (PSW, CY, RBn, [BC], [DE], [HL], [DE+byte], [HL+byte], [HL+B], [HL+C])

General registers and general register pairs can be described with their absolute names, as well as with their function names.

The register names that can be described in the operand field may differ depending on the type of instruction. For details of the method of describing each register name, see the user's manual of each device for which software is being developed.

(c) Relocation attributes of section definition directives

Relocation attributes can appear in the operand field.

See "4.2.2 Section definition directives" for more information about relocation attributes.

(d) Symbols

When a symbol appears in the operand field, the address (or value) assigned to that symbol becomes the operand value.

Example

```
HERE:   jmp32   #THEREE         ; THERE indicates the address of label THERE.
         :
THERE:  add     r11, r12
VALUE   .set    0x100
        movea   VALUE, r11, r12 ; VALUE indicates the value of name VALUE.
```

(e) Expressions

An expression is a combination of constants, location counter (indicated by $) and symbols, by an operator.
Expressions can be specified as instruction operands wherever a numeric value can be specified.
See "4.1.2 Expressions and operators" for more information about expressions.

Example

```
TEN     .set    0x10
        mov     TEN - 0x05, r12
```

In this example, "TEN - 0x05" is an expression.
In this expression, a symbol and a numeric value are connected by the - (minus) operator. The value of the expression is 0x0B, so this expression could be rewritten as "mov 0x0B, r12".

(5) Comment

Describe comments in the comment field, after a semicolon (;).

The comment field continues from the semicolon to the new line code at the end of the line, or to the EOF code of the file.

Comments make it easier to understand and maintain programs.

Comments are not processed by the assembler, and are output verbatim to assembly lists.

Characters that can be described in the comment field are those shown in "(1) Character set".

<Comment example>

```
; sample program                          ] Lines with comment fields only
        .extern __tp_TEXT, 4
        .extern __gp_DATA, 4
        .extern _main
RESET   .cseg   text                    ; Reset Handler address
        jr      __boot                  ; Jump to __boot
.text   .cseg   text                    ; Text section
        .align  4                       ; Code alignment
        .public  __boot                 ; Alignment
 __boot:
        mov     #__tp_TEXT, tp          ; Set tp
        mov     #__gp_DATA, gp          ; Set gp
        .extern __ssbss, 4
        .extern __esbss, 4
   ; start of bss initialize             ] Lines with comment fields only
        mov     #__ssbss, r13
        mov     #__esbss, r13
        cmp     r12, r13
        jnl     sbss_init_end

 sbss_init_loop:
        st.w    r0, 0[r13]
        add     4, r13
        cmp     r12, r13
        jl      sbss_init_loop
 sbss_init_end:
   ; end of bss initialize               ] Lines with comment fields only
        jarl    _main, lp               ; Call main function
.data   .dseg  data
        .align 4
 data_area:
        .dw     0x00                    ; data1
        .dhw    0x01                    ; data2
        .db     0xFF                    ; data3
        .db     0xFE                    ; data4
```

Lines with comments in comment fields

Lines with comments in comment fields

4.1.2 Expressions and operators

An expression is a symbol, constant or location counter (indicated by $), an operator combined with one of the above, or a combination of operators.

Elements of an expression other than the operators are called terms, and are referred to as the 1st term, 2nd term, and so forth from left to right, in the order that they occur in the expression.

The assembler supports the operators shown in "Table 4-3. Operator Types". Operators have priority levels, which determine when they are applied in the calculation. The priority order is shown in "Table 4-4. Operator Precedence Levels".

The order of calculation can be changed by enclosing terms and operators in parentheses "( )".

Example

```
mov32   5 * (SYM + 1), r12
```

In the above example, "5 * (SYM+1)" is an expression. "5" is the 1st term, "SYM" is the 2nd term, and "1" is the 3rd term. The operators are "*", "+", and "( )".

Table 4-3. Operator Types

Operator Type	Operators
Arithmetic operators	+, -, *, /, MOD(%), +sign, -sign
Logic operators	!, &, |, ^
Relational operators	==, !=, >, >=, <, <=, &&, ||
Shift operators	>>, <<
Byte separation operators	HIGH, LOW
2-byte separation operators	HIGHW, LOWW, HIGHW1
Special operators	DATAPOS, BITPOS
Other operator	( )

The above operators can also be divided into unary operators, special unary operators and binary operators.

Unary operators	+sign, -sign, NOT(!), HIGH, LOW, HIGHW, LOWW, HIGHW1
Special unary operators	DATAPOS, BITPOS
Binary operators	+, -, *, /, MOD(%), &, |, ^, ==, =, >, >=, <, <=, >>, <<, &&, ||

Table 4-4. Operator Precedence Levels

Priority	Level	Operators
Higher	1	+sign, -sign, NOT(!)
	2	*, /, MOD(%), >>, <<
	3	&, |, ^
	4	+, -
	5	==, !=, >, >=, <, <=
Lower	6	&&, ||

Expressions are operated according to the following rules.

- The order of operation is determined by the priority level of the operators.
  When two operators have the same priority level, operation proceeds from left to right, except in the case of unary operators, where it proceeds from right to left.
- Sub-expressions in parentheses "( )" are operated before sub-expressions outside parentheses.
- Expressions are operated using unsigned 32-bit values.
  If intermediate values overflow 32 bits, the overflow value is ignored.
- If the value of a constant exceeds 32 bits, an error occurs, and its value is calculated as 0.
- In division, the decimal fraction part is discarded.
  If the divisor is 0, an error occurs and the result is 0.
- Negative values are represented as two's complement.
- External reference symbols are evaluated as 0 at the time when the source is assembled (the evaluation value is determined at link time).

(1) Evaluation examples

Expression	Evaluation
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0xFFFFFFFF
-1 > 1	0x0 (False)
EXT[Note] + 1	1

Note EXT: External reference symbols

4.1.3 Arithmetic operators

The following arithmetic operators are available.

Operator	Overview
+	Addition of values of first and second terms.
-	Subtraction of value of first and second terms.
*	Multiplacation of value of first and second terms.
/	Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.
MOD(%)	Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.
+sign	Returns the value of the term as it is.
-sign	The term value 2 complement is sought.

+

Addition of values of first and second terms.

[Function]

Returns the sum of the values of the 1st and 2nd terms of an expression.

[Application example]

```
        .org    0x100
START:  jmp     #START + 6      ; (1)
```

(1) The jmp instruction causes a jump to "address of the START label plus 6", namely, to address "0x100 + 0x6 = 0x106" when START label is 0x100.
Therefore, (1) in the above example can also be described as: START: jmp #0x106.

-

Subtraction of value of first and second terms.

[Function]

Returns the result of subtraction of the 2nd-term value from the 1st-term value.

[Application example]

```
        .org    0x100
BACK:   jmp     !BACK - 6       ; (1)
```

(1) The jmp instruction causes a jump to "address assigned to BACK minus 6", namely, to address "0x100 - 0x6 = 0xFA" when BACK label is 0x100.
Therefore, (1) in the above example can also be described as: BACK: jmp !0xFA.

*

Multiplacation of value of first and second terms.

[Function]

Returns the result of multiplication (product) between the values of the 1st and 2nd terms of an expression.

[Application example]

```
TEN     .set    0x10
        mov     TEN * 3, r12    ; (1)
```

(1) With the .set directive, the value "0x10" is defined in the symbol "TEN".
The expression "TEN * 3" is the same as "0x10 * 3" and returns the value "0x30".
Therefore, (1) in the above expression can also be described as: mov 0x30, r12.

/

Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.

[Function]

Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.

The decimal fraction part of the result will be truncated. If the divisor (2nd term) of a division operation is 0, an error occurs

[Application example]

```
mov     A, #256 / 50          ; (1)
```

(1) The result of the division "256 / 50" is 5 with remainder 6.
The operator returns the value "5" that is the integer part of the result of the division.
Therefore, (1) in the above expression can also be described as: mov A, #5

MOD(%)

Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.

[Function]

Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.
An error occurs if the divisor (2nd term) is 0.
A blank is required before and after the MOD operator.

[Application example]

```
mov     256 % 50, r12   ; (1)
```

(1) The result of the division "256 / 50" is 5 with remainder 6.
The MOD operator returns the remainder 6.
Therefore, (1) in the above expression can also be described as: mov 6, r12.

+sign

Returns the value of the term as it is.

[Function]

Returns the value of the term of an expression without change.

[Application example]

```
FIVE    .set    +5      ; (1)
```

(1) The value "5" of the term is returned without change.
The value "5" is defined in symbol "FIVE" with the .set directive.

-sign

The term value 2 complement is sought.

[Function]

Returns the value of the term of an expression by the two's complement.

[Application example]

```
NO      .set    -1      ; (1)
```

(1) -1 becomes the two's complement of 1.
0000 0000 0000 0000 0000 0000 0000 0001 becomes:
1111 1111 1111 1111 1111 1111 1111 1111
Therefore, with the .set directive, the value "0xFFFFFFFF" is defined in the symbol "NO".

4.1.4 Logic operators

The following logic operators are available.

Operator	Overview
!	Obtains the logical negation (NOT) by each bit.
&	Obtains the logical AND operation for each bit of the first and second term values.
|	Obtains the logical OR operation for each bit of the first and second term values.
^	Obtains the exclusive OR operation for each bit of the first and second term values.

!

Obtains the logical negation (NOT) by each bit.

[Function]

Negates the value of the term of an expression on a bit-by-bit basis and returns the result.
A blank is required between the ! operator and the term.

[Application example]

```
mov32   !0x3, r12       ; (1)
```

(1) Logical negation is performed on "0x3" as follows:
0xFFFFFFFC is returned.
Therefore, (1) can also be described as: mov32 0xFFFFFFFC, r12.

NOT)	0000	0000	0000	0000	0000	0000	0000	0011
	1111	1111	1111	1111	1111	1111	1111	1100

&

Obtains the logical AND operation for each bit of the first and second term values.

[Function]

Performs an AND (logical product) operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.

A blank is required before and after the & operator.

[Application example]

```
mov32   0x6FA & 0xF, r12        ; (1)
```

(1) AND operation is performed between the two values "0x6FA" and "0xF" as follows:
The result "0xA" is returned. Therefore, (1) in the above expression can also be described as:
mov32 0xA, r12.

	0000	0000	0000	0000	0000	0110	1111	1010
&)	0000	0000	0000	0000	0000	0000	0000	1111
	0000	0000	0000	0000	0000	0000	0000	1010

|

Obtains the logical OR operation for each bit of the first and second term values.

[Function]

Performs an OR (Logical sum) operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.

A blank is required before and after the | operator.

[Application example]

```
mov32   0xA | 0b1101, r12       ; (1)
```

(1) OR operation is performed between the two values "0xA" and "0b1101" as follows:
The result "0xF" is returned.
Therefore, (1) in the above expression can also be described as: mov32 0xF, r12.

	0000	0000	0000	0000	0000	0000	0000	1010
|)	0000	0000	0000	0000	0000	0000	0000	1101
	0000	0000	0000	0000	0000	0000	0000	1111

^

Obtains the exclusive OR operation for each bit of the first and second term values.

[Function]

Performs an Exclusive-OR operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result. A blank is required before and after the ^ operator.

[Application example]

```
mov32   0x9A ^ 0x9D, r12        ; (1)
```

(1) XOR operation is performed between the two values "0x9A" and "0x9D" as follows:
The result "0x7" is returned.
Therefore, (1) in the above expression can also be described as: mov32 0x7, r12.

	0000	0000	0000	0000	0000	0000	1001	1010
^)	0000	0000	0000	0000	0000	0000	1001	1101
	0000	0000	0000	0000	0000	0000	0000	0111

4.1.5 Relational operators

The following relational operators are available.

Operator	Overview
==	Compares whether values of first term and second term are equivalent.
!=	Compares whether values of first term and second term are not equivalent.
>	Compares whether value of first term is greater than value of the second.
>=	Compares whether value of first term is greater than or equivalent to the value of the second term.
<	Compares whether value of first term is smaller than value of the second.
<=	Compares whether value of first term is smaller than or equivalent to the value of the second term.
&&	Calculates the logical product of the logical value of the first and second operands.
||	Calculates the logical sum of the logical value of the first and second operands.

==

Compares whether value s of first term and second term are equivalent.

[Function]

Returns ~0 (True) if the value of the 1st term of an expression is equal to the value of its 2nd term, and 0 (False) if both values are not equal.

A blank is required before and after the == operator.

!=

Compares whether values of first term and second term are not equivalent.

[Function]

Returns ~0 (True) if the value of the 1st term of an expression is not equal to the value of its 2nd term, and 0 (False) if both values are equal.

A blank is required before and after the != operator.

>

Compares whether value of first term is greater than value of the second.

[Function]

Returns ~0(True) if the value of the 1st term of an expression is greater than the value of its 2nd term, and 0 (False) if the value of the 1st term is equal to or less than the value of the 2nd term.

A blank is required before and after the > operator.

>=

Compares whether value of first term is greater than or equivalent to the value of the second term.

[Function]

Returns ~0 (True) if the value of the 1st term of an expression is greater than or equal to the value of its 2nd term, and 0 (False) if the value of the 1st term is less than the value of the 2nd term.

A blank is required before and after the >= operator.

<

Compares whether value of first term is smaller than value of the second.

[Function]

Returns ~0 (True) if the value of the 1st term of an expression is less than the value of its 2nd term, and 0 (False) if the value of the 1st term is equal to or greater than the value of the 2nd term.

A blank is required before and after the < operator

<=

Compares whether value of first term is smaller than or equivalent to the value of the second term.

[Function]

Returns ~0 (True) if the value of the 1st term of an expression is less than or equal to the value of its 2nd term, and 0 (False) if the value of the 1st term is greater than the value of the 2nd term.

A blank is required before and after the <= operator.

&&

Calculates the logical product of the logical value of the first and second operands.

[Function]

Calculates the logical product of the logical value of the first and second operands.

Calculates the logical sum of the logical value of the first and second operands.

[Function]

Calculates the logical sum of the logical value of the first and second operands.

4.1.6Shift operators

The following shift operators are available.

Operator	Overview
>>	Obtains only the right-shifted value of the first term which appears in the second term.
<<	Obtains only the left-shifted value of the first term which appears in the second term.

>>

Obtains only the right-shifted value of the first term which appears in the second term.

[Function]

Returns a value obtained by shifting the value of the 1st term of an expression to the right the number of bits specified by the value of the 2nd term.

The sign bit is not shifted.

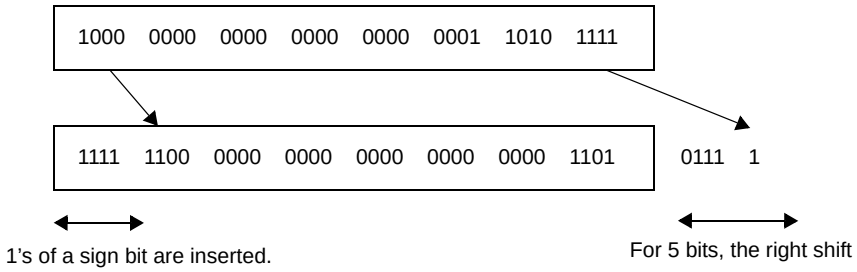
The sign bit is inserted in the high-order bits, the same number of times as the number of bits that were shifted.

If the number of shifted bits is 0, the value of the first term is returned as is. If the number of shifted bits exceeds 31, 0 is returned.

[Application example]

```
mov32   0x800001AF >> 5, r20     ; (1)
```

(1) The value "0x800001AF" is shifted 5 bits to the right, leaving the sign bit.
"0xFC00000D" is forwarded to r20.
Therefore, (1) in the above example can also be described as: mov32 0xFC00000D, r20



<<

Obtains only the left-shifted value of the first term which appears in the second term.

[Function]

Returns a value obtained by shifting the value of the 1st term of an expression to the left the number of bits specified by the value of the 2nd term.

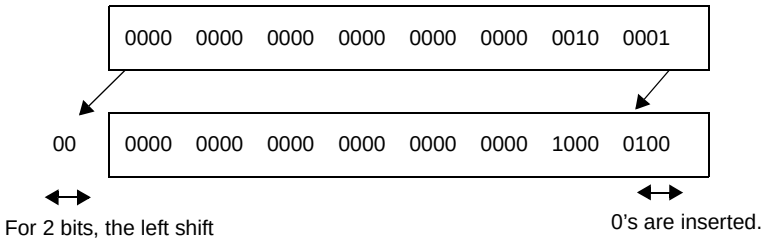
Zeros equivalent to the specified number of bits shifted move into the low-order bits.

If the number of shifted bits is 0, the value of the first term is returned as is. If the number of shifted bits exceeds 31, 0 is returned.

[Application example]

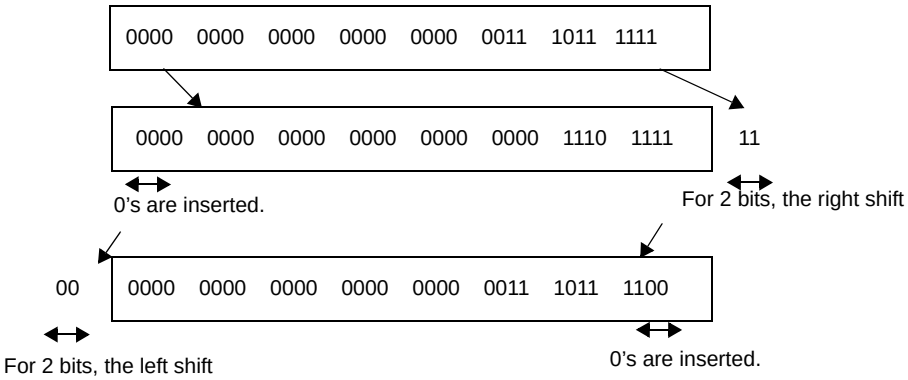
```
mov32   0x21 << 2, r20          ; (1)
```

(1) This operator shifts the value "0x21" to the left by 2 bits.
"0x84" is forwarded to r20.
Therefore, (1) in the above example can also be described as: mov32 0x84, r20




```
mov32   0x3BF >> 2 << 2, r20     ; (2)
```

(2) This operator shifts the value "0x3B" to the right by 2 bits, and shifts to the left by 2 bits.
"0x3BC" is forwarded to r20.
Therefore, (2) in the above example can also be described as: mov32 0x3BC, r20



4.1.7 Byte separation operators

The following byte separation operators are available.

Operator	Overview
HIGH	Returns the high-order 8-bit value of a term.
LOW	Returns the low-order 8-bit value of a term.

HIGH

Returns the high-order 8-bit value of a term.

[Function]

Returns the high-order 8-bit value of a term.
A blank is required between the HIGH operator and the term.

[Application example]

```
mov32   HIGH 0x1234, r12        ; (1)
```

(1) By executing a mov32 instruction, this operator returns the high-order 8-bit value "0x12" of the expression "0x1234".
Therefore, (1) in the above example can also be described as: mov A, #0x12

```
mov32   HIGH P0, r12            ; (2)
```

(2) By executing a mov32 instruction, this operator returns the high-order 8-bit value "0xFF" of the expression P0.
Therefore, (2) in the above example can also be described as: mov 0xFF, r12.

LOW

Returns the low-order 8-bit value of a term.

[Function]

Returns the low-order 8-bit value of a term.
A blank is required between the LOW operator and the term.

[Application example]

```
mov32   LOW 0x1234, r12         ; (1)
```

(1) By executing a mov32 instruction, this operator returns the low-order 8-bit value "0x34" of the expression "0x1234".
Therefore, (1) in the above example can also be described as: mov32 0x34, r12.

4.1.8 2-byte separation operators

The following 2-byte separation operators are available.

Operator	Overview
HIGHW	Returns the high-order 16-bit value of a term.
LOWW	Returns the low-order 16-bit value of a term.
HIGHW1	The value calculated by adding the value at the 15th bit to the uppermost 16 bits of the term.

HIGHW

Returns the high-order 16-bit value of a term.

[Function]

Returns the high-order 16-bit value of a term.
A blank is required between the HIGHW operator and the term.

[Application example]

```
mov32   HIGHW(0x12345678), r12          ; (1)
```

(1) By executing a mov32 instruction, this operator returns the high-order 16-bit value "0x1234" of the expression "0x12345678".
Therefore, (1) in the above example can also be described as: mov32 0x1234, r12.

LOWW

Returns the low-order 16-bit value of a term.

[Function]

Returns the low-order 16-bit value of a term.
A blank is required between the LOWW operator and the term.

[Application example]

```
mov32   LOWW(0x12345678), r12          ; (1)
```

(1) By executing a mov32 instruction, this operator returns the low-order 16-bit value "0x5678" of the expression "0x12345678".
Therefore, (1) in the above example can also be described as: mov32 0x5678, r12.

HIGHW1

The value calculated by adding the value at the 15th bit to the uppermost 16 bits of the term.

[Function]

The value calculated by adding the value at the 15th bit to the uppermost 16 bits of the term.
A blank is required between the HIGHW1 operator and the term.

[Application example]

```
mov32   HIGHW1(0x12345678), r12         ; (1)
```

(1) Given the value 0x12345678, a mov32 instruction adds the value at the 15th bit (1) to the top 16 bits (0x1234), returning the value 0x1235.
Therefore, (1) in the above example can also be described as: mov32 0x1235, r12.

4.1.9 Special operators

The following special operators are available.

Operator	Overview
DATAPOS	Obtains the address part of a bit symbol.
BITPOS	Obtains the bit part of a bit symbol.

DATAPOS

Obtains the address part of a bit symbol.

[Function]

Returns the address portion of a bit symbol.

[Application example]

```
mov32   DATAPOS( DNFA2NFEN2 ), r10      ; (1)
mov32   BITPOS( DNFA2NFEN2 ), r12
clr1    r12, [r10]
```

(1) "DATAPOS DNFA2NFEN2" represents "DATAPOS 0xFF41020C.2", and "0xFF41020C" is returned. Therefore, in the above example can also be described as: mov32 0xFF41020C, r10.

BITPOS

Obtains the bit part of a bit symbol.

[Function]

Returns the bit portion (bit position) of a bit symbol.

[Application example]

```
mov32   DATAPOS( DNFA2NFEN2 ), r10
mov32   BITPOS( DNFA2NFEN2 ), r12       ; (1)
clr1    r12, [r10]
```

(1) "BITPOS DNFA2NFEN2" represents "BITPOS 0xFF41020C.2", and "2" is returned. Therefore, in the above example can also be described as: mov32 2, r12.

4.1.10 Other operator

The following operators is also available.

Operator	Overview
( )	Prioritizes the calculation within ( ).

( )

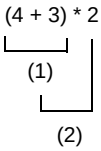
Prioritizes the calculation within ( ).

[Function]

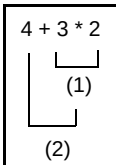
Causes an operation in parentheses to be performed prior to operations outside the parentheses.
This operator is used to change the order of precedence of other operators.
If parentheses are nested at multiple levels, the expression in the innermost parentheses will be calculated first.

[Application example]

```
mov     A, #(4 + 3) * 2
```



Calculations are performed in the order of expressions (1), (2) and the value "14" is returned as a result.
If parentheses are not used,



Calculations are performed in the order (1), (2) shown above, and the value "10" is returned as a result.
See "Table 4-4. Operator Precedence Levels", for the order of precedence of operators.

4.1.11 Restrictions on operations

An expression consists of a "constant", "symbol", "label reference", "operator", and "parentheses".It indicates a value consisting of these elements. The expression distinguishes between Absolute expression and Relative expressions.

(1) Absolute expression

An expression indicating a constant is called an "absolute expression". An absolute expression can be used when an operand is specified for an instruction or when a value etc. is specified for a directive. An absolute expression usually consists of a constant or symbol. The following format is treated as an absolute expression.

(a) Constant expression

If a reference to a previously defined symbol is specified, assumes that the constant of the value defined for the symbol has been specified. Therefore, a defined symbol reference can be used in a constant expression.

Example

```
sym1    .set    0x10    --Define symbol sym1
       mov     sym1, r1 --sym1, already defined, is treated as a constant expression.
```

(b) Symbol

The expressions related to symbols are the following ("±" is either "+" or "-").

- Symbol
- Symbol ± constant expression
- Symbol - symbol
- Symbol - symbol ± constant expression

A "symbol" here means an undefined symbol reference at that point. If a reference to a previously defined symbol is specified, assumes that the "constant" of the value defined for the symbol has been specified.

Example

```
add     SYM1 + 0x100, r11   --SYM1 is an undefined symbol at this point
SYM1    .set 0x10           --Defines SYM1
```

(c) Label reference

The following expressions are related to label reference ("±" is either "+" or "-").

- Label reference - label reference
- Label reference - label reference ± constant expression

Here is an example of an expression related to a label reference.

Example

```
mov     $label1 - $label2, r11
```

A "reference to two labels" as shown in this example must be referenced as follows.

- The same section has a definition in the specified file.
- Same reference method (such as $label and $label, and #label and #label)

When not meeting these conditions, a message is output, and assembly is canceled.

However, if a reference to the absolute address of a label not having a definition in the specified file is specified as label reference on one side of "- label reference" in an "expression related to label reference", it is assumed that the same reference method as that of the label on the other side is used, because of the current organization of the assembler. Note that an absolute expression in this format cannot be specified for a branch instruction. If such an expression is specified, a message is output, and assembly is canceled.

(2) Relative expressions

An expression indicating an offset from a specific address[Note 1] is called a "relative expression". A relative expression is used to specify an operand by an instruction or to specify a value by data definition directive. A relative expression usually consists of a label reference. The following format[Note 2] is treated as an relative expression.

Notes 1. This address is determined when the linker is executed. Therefore, the value of this offset may also be determined when the linker is executed.

2. The absolute value system and the relative value system can regard an expression in the format of "-symbol + label reference", as being an expression in the format of "label reference - symbol," but it cannot regard an expression in the format of "label reference - (+symbol)" as being an expression in the format of "label reference - symbol". Therefore, use parentheses "( )" only in constant expressions.

(a) Label reference

The following expressions are related to label reference ("±" is either "+" or "-").

- Label reference
- Label reference + constant expression
- Label reference - symbol
- Label reference - symbol ± constant expression

Here is an example of an expression related to a label reference.

Example

```
add     #labe11 + 0x10, r10
add     #label2 - SIZE, r10
SIZE    .set 0x10
```

4.1.12 Identifiers

An identifier is a name used for symbols, labels, macros etc.

Identifiers are described according to the following basic rules.

- Identifiers consist of alphanumeric characters and symbols that are used as characters (?,@,_)
  However, the first character cannot be a number (0 to 9).
- Reserved words cannot be used as identifiers.
  With regard to reserved words, see "4.5 Reserved Words".
- The assembler distinguishes between uppercase and lowercase.

4.2 Directives

This chapter explains the directives.

Directives are instructions that direct all types of instructions necessary for the assembler.

4.2.1 Outline

Instructions are translated into object codes (machine language) as a result of assembling, but directives are not converted into object codes in principle.

Directives contain the following functions mainly:

- To facilitate description of source programs
- To initialize memory and reserve memory areas
- To provide the information required for assemblers and linkers to perform their intended processing

The following table shows the types of directives.

Table 4-5. List of Directives

Type	Directives
Section definition directives	.cseg, .dseg, .org, .vseg
Symbol definition directives	.set, .file, .func
Data definition, area reservation directives	.db, .db2/.dhw, .dshw, .db4/.dw, .db8/.ddw, .float, .double, .ds, .align
External definition, external reference directives	.public, .extern, .comm
Macro directives	.macro, .local, .rept, .irp, .exitm, .exitma, .endm

The following sections explain the details of each directive.

In the description format of each directive, "[ ]" indicates that the parameter in square brackets may be omitted from specification, and "..." indicates the repetition of description in the same format.

4.2.2 Section definition directives

A section is a block of routines or data of the same type. A "section definition directive" is a directive that declares the start or end of a section.

Sections are the unit of allocation in the linker.

Example

```
.cseg
:
.dseg
:
```

Two sections with the same section name must have the same relocation attribute. Consequently, multiple sections with differing relocation attributes cannot be given the same section name. If two sections with the same section name have different relocation attributes, an error will occur, and the directive will be ignored.

Sections in a single source program file with the same relocation attribute and section name will be processed as a single continuous section in the assembler.

If the sections are broken into separate source program files, then they will be processed by the linker.

Section names cannot be referenced as symbols.

The following section definition directives are available.

Table 4-6. Section Definition Directives

Directive	Overview
.cseg	Indicates to the assembler the starting of a code section (located in ROM area)
.dseg	Indicates to the assembler the start of a data section (located in RAM area)
.org	Advances the value of the location counter
.vseg	Indicates to the assembler the start of a section for debug information

.cseg

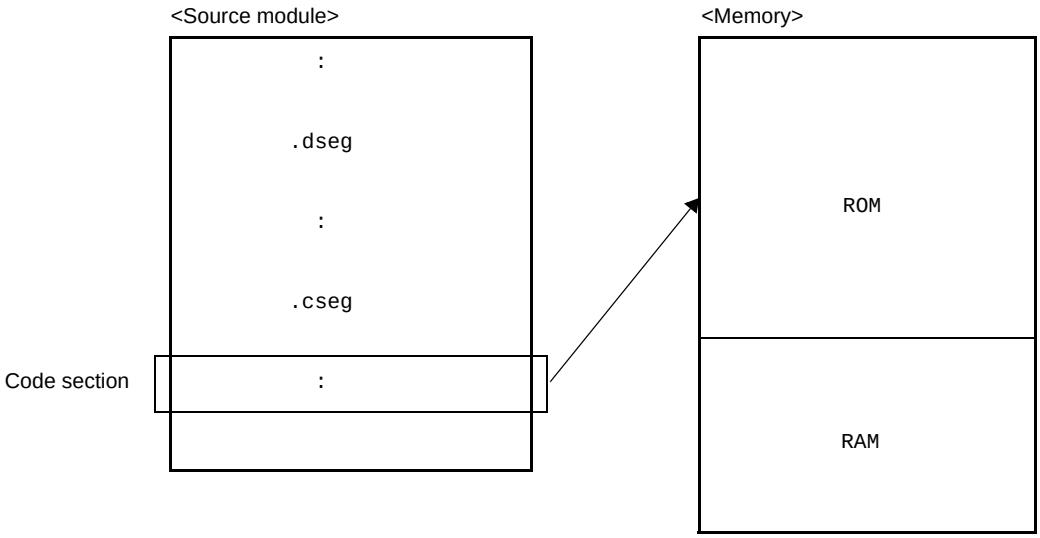
Indicate to the assembler the start of a code section (located in ROM area).

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*section-name*]	.cseg	[*relocation-attribute*]	[; *comment*]

[Function]

- The .cseg directive indicates to the assembler the start of a code section.
- All instructions described following the .cseg directive belong to the code section until it comes across a section definition directives (.cseg, .dseg or .org), and finally those instructions are located within a ROM address after being converted into machine language.



[Use]

- The .cseg directive is used to describe instructions, .db, .dw directives, etc. in the code section defined by the .cseg directive.
- Description of one functional unit such as a subroutine should be defined as a single code section.

[Description]

- A relocation attribute defines a range of location addresses for a code section.

Table 4-7. Relocation Attributes of .cseg

Relocation Attribute	Description Format	Explanation
OPT_BYTE	OPT_BYTE	It is a user option byte and on-chip debugging specific attribute. Not specify except user option byte and on-chip debugging. Tells the assembler to locate the specified section within the address range 0x7A to 0x7F (V850E1, V850E2 core). In the device with an instruction set of V850E2V3, it can't be specified.
SECUR_ID	SECUR_ID	It is a security ID specific attribute. Not specify except security ID. Tells the assembler to locate the specified section within the address range 0x70 to 0x79 (V850E1, V850E2 core). In the device with an instruction set of V850E2V3, it can't be specified.
TEXT	TEXT	Allocates the program. This is a reserved section with section name ".text", section type "PROGBITS", and section attribute "AX". It is assumed that two TEXT sections are specified before an assembly language source program in an assembly language source file (for example, if ".dw1" is specified before a section definition directive, this will be allocated to a ".text" section). Note, however, that if the ".text" section is not explicitly specified, and the label definition, instruction, location counter control directive, or secure-area directive of the TEXT section specified by default is not specified, then no ".text" section will be generated.
CONST	CONST	This section is for constant (read-only) data. It allocates a memory range consisting of r0 and 2 instructions, and referenced using 32-bit displacement. This is a reserved section with section name ".const", section type "PROGBITS", and section attribute "A".
SCONST	SCONST	This section is for constant (read-only) data. It allocates a memory range (up to 32 Kbytes, in the positive direction from r0), referenced with 1 instructions using r0 and 16-bit displacement. This is a reserved section with section name ".sconst", section type "PROGBITS", and section attribute "A".

- If no relocation attribute is specified for the code segment, the assembler will assume that "TEXT" has been specified.
- If the size of a section exceeds the size of its area, an error will occur. If this happens, the location counter will be advanced, and assembly will continue.

- By describing a section name in the symbol field of the .cseg directive, the code section can be named. If no section name is specified for a code section, the assembler will automatically give a default section name to the code section.
  The default section names of the code sections are shown below.

Relocation Attribute	Default Section Name
OPT_BYTE[Note]	OPTION_BYTES
SECUR_ID[Note]	SECURITY_ID
TEXT	.text
CONST	.const
SCONST[Note]	.sconst

Note A specification possible section name is only a default section name in these relocation attributes.

- If two or more code sections have the same relocation attribute, these code sections may have the same section name.
  These same-named code sections are processed as a single code section within the assembler.
  An error occurs if the same-named sections differ in their relocation attributes. Therefore, the number of the same-named sections for each relocation attribute is one.
- Description of a code section can be divided into units. The same relocation attribute and the samenamed code section described in one module are handled by the assembler as a series of sections.
- The same-named data sections in two or more different modules can be specified only when their relocation attributes are SECUR_ID, and are combined into a single data section at linkage.
- No section name can be referenced as a symbol.
- Specify user option byte and on-chip debugging by using OPT_BYTE.
  When the user option byte is not specified for the chip having the user option byte feature, define a default section of "OPT_BYTE" to each address and set the initial value by reading from a device file.
- In the case of multi-core, the assembler will automatically assign default section names for each relocation attribute in code sections without section names specified.
  The default section names are shown below.
  - CSEG default section names (for "-Xmulti=pe*n*")

Relocation Attribute	Default Section Name
TEXT	.text.pe*n*
CONST	.const.pe*n*
SCONST	.const.pe*n*

  - CSEG default section names (for "-Xmulti=cmn")

Relocation Attribute	Default Section Name
TEXT	.text.cmn
CONST	.const.cmn
SCONST	.const.cmn

.dseg

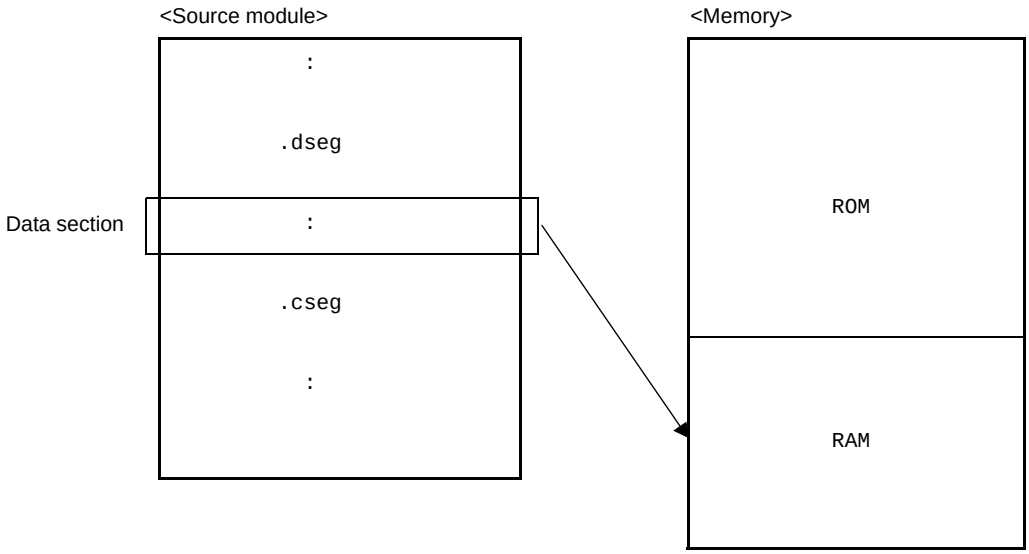
Indicate to the assembler the start of a data section (located in RAM area).

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*section-name*]	.dseg	[*relocation-attribute*]	[; *comment*]

[Function]

- The .dseg directive indicates to the assembler the start of a data section.
- A memory following the .dseg directive belongs to the data section until it comes across a section definition directives (.cseg, .dseg or .org), and finally it is reserved within the RAM address.



[Use]

- The .ds directive is mainly described in the data section defined by the .dseg directive.
  Data sections are located within the RAM area. Therefore, no instructions can be described in any data section.
- In a data section, a RAM work area used in a program is reserved by the .ds directive and a label is attached to each work area. Use this label when describing a source program.
  Each area reserved as a data section is located by the linker so that it does not overlap with any other work areas on the RAM (stack area, and work areas defined by other modules).

[Description]

- A relocation attribute defines a range of location addresses for a data section.
  The relocation attributes available for data sections are shown below.

Table 4-8. Relocation Attributes of DSEG

Relocation Attribute	Description Format	Explanation
BSS	BSS	Allocates a memory range consisting of gp and 2 instructions without an initial value, and referenced using 32-bit displacement.
DATA	DATA	Allocates a memory range consisting of gp and 2 instructions with an initial value, and referenced using 32-bit displacement.
SBSS	SBSS	Allocates a memory range (up to 64 Kbytes, combined with SDATA section), referenced with 1 instructions using gp and 16-bit displacement, not having an initial value.
SDATA	SDATA	Allocates a memory range (up to 64 Kbytes, combined with SDATA section), referenced with 1 instructions using gp and 16-bit displacement, having an initial value.
SEBSS	SEBSS	Allocates the high-level address portion of the memory range (up to 32 Kbytes in the negative direction from ep) (the size of the SEDATA section) referenced with 1 instructions using ep and 16-bit displacement, not having an initial value.
SEDATA	SEDATA	Allocates the high-level address portion of the memory range (up to 32 Kbytes in the negative direction from ep) (the size of the SEDATA section) referenced with 1 instructions using ep and 16-bit displacement, having an initial value.
SIBSS	SIBSS	Allocates the high-level address portion of the memory range (up to 32 Kbytes in the positive direction from ep) (the size of the SIBSS and TI* sections) referenced with 1 instructions using ep and 16-bit displacement, not having an initial value.
SIDATA	SIDATA	Allocates the high-level address portion of the memory range (up to 32 Kbytes in the positive direction from ep) (the size of the SIBSS and TI* sections) referenced with 1 instructions using ep and 16-bit displacement, having an initial value.
TIBSS	TIBSS	This assumes allocation in internal RAM without initial values, and ep relative access using sld/sst instructions. If TIDATA.BYTE, TIBSS.BYTE, TIDATA.WORD, TIBSS.WORD, and TIDATA are not used, then TIBSS is allocated to the address indicated by ep. If TIDATA.BYTE, TIBSS.BYTE, TIDATA.WORD, TIBSS.WORD, or TIDATA is used, then TIBSS is allocated to the address indicated by ep, with the size of TIDATA.BYTE/TIBSS.BYTE/TIDATA.WORD/TIBSS.WORD/TIDATA added. The scope accessed by sld/sst instructions differs depending on the size of the data. For this reason, we recommend placing byte data in a TIBSS.BYTE section, and data larger than byte data in a TIBSS.WORD section. Use a TIBSS section if you do not need to consider the access area in fine detail like this.
TIBSS.BYTE	TIBSS.BYTE	This assumes allocation in internal RAM, and ep relative access using sld/sst instructions. When accessing byte data, an sld/sst instruction can access areas up to 128 bytes. For this reason, we recommend placing byte data with no initial value in a TIBSS.BYTE section.

Relocation Attribute	Description Format	Explanation
TIBSS.WORD	TIBSS.WORD	This assumes allocation in internal RAM, and ep relative access using sld/sst instructions. When accessing data larger than byte data, an sld/sst instruction can access areas up to 256 bytes. For this reason, we recommend placing data with no initial value that is larger than byte data in a TIBSS.WORD section.
TIDATA	TIDATA	This assumes allocation in internal RAM with initial values, and ep relative access using sld/sst instructions. If TIDATA.BYTE and TIDATA.WORD are not used, then TIDATA is allocated to the address indicated by ep. If TIDATA.BYTE or TIDATA.WORD is used, then TIDATA is allocated to the address indicated by ep, with the size of TIDATA.BYTE/ TIDATA.WORD added. The scope accessed by sld/sst instructions differs depending on the size of the data. For this reason, we recommend placing byte data in a TIDATA.BYTE section, and data larger than byte data in a TIDATA.WORD section. Use a TIDATA section if you do not need to consider the access area in fine detail like this.
TIDATA.BYTE	TIDATA.BYTE	This assumes allocation in internal RAM, and ep relative access using sld/sst instructions. When accessing byte data, an sld/sst instruction can access areas up to 128 bytes. For this reason, we recommend placing byte data with initial value in a TIDATA.BYTE section.
TIDATA.WORD	TIDATA.WORD	This assumes allocation in internal RAM, and ep relative access using sld/sst instructions. When accessing data larger than byte data, an sld/sst instruction can access areas up to 256 bytes. For this reason, we recommend placing data with no initial value that is larger than byte data in a TIDATA.WORD section.

- If no relocation attribute is specified for the code segment, the assembler will assume that "DATA" has been specified.
- If the size of a section exceeds the size of its area, an error will occur. If this happens, the location counter will be advanced, and assembly will continue.
- Machine language instructions cannot be described in a data section. If described, an error is output and the line is ignored.
- By describing a section name in the symbol field of the .dseg directive, the data section can be named. If no section name is specified for a data section, the assembler automatically gives a default section name.
  The default section names of the data sections are shown below.

Relocation Atribute	Default Section Name
BSS	.bss
DATA	.data
SBSS	.sbss
SDATA	.sdata
SEBSS[Note]	.sebss
SEDATA[Note]	.sedata

Relocation Atribute	Default Section Name
SIBSS[Note]	.sibss
SIDATA[Note]	.sidata
TIBSS[Note]	.tibss
TIBSS.BYTE[Note]	.tibss.byte
TIBSS.WORD[Note]	.tibss.word
TIDATA[Note]	.tidata
TIDATA.BYTE[Note]	.tidata.byte
TIDATA.WORD[Note]	.tidata.word

Note A specification possible section name is only a default section name in these relocation attributes.

- If two or more data sections have the same relocation attribute, these data sections may have the same section name.
  These sections are processed as a single data section within the assembler.
- Description of a data section can be divided into units. The same relocation attribute and the same-named code section described in one module are handled by the assembler as a series of sections.
- An error occurs if the same-named sections differ in their relocation attributes. Therefore, the number of the same-named sections for each relocation attribute is one.
- No section name can be referenced as a symbol.
- They are as follows for multi-core. [V850E2V3]
  - If the "-Xmulti=pen" option is specified
    For each core's program, they can be allocated to data sections of all relocation attributes in the same way as a single-core program.
  - If the "-Xmulti=cmn" option is specified
    Only a relocation attribute DATA/BSS section can be allocated to the common module's data section. Specifying other than a relocation attribute DATA/BSS section will cause an error.
    In the case of multi-core, the assembler will automatically assign default section names for each relocation attribute in data sections without section names specified.
    The default section names are shown below.
    - DSEG default section names (for "-Xmulti=pe*n*")

Relocation Attribute	Default Section Name
BSS	.bss.pe*n*
DATA	.data.pe*n*
SBSS	.sbss.pe*n*
SDATA	.sdata.pe*n*
SEBSS	.sebss.pe*n*
SEDATA	.sedata.pe*n*
SIBSS	.sibss.pe*n*
SIDATA	.sidata.pe*n*
TIBSS	.tibss.pe*n*
TIBSS.BYTE	.tibss.byte.pe*n*
TIBSS.WORD	.tibss.word.pe*n*

Relocation Attribute	Default Section Name
TIDATA	.tidata.pe*n*
TIDATA.BYTE	.tidata.byte.pe*n*
TIDATA.WORD	.tidata.word.pe*n*

- DSEG default section names (for "-Xmulti=cmn")

Relocation Attribute	Default Section Name
BSS	.bss.cmn
DATA	.data.cmn

.org

Advances the value of the location counter.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
	.org	*value*	

[Function]

Advances the value of the location counter for the current section, to the value specified by the operand.

[Description]

Advances the value of the location counter for the current section, specified by the previously specified section definition directive, to the value(Less than 2^{31}) specified by the operand. If a hole results from advancing the value of the location counter, it is filled with 0.

[Example]

Advances the location counter value 16 bytes.

```
.org    16
```

[Caution]

- If a value that is smaller than the current value of the location counter is specified, the assembler outputs the message then stops assembling.
- If this directive is used in the sdata-attribute section, valid information may not be obtained when a guideline value for determining the size of the data to be allocated to the sdata/sbss-attribute section is displayed (by using the -Xsdata_info option).
- This directive merely advances the value of the location counter in a specified file for the section. It does not specify either an absolute address[Note 1] or an offset in a section[Note 2].

Notes 1. Offset from address 0 in a linked object module file.

2. Offset from the first address of the section (output section) to which that section is allocated in a linked object module file.

.vseg

Indicate to the assembler the start of a section for debug information.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*section-name*]	.vseg	[*comment*]	[; *comment*]

[Function]

- The ".vseg" directive tells the assembler to start a section for debugging information.
  Do not change this section, because it is for debugging information.

4.2.3 Symbol definition directives

Symbol definition directives specify symbols for the data that is used when writing to source modules. With these, the data value specifications are made clear and the details of the source module are easier to understand.

Symbol definition directives indicate the symbols of values used in the source module to the assembler.

The following symbol definition directives are available.

Table 4-9. Symbol Definition Directives

Directive	Overview
.set	Defines a symbol
.file	Generates a symbol table entry
.func	Generates a symbol table entry

.set

Defines a symbol.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
symbol	.set	*value*	

[Function]

Defines a symbol having a symbol name specified by the symbol field and a value (Integer value) specified by the operand field.

[Description]

Defines a symbol having a symbol name specified by the symbol field and a value (Integer value) specified by the operand field. If the .set directive is specified for a given symbol more than once within a single assembler source file, reference to that symbol will have the following value, depending on the position of that reference.

- If the reference appears between the beginning of the file and the first .set directive for that symbol
  Value specified with the last .set directive for that symbol.
- If the reference does not appear between a certain .set directive and the next .set directive, or if there is no subsequent .set directive, between the first .set directive and the end of the assembler source file
  Value specified by that .set directive.

[Example]

Defines the value of symbol sym1 as 0x10.

```
.set    sym1, 0x10
```

[Caution]

- Any label reference or undefined symbol reference must not be used to specify a value.
  Otherwise, the assembler outputs the following message then stops assembling.

  E0550203: illegal expression (*string*)

- If a label name, a macro name defined by the .macro directive, or a symbol of the same name as a formal parameter of a macro is specified, the assembler outputs the following message and stops assembling.

  E0550212: symbol already define as *string*

.file

Generates a symbol table entry (FILE type).

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
	.file	"*file-name*"	[; *comment*]

[Function]

- Generates a symbol table entry[Note] having a file name specified by the operand and type FILE when an object module file is generated. If this directive does not exist in the input source file, it is assumed that ".file"input file name" has been specified, and a symbol table entry with the input file name and type FILE is generated.

Note The binding class is LOCAL.

[Use]

- The ".file" directive is compiler debugging information.

[Description]

- The file name is written with the specified image.
- This is the name of the C source program file that the compiler outputs.

.func

Generates a symbol table entry (FUNC type).

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
	.func	*function-name, function-size, stack-size*	[; *comment*]

[Function]

- Generates a symbol table entry of type FUNC when an object module file is generated.

[Use]

- The ".func" directive is compiler debugging information.

[Description]

- The first operand is the C-language function name output by the compiler; the second operand is an expression indicating that function; and the third operand is a number indicating the stack size of the function.
- This is the function information of the C source program that the compiler outputs.

4.2.4 Data definition, area reservation directives

The data definition directive defines the constant data used by the program.

The defined data value is generated as object code.

The area reservation directive secures the area for memory used by the program.

The following data definition and partitioning directives are available.

Table 4-10. Data Definition, Area Reservation Directives

Directive	Overview
.db	Initialization of byte area
.db2/.dhw	Initialization of 2-byte area
.dshw	Initializes a 2-byte area with the specified value, right-shifted one bit
.db4/.dw	Initialization of 4-byte area
.db8/.ddw	Initialization of 8-byte area
.float	Initialization of 4-byte area
.double	Initialization of 8-byte area
.ds	Secures the memory area of the number of bytes specified by operand
.align	Aligns the value of the location counter

.db

Initialization of byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.db	(*absolute-expression*)	[; *comment*]
	or		
[*label*:]	.db	*expression*[, ...]	[; *comment*]
	or		
[*label*:]	.db	*"Character string constants"*	[; *comment*]

[Function]

- The .db directive tells the assembler to initialize a byte area.
  The number of bytes to be initialized can be specified as "size".
- The .db directive also tells the assembler to initialize a memory area in byte units with the initial value(s) specified in the operand field.

[Use]

- Use the .db directive when defining an expression or character string used in the program.

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of bytes with the value "0".

(b) An absolute expression can be described as a size. If the size description is illegal, the CX outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be 1-byte data. Therefore, the value of the operand must be in the range of 0x0 to 0xFF. If the value exceeds 1 byte, the assembler will use only lower 1 byte of the value as valid data.

(b) Character string constants

If the first operand is surrounded by corresponding double quotes ("), then it is assumed to be a string constant.

If a character string constants is described as the operand, an 8-bit ASCII code will be reserved for each character in the string.

- The .db directive cannot be described in a bit section.
- Two or more initial values may be specified within a statement line of the .db directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.
- If the relocation attribute of the section containing the .db directive is BSS or SBSS, then an error is output, because initial values cannot be specified.

[Example]

```
                .cseg   text
WORK1:          .db     (1)                 ; (1)
WORK2:          .db     (2)                 ; (1)
                .cseg   text
MASSAG:         .db     "ABCDEF"            ; (2)
DATA1:          .db     0xA, 0xB, 0xC       ; (3)
DATA2:          .db     (3 + 1)             ; (4)
DATA3:          .db     "AB" + 1            ; (5)   <- Error
```

(1) Because the size is specified, the assembler will initialize each byte area with the value "0".

(2) A 6-byte area is initialized with character string 'ABCDEF'

(3) A 3-byte area is initialized with "0xA, 0xB, 0xC".

(4) A 4-byte area is initialized with "0x0".

(5) This description occurs in an error.

.db2/.dhw

Initialization of 2-byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.db2	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.db2	*expression*[, ...]	[; *comment*]
		or	
[*label*:]	.dhw	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.dhw	*expression*[, ...]	[; *comment*]

[Function]

- The .db2 and .dhw directive tells the assembler to initialize 2-byte area.
  The number of 2-byte data to be initialized can be specified as "size".
- The .db2 and .dhw directive also tells the assembler to initialize a memory area in 2-byte units with the initial value(s) specified in the operand field.

[Use]

- Use the .db2 and .dhw directive when defining a 2-byte numeric constant such as an address or data used in the program.

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of 2-byte with the value "0".

(b) An absolute expression can be described as a size. If the size description is illegal, the CX outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be 2-byte data. Therefore, the value of the operand must be in the range of 0x0 to 0xFFFF. If the value exceeds 2-byte, the assembler will use only lower 2-byte of the value as valid data. No character string constants can be described as an initial value.

- The .db2 and .dhw directive cannot be described in a bit section.

- If the relocation attribute of the section containing the .db2 and .dhw directive is BSS or SBSS, then an error is output, because initial values cannot be specified.
- Two or more initial values may be specified within a statement line of the .db2 and .dhw directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

.dshw

Initializes a 2-byte area with the specified value, right-shifted one bit.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.dshw	*expression*[, ...]	[; *comment*]

[Function]

- Initializes a 2-byte area with the specified value, right-shifted one bit.

[Description]

- The value is secured as 2-byte data, as the value of the expression right-shifted 1 bit.
- The .dshw directive cannot be described in a bit section.
- If the relocation attribute of the section is BSS or SBSS, then an error is output, because the .dshw directive cannot be described.
- It is possible to code an absolute expression in the operand expression.
- The value of the expression, right-shifted one bit, must be in the range 0x0 to 0xFFF. In other cases, the data from the lower two bytes will be secured.
- Any number of expressions may be specified on a single line, by separating them with commas.
- It is not possible to code string constants in the operand.

.db4/.dw

Initialization of 4-byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.db4	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.db4	*expression*[, ...]	[; *comment*]
		or	
[*label*:]	.dw	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.dw	*expression*[, ...]	[; *comment*]

[Function]

- The .db4 and .dw directive tells the assembler to initialize 4-byte area.
  The number of 4-byte data to be initialized can be specified as "size".
- The .db4 and .dw directive also tells the assembler to initialize a memory area in 4-byte units with the initial value(s) specified in the operand field.

[Use]

- Use the .db4 and .dw directive when defining a 4-byte numeric constant such as an address or data used in the program.

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of 4-byte with the value "0".

(b) An absolute expression can be described as a size. If the size description is illegal, the CX outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be 4-byte data. Therefore, the value of the operand must be in the range of 0x0 to 0xFFFFFFFF. If the value exceeds 4-byte, the assembler will use only lower 2-byte of the value as valid data.
No character string constants can be described as an initial value.

- The .db4 and .dw directive cannot be described in a bit section.

- Two or more initial values may be specified within a statement line of the .db4 and .dw directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.
- If the relocation attribute of the section containing the .db4 and .dw directive is BSS or SBSS, then an error is output, because initial values cannot be specified.

.db8/.ddw

Initialization of 8-byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.db8	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.db8	*absolute-expression*[, ...]	[; *comment*]
		or	
[*label*:]	.ddw	(*absolute-expression*)	[; *comment*]
		or	
[*label*:]	.ddw	*absolute-expression*[, ...]	[; *comment*]

[Function]

- The .db8 and .ddw directive tells the assembler to initialize 8-byte area.
  The number of 8-byte data to be initialized can be specified as "size".
- The .db8 and .ddw directive also tells the assembler to initialize a memory area in 8-byte units with the initial value(s) specified in the operand field.

[Use]

- Use the .db8 and .ddw directive when defining a 8-byte numeric constant such as an address or data used in the program.

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of 8-byte with the value "0".

(b) An absolute expression can be described as a size. If the size description is illegal, the CX outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be 8-byte data. Therefore, the value of the operand must be in the range of 0x0 to 0xFFFFFFFFFFFFFFFF. If the value exceeds 8-byte, the assembler will use only lower 8-byte of the value as valid data.
No character string constants can be described as an initial value.

- The .db8 and .ddw directive cannot be described in a bit section.

- If the relocation attribute of the section is BSS or SBSS, then an error is output, because the .db8 and .ddw directive cannot be described.
- Two or more initial values may be specified within a statement line of the .db8 and .ddw directive.

.float

Initialization of 4-byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.float	*absolute-expression*[, ...]	[; *comment*]

[Function]

- The .float directive tells the assembler to initialize 4-byte area.
- The .float directive also tells the assembler to initialize a memory area in 4-byte units with the absolute-expression specified in the operand field.

[Description]

- The value of the absolute expression is secured as a single-precision floating-point number. Consequently, the value of the expression must be between 1.40129846e-45 and 3.40282347e+3. In other cases, the data from the lower four bytes will be secured as a single-precision floating-point number.
- The .float directive cannot be described in a bit section.
- If the relocation attribute of the section is BSS or SBSS, then an error is output, because the .float directive cannot be described.
- Two or more absolute-expression may be specified within a statement line of the .float directive.

.double

Initialization of 8-byte area.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.double	*absolute-expression*[, ...]	[; *comment*]

[Function]

- The .double directive tells the assembler to initialize 8-byte area.
- The .double directive also tells the assembler to initialize a memory area in 8-byte units with the initial value(s) specified in the operand field.

[Description]

- The value of the absolute expression is secured as a double-precision floating-point number. Consequently, the value of the expression must be between 4.9406564584124654e-324 and 1.7976931348623157e+308. In other cases, the data from the lower eight bytes will be secured as a double-precision floating-point number.
- The .double directive cannot be described in a bit section.
- If the relocation attribute of the section is BSS or SBSS, then an error is output, because the .double directive cannot be described.
- Two or more absolute-expression may be specified within a statement line of the .double directive.

.ds

Secures the memory area of the number of bytes specified by operand.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.ds	(*absolute-expression*)[, ...]	[; *comment*]
		or	
[*label*:]	.ds	*absolute-expression*	[; *comment*]

[Function]

- The .ds directive tells the assembler to reserve a memory area for the number of bytes specified in the operand field.

[Use]

- The .ds directive is mainly used to reserve a memory (RAM) area to be used in the program.
  If a label is specified, the value of the first address of the reserved memory area is assigned to the label. In the source module, this label is used for description to manipulate the memory.

[Description]

- If a value in the first operand is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.
- The first operand is a size specification. If a second operand is also specified, then it will be treated as the initial value for that value.

(1) With size specification:

(a) If a size is specified in the operand, then if an initial value is specified, the compiler will fill the specified number of bytes with the specified value; otherwise, it will fill that number of bytes with zeroes ("0"). Note, however, that no area will be secured if the specified number of bytes is 0.

(b) An absolute expression can be described as a size. If the size description is illegal, the CX outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be byte data. Therefore, the value of the operand must be in the range of 0x0 to 0xFF. If the value exceeds byte, the assembler will use only lower 1-byte of the value as valid data.

- The .ds directive cannot be described in a bit section.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.
- If the relocation attribute of the section containing this directive is BSS or SBSS, then an error is output and this directive is ignored, because initial values cannot be specified.

.align

Aligns the value of the location counter.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.align	*line-condition*[, *absolute-expression*]	[; *comment*]

[Function]

- Aligns the value of the location counter.

[Description]

- Aligns the value of the location counter for the current section, specified by the previously specified section definition directive under the alignment condition specified by the first operand. If a hole results from aligning the value of the location counter, it is filled with the value of the absolute expression specified by the second operand, or with the default value of 0.
- The .align directive cannot be described in a bit section.
- Specify an even number of 2 or more, but less than 2^{31}, as the alignment condition. Otherwise, the CX outputs the error message then stops assembling.
- The value of the second operand's absolute-expression must be in the range of 0x0 to 0xFF. If the value exceeds range of 0x0 to 0xFF, the assembler will use only lower 1-byte of the value as valid data.
- This directive merely aligns the value of the location counter in a specified file for the section. It does not align an address after arrangement.

4.2.5 External definition, external reference directives

External definition, external reference directives clarify associations when referring to symbols defined by other modules.

This is thought to be in cases when one program is written that divides module 1 and module 2. In cases when you want to refer to a symbol defined in module 2 in module 1, there is nothing declared in either module and and so the symbol cannot be used. Due to this, there is a need to display "I want to use" or "I don't want to use" in respective modules.

An "I want to refer to a symbol defined in another module" external reference declaration is made in module 1. At the same time, a "This symbol may be referred to by other symbols" external definition declaration is made in module 2.

This symbol can only begin to be referred to after both external reference and external definition declarations in effect.

External definition, external reference directives are used to to form this relationship and the following instructions are available.

Table 4-11. External Definition, External Reference Directives

Directive	Overview
.public	Declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module
.extern	Declares to the linker that a symbol (other than bit symbols) in another module is to be referenced in this module
.comm	Declares an undefined external symbol

.public

Declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.public	*label-name*[, *size*]	[; *comment*]

[Function]

- The PUBLIC directive declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module.

[Use]

- When defining a symbol to be referenced from another module, the .public directive must be used to declare the symbol as an external definition.

[Description]

- A label with the same name as the one specified by the first operand is declared as an external label[Note].
  Note that if a second operand was specified, this specifies the size of the data indicated by that label.

  Note This is an external symbol (symbol with a GLOBAL binding class).

- Although this directive does not function any differently than an ".extern" directive in that it declares an external label, if this directive is used to declare a label with a definition in the specified file as an external label, use the ".extern" directive to declare labels without definitions in the specified file as external labels.
- The .public directive may be described anywhere in a source program.
- The ".public" directive can only define one symbol per line.
- Symbol(s) to be described in the operand field must be defined within the same module. If it is not defined, an error will be output, and the symbol's ".public" declaration will be ignored.
  The symbol name for which the error occurs will be included in the error message.
- The following symbols cannot be used as the operand of the .public directive:

(1) Symbol defined with the .set directive

(2) Section name

[Example]

- Module 1

```
        .public A1              ; (1)
        .extern B1

A1      .set    0x10

        .cseg   text
        jr      B1
```

- Module 2

```
        .public B1              ; (2)
        .extern A1
        .cseg   text
B1:
        mov     A1, r12
```

(1) This .public directive declares that symbol "A1" is to be referenced from other modules.

(2) This .public directive declares that symbol "B1" is to be referenced from another module.

.extern

Declares to the linker that a symbol (other than bit symbols) in another module is to be referenced in this module.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.extern	*label-name*[, *size*]	[; *comment*]

[Function]

- The .extern directive declares to the linker that a symbol in another module is to be referenced in this module.

[Use]

- When referencing a symbol defined in another module, the .extern directive must be used to declare the symbol as an external reference.

[Description]

- A label with the same name as the one specified by the first operand is declared as an external label[Note].
  Note that if a second operand was specified, this specifies the size of the data indicated by that label.

  Note This is an external symbol (symbol with a GLOBAL binding class).

- Although this directive does not function any differently than an ".public" directive in that it declares an external label, if this directive is used to declare a label without a definition in the specified file as an external label, use the ".public" directive to declare labels with definitions in the specified file as external labels.
- The .extern directive may be described anywhere in a source program.
- The ".extern" directive can only define one symbol per line.
- No error is output even if a symbol declared with the .extern directive is not referenced in the module.
- A symbol that has been declared cannot be described as the operand of the .extern directive. Conversely, a symbol that has been declared as .extern cannot be redefined or declared with any other directive.

.comm

Declares an undefined external symbol.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.comm	*symbol-name, size, alignment-condition*	[; *comment*]

[Function]

- Declares an undefined external symbol having a symbol name specified by the first operand, a size specified by the second operand, and an alignment condition specified by the third operand.

[Description]

(1) If the -Xsdata option is specified upon starting the CX

(a) If the specified by the second operand size is 1 or more, but no more than *num* bytes
Generates a symbol having value of section header table index GPCOMMON upon generating the symbol table entry for the label when the object module file is generated.

(b) If the specified by the second operand size is 0 or more than *num* bytes
Generates a symbol having value of section header table index COMMON upon generating the symbol table entry for the label when the object module file is generated.

(2) If the -Xsdata option is not specified upon starting the CX

(a) Generates a symbol having value of section header table index GPCOMMON upon generating the symbol table entry for the label when the object module file is generated.

- If the same label name as that specified by the first operand is defined by means of normal label definition in the same file as this directive.

  - If the label is declared as having symbol table entry index GPCOMMON and is defined by means of normal label definition in the data-attribute section, or if it is declared as having symbol table entry index COMMON by this directive and is defined by means of normal label definition in the sdata-attribute section.

```
        .comm   lab1, 4, 4  --GPCOMMON if assembly is executed without -G
          :
        .data   .dseg   data
lab1:                       --Normal label definition in .data section
```

The assembler outputs the following message then stops assembling.

```
E0550213: label identifier redefined
```

- Else

The label defined by means of normal label definition is regarded as being an external label and the specification of this directive is ignored. Generates a symbol table entry having binding class GLOBAL upon generating the symbol table entry for the label when the object module file is generated.

```
        .comm   lab1, 4, 4  --GPCOMMON if assembly is executed without -G
          :
        .sdata  .dseg   sdata
lab1:                       --Normal label definition in .sdata section
```

- If a label having the same name as that specified by the first operand is defined by the .lcomm directive in the same file as this directive.

- If the size or alignment condition specified by the .lcomm directive differs from the size or alignment condition specified by this directive.

```
.comm   lab1, 4, 4
  :
.sbss   .dseg   sbss
.lcomm  lab1, 4, 2          --Alignment condition differs
```

The assembler outputs the following message then stops assembling.

```
E0550213: label identifier redefined
```

- If the label is declared, by this directive, as having section header table index GPCOMMON and is defined in the bss-attribute section by the .lcomm directive, or if it is declared by this directive as having section header table index COMMON and is defined in the sbss-attribute section by the .lcomm directive.

```
.comm   lab1, 4, 4          --GPCOMMON if assembly is executed without -G
  :
.bss    .dseg   bss
.lcomm  lab1, 4, 4          --Definition in .bss section
```

The assembler outputs the following message then stops assembling.

```
E0550213: label identifier redefined
```

- Else

The assembler regards the label defined by .lcomm as being an external label, ignoring the specification made by this directive. Generates a symbol table entry having binding class GLOBAL upon generating the symbol table entry for the label when the object module file is generated.

```
.comm   lab1, 4, 4          --GPCOMMON if assembly is executed without -G
  :
.sbss   .dseg   sbss
.lcomm  lab1, 4, 4          --Definition in .bss section
```

- If a label having the same name as that specified by the first operand is (re-)defined by this directive in the same file as this directive.
  - If the size or boundary condition is differen.

```
.comm   lab1, 4, 4
  :
.comm   lab1, 2, 4          --Size differs
```

The assembler outputs the following message then stops assembling.

```
E0550213: label identifier redefined
```

  - When the size and boundary conditions are the same.
    The assembler assumes the .comm directive to be specified once only.

[Example]

Declares undefined external label of size 4 with alignment condition 4.

```
.sbss   .dseg   sbss
.comm   _p, 4, 4
```

4.2.6 Macro directives

When describing a source it is inefficient to have to describe for each series of high usage frequency instruction groups. This is also the source of increased errors.

Via macro directives, using macro functions it becomes unnecessary to describe many times to the same kind of instruction group series, and coding efficiency can be improved.

Macro basic functions are in substitution of a series of statements.

The following macro directives are available.

Table 4-12. Macro Directives

Directive	Overview
.macro	Executes a macro definition by assigning the macro name specified in the symbol field to a series of statements described between .macro directive and the .endm directive.
.local	The specified string is declared as a local symbol that will be replaced as a specific identifier.
.rept	Tells the assembler to repeatedly expand a series of statements described between .rept directive and the .endm directive the number of times equivalent to the value of the expression specified in the operand field.
.irp	Tells the assembler to repeatedly expand a series of statements described between .irp directive and the .endm directive the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters (from the left, the order) specified in the operand field.
.exitm	This directive skips the repetitive assembly of the .irp and .rept directives enclosing this directive at the innermost position.
.exitma	This directive skips the repetitive assembly of the irp and .rept directives enclosing this directive at the outermost position.
.endm	Instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

.macro

Executes a macro definition by assigning the macro name specified in the symbol field to a series of statements described between .macro directive and the .endm directive.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
macro-name	.macro	[*formal-parameter*[, ...]]	[; *comment*]
	:		
	Macro body		
	:		
	.endm		[; *comment*]

[Function]

- The .macro directive executes a macro definition by assigning the macro name specified in the symbol field to a series of statements (called a macro body) described between this directive and the .endm directive.

[Use]

- Define a series of frequently used statements in the source program with a macro name. After its definition only describe the defined macro name, and the macro body corresponding to the macro name is expanded.

[Description]

- If the .endm directive corresponding to .macro directive does not exist, the CX outputs the message.
- For the macro name to be described in the symbol field, see the conventions of symbol description in "(2) Symbol".
- To reference a macro, describe the defined macro name in the mnemonic field.
- For the formal parameter(s) to be described in the operand field, the same rules as the conventions of symbol description will apply.
- Formal parameters are valid only within the macro body.
- An error occurs if any reserved word is described as a formal parameter. However, if a user-defined symbol is described, its recognition as a formal parameter will take precedence.
- The number of formal parameters must be the same as the number of actual parameters. If a shortage of actual parameters, the CX outputs the error message.
- A name or label defined within the macro body if declared with the .local directive becomes effective with respect to one-time macro expansion.
- The number of macros that can be defined within a single source module is not specifically limited. In other words, macros may be defined as long as there is memory space available.
- Formal parameter definition lines, reference lines, and symbol names are not output to a cross-reference list.
- Two or more sections must not be defined in a macro body. If defined, an error will be output.
- An error will be output if there are extra formal parameters that are not referenced in the macro body.
- If an undefined macro is called in a macro body, the CX outputs the message then stops assembling.
- If a currently defined macro is called in a macro body, the CX outputs the message then stops assembling.
- If a parameter defined by a label or directive is specified for a formal parameter, the CX outputs the message and stops assembling.

- The only actual parameters that can be specified in the macro call are label names, symbol names, numbers, registers, and instruction mnemonics.

  If a label expression (LABEL-1), addressing-method specification label (#LABEL), or base register specification ([gp]) or the like is specified, then a message will be output depending on the actual parameter specified, and assembly will halt.

- A line of a sentence can be designated in the macro-body. Such as operand can't designate the part of the sentence. If operand has a macro call, performs a label reference is undefined macro name, or the CX outputs the message then stops assembling.
- An error will be output if a macro is defined in the macro body of a macro definition, but processing will continue (the content up to the corresponding ".endm" directive is ignored). Referencing a macro name will cause a definition error.

[Example]

```
ADMAC   .macro  PARA1, PARA2    ; (1)
        mov     PARA1, r12
        add     PARA2, r12
        .endm                   ; (2)

        ADMAC   0x10, 0x20      ; (3)
```

(1) A macro is defined by specifying macro name "ADMAC" and two formal parameters "PARA1" and "PARA2".

(2) This directive indicates the end of the macro definition.

(3) Macro "ADMAC" is referenced.

.local

The specified string is declared as a local symbol that will be replaced as a specific identifier.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
	.local	*symbol-name*[, ...]	[; *comment*]

[Function]

- The specified string is declared as a local symbol that will be replaced as a specific identifier.

[Use]

- If a macro that defines a symbol within the macro body is referenced more than once, the assembler will output a double definition error for the symbol.
  By using the .local directive, you can reference (or call) a macro, which defines symbol(s) within the macro body, more than once.

[Description]

- Specifying 4,294,967,294 or more local symbols as formal parameters to ".local" quasi directives will cause the following error message to be output, and the assembly will halt.

F0550514: Paramater table overflow.

- Local symbol names generated by the assembler are generated in the range of .??00000000 to .??FFFFFFFF.

[Example]

```
m1      .macro  x
        .local  a, b
        a:      .dw     a
        b:      .dw     x
.endm
m1      10
m1      20
```

The expansion is as follows.

```
.??00000000:    .dw     .??00000000
.??00000001:    .dw     10
.??00000002:    .dw     .??00000002
.??00000003:    .dw     20
```

.rept

Tells the assembler to repeatedly expand a series of statements described between this directive and the .endm directive the number of times equivalent to the value of the expression specified in the operand field.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.rept	*absolute-expression*	[; *comment*]
	:		
	.endm		[; *comment*]

[Function]

- The .rept directive tells the assembler to repeatedly expand a series of statements described between this directive and the .endm directive (called the REPT-ENDM block) the number of times equivalent to the value of the expression specified in the operand field.

[Use]

- Use the .rept and .endm directives to describe a series of statements repeatedly in a source program.

[Description]

- An error occurs if the .rept directive is not paired with the .endm directive.
- If the .exitm directive appears in the REPT-ENDM block, subsequent expansion of the REPT-ENDM block by the assembler is terminated.
- Assembly control instructions may be described in the REPT-ENDM block.
- Macro definitions cannot be described in the REPT-ENDM block.
- The value is evaluated as a 32-bit signed integer.
- If there is no arrangement of statements (block), nothing is executed.
- If the result of evaluating the expression is negative, the CX outputs the message then stops assembling.
- An error will be output if a macro is defined in the macro body of a macro definition, and processing will continue, without performing expansion.

[Example]

```
.cseg   text
    ; REPT-ENDM block
.rept   3               ; (1)
        nop
        ; Source text
.endm                   ; (2)
```

(1) This .rept directive tells the assembler to expand the REPT-ENDM block three consecutive times.

(2) This directive indicates the end of the REPT-ENDM block.

.irp

Tells the assembler to repeatedly expand a series of statements described between .irp directive and the .endm directive the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters (from the left, the order) specified in the operand field.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.irp	*formal-parameter*[*actual-parameter*[, ...]]	[; *comment*]
	:		
	.endm		[; *comment*]

[Function]

- The .irp directive tells the assembler to repeatedly expand a series of statements described between this directive and the .endm directive (called the IRP-ENDM block) the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters (from the left, the order) specified in the operand field.

[Use]

- Use the .irp and .endm directives to describe a series of statements, only some of which become variables, repeatedly in a source program.

[Description]

- If the .endm directive corresponding to .irp directive does not exist, the CX outputs the message.
- If the .exitm directive appears in the IRP-ENDM block, subsequent expansion of the IRP-ENDM block by the assembler is terminated.
- Macro definitions cannot be described in the IRP-ENDM block.
- Assembly control instructions may be described in the IRP-ENDM block.
- If the same parameter name is specified for a formal parameter and an actual parameter, the CX outputs the message and stops assembling.
- If a parameter defined by a label or other directive is specified for a formal parameter and an actual parameter, the CX outputs the message and stops assembling.
- An error will be output if a macro is defined in the macro body of a macro definition, and processing will continue, without performing expansion.

[Example]

```
.cseg   text

.irp    PARA 0xA, 0xB, 0xC              ; (1)
    ; IRP-ENDM block
        add     PARA, r12
        mov     r11, r12
.endm                                   ; (2)
    ; Source text
```

(1) The formal parameter is "PARA" and the actual parameters are the following three: "0xA", "0xB", and "0xC".
This .irp directive tells the assembler to expand the IRP-ENDM block three times (i.e., the number of actual parameters) while replacing the formal parameter "PARA" with the actual parameters "0xA", "0xB", and "0xC"

(2) This directive indicates the end of the IRP-ENDM block.

.exitm

This directive skips the repetitive assembly of the .irp and .rept directives enclosing this directive at the innermost position.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.exitm		[; *comment*]

[Function]

- This directive skips the repetitive assembly of the .irp and .rept directives enclosing this directive at the innermost position.

[Description]

- If this directive is not enclosed by .irp and .rept directives, the CX outputs the message then stops assembling.

.exitma

This directive skips the repetitive assembly of the irp and .rept directives enclosing this directive at the outermost position.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	.exitma		[; *comment*]

[Function]

- This directive skips the repetitive assembly of the irp and .rept directives enclosing this directive at the outermost position.

[Description]

- If this directive is not enclosed by .irp and .rept directives, the CX outputs the message then stops assembling.

.endm

Instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

[Syntax]

Symbol field	Mnemonic field	Operand field	Comment field
	.endm		[; *comment*]

[Function]

- The .endm directive instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

[Use]

- The .endm directive must always be described at the end of a series of statements following the .macro, .rept, and/or the .irp directives.

[Description]

- A series of statements described between the .macro directive and .endm directive becomes a macro body.
- A series of statements described between the .rept directive and .endm directive becomes a REPT-ENDM block.
- A series of statements described between the .irp directive and .endm directive becomes an IRP-ENDM block.
- If the .macro, .rept, or .irp directive corresponding to this directive does not exist, the CX outputs the message then stops assembling.

[Example]

(1) MACRO-ENDM

```
ADMAC   .macro  PARA1, PARA2
                mov     A, #PARA1
                add     A, #PARA2
        .endm
```

(2) REPT-ENDM

```
.cseg   text
  :
.rept   3
        inc     B
        DEC     C
.endm
```

(3) IRP-ENDM

```
.cseg   text
  :
.irp    PARA, <1, 2, 3>
        add     A, #PARA
        mov     [DE], A
.endm
```

4.3 Control Instructions

This chapter describes control instructions.

Control Instructions provide detailed instructions for assembler operation.

4.3.1 Outline

Control instructions provide detailed instructions for assembler operation and so are written in the source.

Control instructions do not become the target of object code generation.

Control instruction categories are displayed below.

Table 4-13. Control Instruction List

Control Instruction Type	Control Instruction
Compile target type specification control instruction	PROCESSOR
Symbol control instructions	EXT_ENT_SIZE, EXT_FUNC
Assembler control instructions	CALLT, REG_MODE, EP_LABEL, NO_EP_LABEL, NO_MACRO, MACRO, DATA, SDATA, NOWARNING, WARNING
File input control instructions	INCLUDE, BINCLUDE
Smart correction control instruction	SMART_CORRECT
Conditional assembly control instructions	IFDEF, IFNDEF, IF, IFN, ELSEIF, ELSEIFN, ELSE, ENDIF

As with directives, control instructions are specified in the source.

Also, among the control instructions displayed in "Table 4-13. Control Instruction List", the following can be written as an cx option even in the command line when the CX is activated.

Table 4-14. Control Instructions and Assembler Option

Control Instruction	Option
PROCESSOR	-C

4.3.2 Compile target type specification control instruction

Compile target type specification control instructions specify the Compile target type in the source module file.
The following compile target type specification control instructions are available.

Table 4-15. Compile Target Type Specification Control Instructions

Control Instruction	Overview
PROCESSOR	Specifies in a source module file the compile target type

PROCESSOR

Specifies in a source module file the compile target type.

[Syntax]

```
[Δ]$[Δ]PROCESSOR[Δ]([Δ]processor-type[Δ])[Δ][;comment]
```

[Function]

- The PROCESSOR control instruction specifies in a source module file the processor type of the target device subject to compile.

[Use]

- The processor type of the target device subject to compile must always be specified in the source module file or in the startup command line of the compiler.
- If you omit the processor type specification for the target device subject to compile in each source module file, you must specify the processor type at each compile operation. Therefore, by specifying the target device subject to compile in each source module file, you can save time and trouble when starting up the compiler.

[Description]

- For the specifiable processor name, see the user's manual of the device used or "Device Files Operating Precautions".
- If the specified processor type differs between the source module file and the option, the compiler will output a warning message and give precedence to the processor type specification in the option.

[Example]

```
$ PROCESSOR (f3507)
```

4.3.3 Symbol control instructions

Using the symbol control instruction, can generate a symbol table entry, define symbols, and specify the size of the data indicated by a label.

The following symbol control instructions are available.

Table 4-16. Symbol Control Instructions

Control Instruction	Overview
EXT_ENT_SIZE	Specifies a flash table entry sizes
EXT_FUNC	Generates a flash table entry

EXT_ENT_SIZE

Specifies a flash table entry size.

[Syntax]

```
[Δ]$[Δ]EXT_ENT_SIZE[Δ]size[Δ][;comment]
```

[Function]

- Sets the value specified by the operand as the flash table entry size.

[Use]

- Sets the value specified by the operand as the flash table entry size when an object module file is generated. Specify this instruction to use the function for relinking a flash area or external ROM.

[Description]

- To specify a branch from an area that cannot be rewritten or replaced (boot area) to a rewritable or replaceable area (flash area), a branch table is generated at a specified address in the flash area by specifying this control instruction and two-stage branch is performed via the table.
- The entry size of this table is 4 bytes by default. A jr instruction is generated and execution can branch in a range of 22 bits from the branch instruction.
- If it is necessary to branch to an address exceeding the range of 22 bits from the branch instruction in this table, execution can branch over the entire 32-bit address space when 8 is specified in the case of the V850Ex core.
- This control instruction must be described in a source file which contains a relevant branch instruction (in the boot area) and a source file which contains a relevant label definition (in the flash area).
- The size specified by this control instruction is the only value for the entire area, including the boot area and flash area.
  If a different size is specified, the CX outputs the message and stops assembling.
- Specify 4 (default) or 8 as the size.

EXT_FUNC

Generates a flash table entry.

[Syntax]

```
[Δ]$[Δ]EXT_FUNC[Δ]label-name,ID-value[Δ][;comment]
```

[Function]

- Generates a flash table entry having a label name and ID value specified by the operands.

[Use]

- Generates a flash table entry having a label name and ID value specified by the operands when an object module file is generated. Specify this instruction to use the function for relinking a flash area or external ROM.

[Description]

- To specify a branch from an area that cannot be rewritten or replaced (boot area) to a rewritable or replaceable area (flash area), a branch table is generated to a specified address in a flash area by specifying this control instruction and two-stage branch is performed via the table.
- This control instruction must be written in a source file which contains a relevant branch instruction (in the boot area) and a source file which contains a relevant label definition (in the flash area).
- If the same label name is specified with a different ID value, the CX outputs the message then stops assembling.
- If the same ID value is specified with a different label name, the CX outputs the message then stops assembling.
- It is recommended that all relevant label names be written in a single file and included into source files of the boot area and flash area using the INCLUDE control instruction. This prevents contradictions described above.
- The ID value must be a positive number. The size of a branch table to be allocated depends on the maximum ID value. Renesas Electronics recommends that the ID value be specified without spaces.

4.3.4 Assembler control instructions

The assembler control instruction can be used to control the processing performed by the assembler.
The following assembler control instructions are available.

Table 4-17. Assembler Control Instructions

Control Instruction	Overview
CALLT	A control instruction which is reserved for the compiler
REG_MODE	Outputs a register mode information section
EP_LABEL	Performs a label reference by %label as a reference by ep offset
NO_EP_LABEL	Cancels the specification made with the EP_LABEL directive
NO_MACRO	Does not expand the subsequent instructions
MACRO	Cancels the specification made with the NO_MACRO directive
DATA	Assumes that external data having symbol name extern_symbol has been allocated to the data or bss attribute section, and expands the instructions which reference that data
SDATA	Assumes that external data having symbol name extern_symbol has been allocated to the sdata or sbss attribute section, and dose not expand the instructions which reference that data
NOWARNING	Does not output warning messages
WARNING	Output warning messages

CALLT

A control instruction which is reserved for the compiler.

[Syntax]

```
[Δ]$[Δ]CALLT[Δ][;comment]
```

[Function]

- A control instruction which is reserved for the compiler.

[Description]

- Do not delete a callt instruction when it exists in the assembler source file output by the compiler. If it is deleted, the prologue epilogue runtime linking cannot be checked.

REG_MODE

A register mode information section is output.

[Syntax]

```
[Δ]$[Δ]REG_MODE[Δ]specify-register-mode[Δ][;comment]
```

[Function]

- A register mode information section is output into the object module file generated by the assembler.

[Description]

- Specify the register mode as "22" (indicating register mode 22); "26" (indicating register mode 26); "32" (indicating register mode 32); or "common" (indicating universal register mode).
- A register mode information section stores information about the number of working registers and register-variable registers used by the compiler. It is set in the object module file via this control instruction.
- If register mode 22 is used, then there are 5 working registers and 5 register-variable registers; if register mode 26 is used, then there are 7 of each; and if register mode 32 is used, then there are 10 of each.
- If register mode 32 is used, a register mode information section is not output into the object module file generated by the assembler.

EP_LABEL

Performs a label reference by %label as a reference by ep offset.

[Syntax]

```
[Δ]$[Δ]EP_LABEL[Δ][;comment]
```

[Function]

- Performs a label reference by %label as a reference by ep offset for the subsequent instructions.
- If $EP_LABEL is omitted, then the assembler will assume that $EP_LABEL was specified.

NO_EP_LABEL

Cancels the specification made with the EP_LABEL directive.

[Syntax]

```
[Δ]$[Δ]NO_EP_LABEL[Δ][;comment]
```

[Function]

- Cancels the specification made with the EP_LABEL directive for the subsequent instructions.
- If $NO_EP_LABEL is omitted, then the assembler will assume that $EP_LABEL was specified.

NO_MACRO

Does not expand the subsequent instructions.

[Syntax]

```
[Δ]$[Δ]NO_MACRO[Δ][;comment]
```

[Function]

- Does not expand the subsequent instructions, other than the setfcond/jcond/jmp/cmovcond/sasfcond instructions.

MACRO

Cancels the specification made with the NO_MACRO directive.

[Syntax]

```
[Δ]$[Δ]MACRO[Δ][;comment]
```

[Function]

- Cancels the specification made with the NO_MACRO directive for the subsequent instructions.

DATA

Assumes that external data having symbol name extern_symbol has been allocated to the data or bss attribute section, and expands the instructions which reference that data.

[Syntax]

```
[Δ]$[Δ]DATA[Δ]symbol-name[Δ][;comment]
```

[Function]

- Assumes that external data having symbol name extern_symbol has been allocated to the data or bss attribute section, regardless of the size specified with the -Xsdata option, and expands the instructions which reference that data.
- This format is used when a variable for which "data" is specified in #pragma section or section file is externally referenced by an assembler source file.

SDATA

Assumes that external data having symbol name extern_symbol has been allocated to the sdata or sbss attribute section, and dose not expand the instructions which reference that data.

[Syntax]

```
[Δ]$[Δ]SDATA[Δ]symbol-name[Δ][;comment]
```

[Function]

- Assumes that external data having symbol name extern_symbol has been allocated to the sdata or sbss attribute section, regardless of the size specified with the -Xsdata option, and does not expand the instructions which reference that data.
- This format is used when a variable for which "sdata" is specified in #pragma section or section file is externally referenced by an assembler source file.

NOWARNING

Does not output warning messages.

[Syntax]

```
[Δ]$[Δ]NOWARNING[Δ][;comment]
```

[Function]

- Does not output warning messages for the subsequent instructions.

WARNING

Output warning messages.

[Syntax]

```
[Δ]$[Δ]WARNING[Δ][;comment]
```

[Function]

- Output warning messages for the subsequent instructions.

4.3.5 File input control instructions

Using the file input control instruction, the CX can input an assembler source file or binary file to a specified position.
The following file input control instructions are available.

Table 4-18. File Input Control Instructions

Control Instruction	Overview
INCLUDE	Quotes a series of statements from another source module file
BINCLUDE	Inputs a binary file

INCLUDE

Quote a series of statements from another source module file.

[Syntax]

```
[Δ]$[Δ]INCLUDE[Δ]([Δ]file-name[Δ])[Δ][;comment]
```

[Function]

- The INCLUDE control instruction tells the assembler to insert and expand the contents of a specified file beginning on a specified line in the source program for assembly.

[Use]

- A relatively large group of statements that may be shared by two or more source modules should be combined into a single file as an INCLUDE file.
  If the group of statements must be used in each source module, specify the filename of the required INCLUDE file with the INCLUDE control instruction.
  With this control instruction, you can greatly reduce time and labor in describing source modules.

[Description]

- The INCLUDE control instruction can only be described in ordinary source programs.
- The search pass of an INCLUDE file can be specified with the option (-I).
- The assembler searches INCLUDE file read paths in the following sequence:

(1) Folder specified by the option (-I)

(2) Folder in which the source file exists

(3) Folder containing the (original) C source file

(4) Currently folder

- The INCLUDE file can do nesting (the term "nesting" here refers to the specification of one or more other INCLUDE files in an INCLUDE file).
- The maximum nesting level for include files is 4,294,967,294 (=0xFFFFFFFE) (theoretical value). The actual number that can be used depends on the amount of memory, however.
- If the specified INCLUDE file cannot be opened, the CX outputs the message and stops assembling.
- If an include file contains a block from start to finish, such as a section definition directive, macro definition directive, or conditional assembly control instruction, then it must be closed with the corresponding code. If it is not so closed, then an error will be output, and assembly will continue assuming the include file is closed.
- Section definition directive, macro definition directives, and conditional assembly control instructions that are not targets for assembly are not checked.

BINCLUDE

Inputs a binary file.

[Syntax]

```
[Δ]$[Δ]BINCLUDE[Δ]([Δ]file-name[Δ])[Δ][;comment]
```

[Function]

- Assumes the contents of the binary file specified by the operand to be the result of assembling the source file at the position of this control instruction.

[Description]

- The search pass of an INCLUDE file can be specified with the option (-I).
- The assembler searches INCLUDE file read paths in the following sequence:

(1) Folder specified by the option (-I)

(2) Folder in which the source file exists

(3) Folder containing the (original) C source file

(4) Currently folder

- This control instruction handles the entire contents of the binary files. When a relocatable file is specified, this control instruction handles files configured in ELF format. Note that it is not just the contents of the .text selection, etc. that are handled.
- If a non-existent file is specified, the CX outputs the message then stops assembling.

4.3.6 Smart correction control instruction

You can use the smart correction control instruction to instruct that an uncorrected function be changed to a corrected function in an object module file.

The following smart correction control instructions are available.

Table 4-19. Smart Correction Control Instruction

Control Instruction	Overview
SMART_CORRECT	Changes an uncorrected function to a corrected function

SMART_CORRECT

Instruct that the uncorrected function be changed to the corrected function in an object module file.

[Syntax]

```
[Δ]$[Δ]SMART_CORRECTΔstart-label-uncorrected-function,end-label-uncorrected-function,
start-label-corrected-functionΔ[;comment]
```

[Function]

- Instruct that the uncorrected function be changed to the corrected function in an object module file.

[Description]

- Instruct that the uncorrected function be changed to the corrected function in an object module file.
- The assembler outputs a branch instruction to branch from the start of the uncorrected function to the corrected function.
- The branch instruction to branch to the corrected function (_func) is as follows.

```
jr32    _func
```

- If the code size of the uncorrected function is smaller than the size of the code needed to call the corrected function, then an error message is output, and assembly halts.

4.3.7 Conditional assembly control instructions

Using conditional assembly control instruction, the CX can control the range of assembly according to the result of evaluating a conditional expression.

The following conditional assembly control instructions are available.

Table 4-20. Conditional Assembly Control Instructions

Control Instruction	Overview
IFDEF	Control based on symbol (assembly performed when the symbol is defined)
IFNDEF	Control based on symbol (assembly performed when the symbol is not defined)
IF	Control based on absolute expression (assembly performed when the value is true)
IFN	Control based on absolute expression (assembly performed when the value is false)
ELSEIF	Control based on absolute expression (assembly performed when the value is true)
ELSEIFN	Control based on absolute expression (assembly performed when the value is false)
ELSE	Control based on absolute expression/symbol
ENDIF	End of control range

The maximum number of nest level of the conditional assembly control instruction is 4,294,967,294 (=0xFFFFFFFE) (theoretical value). The actual number that can be used depends on the amount of memory, however.

IFDEF

Control based on symbol (assembly performed when the symbol is defined).

[Syntax]

```
[Δ]$[Δ]IFDEF[Δ]switch-name[Δ][;comment]
```

[Function]

- If the switch name specified by the operand is defined.

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

- If the specified switch name is not defined.
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(2) Symbol").
- Switch names can overlap with user-defined symbols other than reserved words. Note, however, that overlapping between switch names is checked.
- Switch names are not output to the assembly list file's symbol-list information or cross-reference information.

IFNDEF

Control based on symbol (assembly performed when the symbol is not defined).

[Syntax]

```
[Δ]$[Δ]IFNDEF[Δ]switch-name[Δ][;comment]
```

[Function]

- If the switch name specified by the operand is defined.
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

- If the specified switch name is not defined.

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(2) Symbol").
- Switch names can overlap with user-defined symbols other than reserved words. Note, however, that overlapping between switch names is checked.
- Switch names are not output to the assembly list file's symbol-list information or cross-reference information.

IF

Control based on absolute expression (assembly performed when the value is true).

[Syntax]

```
[Δ]$[Δ]IF[Δ]absolute-expression[Δ][;comment]
```

[Function]

- If the absolute expression specified by the operand is evaluated as being true (≠ 0).

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

- If the absolute expression is evaluated as being false (= 0).
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- Absolute expressions are evaluated as 32-bit signed integers.

IFN

Control based on absolute expression (assembly performed when the value is false).

[Syntax]

```
[Δ]$[Δ]IFN[Δ]absolute-expression[Δ][;comment]
```

[Function]

- If the absolute expression specified by the operand is evaluated as being true (≠ 0).
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

- If the absolute expression is evaluated as being false (= 0).

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- Absolute expressions are evaluated as 32-bit signed integers.

ELSEIF

Control based on absolute expression (assembly performed when the value is true).

[Syntax]

```
[Δ]$[Δ]ELSEIF[Δ]absolute-expression[Δ][;comment]
```

[Function]

- If the absolute expression specified by the operand is evaluated as being true (≠ 0).

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

- If the absolute expression is evaluated as being false (= 0).
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- Absolute expressions are evaluated as 32-bit signed integers.

ELSEIFN

Control based on absolute expression (assembly performed when the value is false).

[Syntax]

```
[Δ]$[Δ]ELSEIFN[Δ]absolute-expression[Δ][;comment]
```

[Function]

- If the absolute expression specified by the operand is evaluated as being true (≠ 0).
  Skips to the ELSEIF, ELSEIFN, ENSE, or ENDIF control instruction corresponding to this control instruction.

- If the absolute expression is evaluated as being false (= 0).

  (a) If this control instruction and the corresponding ELSEIF, ELSEIFN, or ELSE control instruction exist, assembles the block enclosed within this control instruction and the corresponding control instruction.

  (b) If none of the corresponding control instruction detailed above exist, assembles the block enclosed within this control instruction and the corresponding ENDIF control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.
- Absolute expressions are evaluated as 32-bit signed integers.

ELSE

Control based on absolute expression/symbol.

[Syntax]

```
[Δ]$[Δ]ELSE[Δ]absolute-expression[Δ][;comment]
```

[Function]

- If the specified switch name is not defined by the IFDEF control instruction, if the absolute expression of the IF, or ELSEIF control instruction is evaluated as being false (= 0), or if the absolute expression of the IFN, or ELSEIFN control instruction is evaluated as being true (≠ 0), assembles the arrangement of statements (block) enclosed within this control instruction and the corresponding ENDIFcontrol instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.

ENDIF

End of control range.

[Syntax]

```
[Δ]$[Δ]ENDIF[Δ]absolute-expression[Δ][;comment]
```

[Function]

Indicates the end of the control range of a conditional assembly control instruction.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- This control instruction can be placed in an ordinary source program.

4.4 Macro

This section lainshe hthe cro function.

This is very convenient function to describe serial instruction group for number of times in the program.

4.4.1 Outline

This macro function is very convenient function to describe serial instruction group for number of times in the program.

Macro function is the function that is deployed at the location where serial instruction group defined as macro body is referred by macros as per .macro, .endm directives.

Macro differs from subroutine as it is used to improve description of the source.

Macro and subroutine has features respectively as follows. Use them effectively according to the respective purposes.

- Subroutine

  Process required many times in program is described as one subroutine. Subroutine is converted in machine language only once by assembler.

  Subroutine/call instruction (generally instruction for argument setting is required before and after it) is described only in subroutine reference. Consequently, memory of program can be used effectively by using subroutine.

  It is possible to draw structure of program by executing subroutine for process collected serially in program (Because program is structured, entire program structure can be easily understood as well setting of the program also becomes easy.).

- Macro

  Basic function of macro is to replace instruction group.

  Serial instruction group defined as macro body by .macro, .endm directives are deployed in that location at the time of referring macro. Assembler deploys macro/body that detects macro reference and converts the instruction group to machine language while replacing temporary parameter of macro/body to actual parameter at the time of reference.

  Macro can describe a parameter.

  For example, when process sequence is the same but data described in operand is different, macro is defined by assigning temporary parameter in that data. When referring the macro, by describing macro name and actual parameter, handling of various instruction groups whose dercription is different in some parts only is possible.

Subroutine technique is used to improve efficiency of coding for macro to use to draw structure of program and reducing memory size.

4.4.2 Usage of macro

A macro is described by registering a pattern with a set sequence and by using this pattern. A macro is defined by the user. A macro is defined as follows. The macro body is enclosed by ".macro" and ".endm".

```
PUSHMAC .macro  REG     --The following two statements constitute the macro body.
        add     -4, sp
        st.w    REG, 0x0[sp]
.endm
```

If the following description is made after the above definition has been made, the macro is replaced by a code that "stores r19 in the stack".

```
PUSHMAC r19
```

In other words, the macro is expanded into the following codes.

```
add     -4, sp
st.w    r19, 0x0[sp]
```

4.4.3 Macro operator

This section describes the combination symbols "~" and "$", which are used to link strings in macros.

(1) ~ (Concatenation)

- The concatenation "~" concatenates one character or one character string to another within a macro body. At macro expansion time, the character or character string on the left of the concatenation is concatenated to the character or character string on the right of the sign. The "~" itself disappears after concatenating the strings.
- The symbols before and after the combination symbol "~" in the symbols of a macro definition can be recognized as formal parameters or local symbols, and combination symbols can also be used as delimiter symbols. At macro expansion time, strings before and after the "~" in the symbol are evaluated as the local symbols and formal parameters, and concatenated into single symbols.
- The character "~" can only be used as a combination symbol in a macro definition.
- The "~" in a character string and comment is simply handled as data.
- Two "~" signs in succession are handled as a single "~" sign.

Examples 1.

```
abc     .macro  x
        abc~x:  mov     r10, r20
                sub def~x, r20
.endm
abc NECEL
```

```
[Development result]
abcNECEL:       mov     r10, r20
                sub     defNECEL, r20
```

2.

```
abc     .macro  x, xy
        a_~xy:  mov     r10, r20
        a_~x~y: mov     r20, r10
.endm
abc necel, NECEL
```

```
[Development result]
a_NECEL:    mov     r10, r20
a_necely:   mov     r20, r10
```

3.

```
abc     .macro  x, xy
        ~ab:    mov r10, r20
.endm
abc necel, NECEL
```

```
[Development result]
ab:     mov     r10, r20
```

(2) $ (Dollar symbol)

If a symbol prefixed with a dollar symbol ($) is specified as an actual argument for a macro call, the assembler assumes the symbol to be specified as an actual argument. If, however, an identifier other than a symbol or an undefined symbol name is specified immediately after the dollar symbol ($), the as850 outputs the message then stops assembling.

Example

```
mac1    .macro  x
        mov     x, r10
.endm
.macro  mac2
        .set    value, 10
        mac1    value
        mac1    $value
.endm
mac2
```

```
[Development result]
.set    value, 10
mov     value, r10
mov     10, r10
```

4.5 Reserved Words

The assembler has reserved words. Reserve word cannot be used in symbol, label, section name, macro name. If a reserved word is specified, the CX outputs the message and stops assembling. Reserve word doesn't distinguish between uppercase and lowercase.

The reserved words are as follows.

- Instructions (such as add, sub, and mov)
- Directives
- Control instructions
- Register names, Internal register name

4.6 Assembler Generated Symbols

The following is a list of symbols generated by the assembler for use in internal processing.

This excludes, however, reserved section names. Symbols with the same names as the symbols below cannot be used.

Table 4-21. Assembler Generated Symbols

Symbol Name	Explanation
.__multi_*N* .__multi_*N*.end (*N* : 0 to 4294967294))	Multi-core information symbols
.??00000000 to .??FFFFFFFF	.local directive generation local symbols
__s_PPPP_SSSS0000 (PPPP : Primary file names) (SSSS : text section name)	Symbols for assembler debugging information Example : __s_src_sub_sample0000

4.7 Instructions

This section describes various instruction functions of V850 microcontroller products.

See the device with an instruction set of V850E2V3 product user's manual and architecture edition for details about the device with an instruction set of V850E2V3.

4.7.1 Memory space

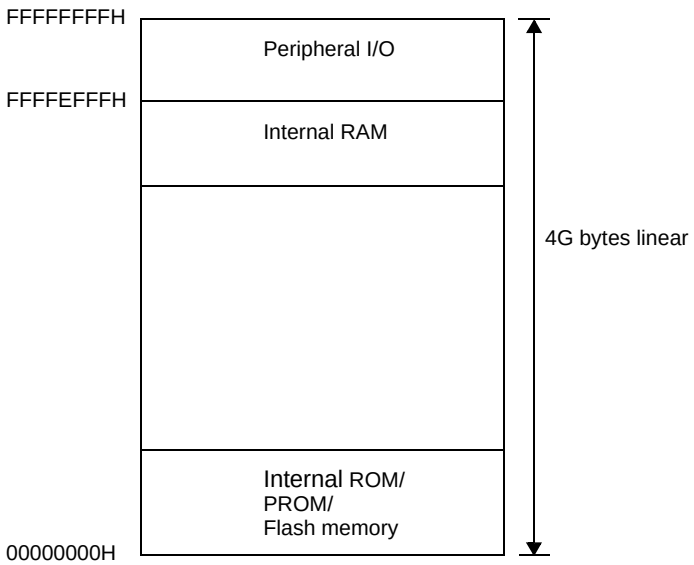
V850 microcontroller has architecture of 32 bit and supports linear address space (data space) of maximum 4G byte in operand addressing.

On other hand, linear address space (program space) of maximum 16M byte is supported in address of instruction address.

Memory map of V850 microcontroller is shown below.

However, see user's manual of each product for details as contents of internal ROM, internal RAM etc are different for each product.

Figure 4-2. Memory Map of V850 Microcontroller



4.7.2 Register

Register can be divided broadly in 2 types of registers such as program register used for general program and system register used for controlling of executing environment. Register has width of 32 bits.

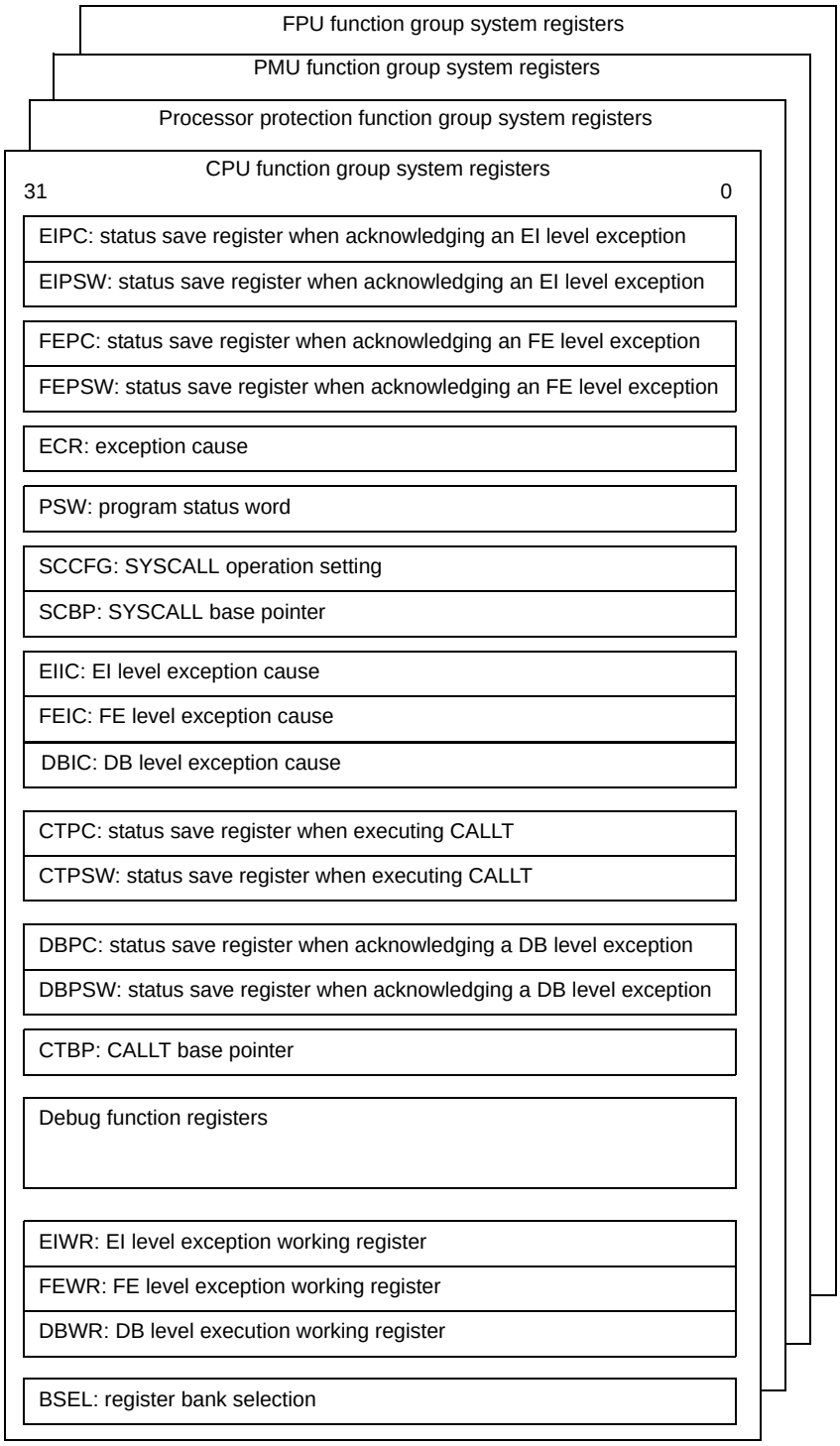
Figure 4-3. Program Register

31 0

r0: Zero register
r1: Assembler reserve register
r2
r3: Stack pointer(SP)
r4: Global pointer(GP)
r5: Text pointer(TP)
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15
r16
r17
r18
r19
r20
r21
r22
r23
r24
r25
r26
r27
r28
r29
r30: Element pointer(EP)
r31: Link pointer(LP)

PC: Program counter

Figure 4-4. System Register



See the device with an instruction set of V850E2V3 product user's manual and architecture edition for details.

(1) Program register

The program registers include general-purpose registers (r0 to r31) and a program counter (PC).

Table 4-22. Program Registers

Name	Purpose	Operation
r0	Zero register	Always holds 0.
r1	Assembler reserved registe	Working register when generating the address.
r2	Address/data variable register (when the real-time OS to be used is not using r2).	
r3	Stack pointer	Used for stack frame generation when function is called.
r4	Global pointer	Used to access global variable in data area.
r5	Text pointer	Used as register for pointing to start address of text area (area where program code is placed).
r6 to r29	Address/data variable registers.	
r30	Element pointer	Used as base pointer when generating address at the time of accessing the memory.
r31	Link pointer	Used when compiler calls function.
PC	Program counter	Saves instruction address in program execution.

(a) General purpose registerr0 to r31

Thirty-two general-purpose registers, r0 to r31, are provided. These registers can be used for address variables or data variables.

However, care must be exercised as follows in using the r0 to r5, r30, and r31 registers.

<1> r0, r30

r0 and r30 are implicitly used by instructions.

r0 is a register that always holds 0, and is used for operations using 0 and offset 0 addressing.

r30 is used as base pointer by SLD instruction or SST instruction when accessing memory.

<2> r1, r3 to r5, r31

r1, r3 to r5, and r31 are implicitly used by the assembler and C compiler.

Before using these registers, therefore, their contents must be saved so that they are not lost. The contents must be restored to the registers after the registers have been used.

<3> r2

r2 is sometimes used by a real-time OS.

When the real-time OS is not using r2, r2 can be used as an address variable register or a data variable register.

(b) Program counter: PC

This register holds an instruction address during program execution.

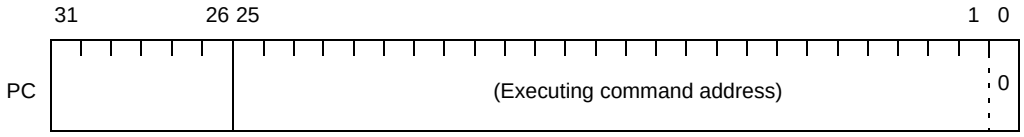
Further, meaning of each bit of PC differs according to the types (V850ES, V850E1, V850E2) of CPU.

<1> V850ES, V850E1

Bits 25-0 are valid and bits 31-26 are reserved for future function expansion (fixed to 0).

If a carry occurs from bit 25 to bit 26, it is ignored. Bit Bit 0 is always fixed to 0 so that execution cannot branch to an odd address.

Figure 4-5. Program Counter [V850ES, V850E1]

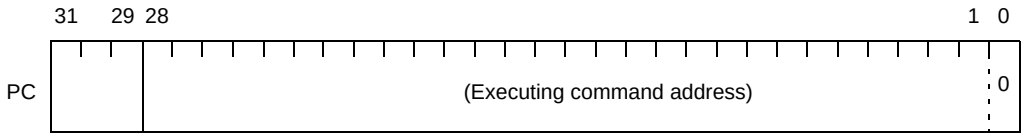


<2> V850E2

Bits 28-0 are valid and bits 31-29 are reserved for future function expansion (fixed to 0).

If a carry occurs from bit 28 to bit 29, it is ignored. Bit 0 is always fixed to 0 so that execution cannot branch to an odd address.

Figure 4-6. Program Counter [V850E2]



4.7.3 Addressing

The CPU generates two types of addresses: instruction addresses used for instruction fetch and branch operations; and operand addresses used for data access.

(1) Instruction address

An instruction address is determined by the contents of the program counter (PC), and is automatically incremented (+2) according to the number of bytes of an instruction to be fetched each time an instruction is executed. When a branch instruction is executed, the branch destination address is loaded into the PC using one of the following two addressing modes.

(a) Relative addressing (PC relative)

The signed 9- or 22-bit data of an instruction code (displacement: disp x) is added to the value of the program counter (PC). At this time, the displacement is treated as 2's complement data with bits 8 and 21 serving as sign bits (S).

JR disp22 instruction, JARL disp22, reg2 instruction, JR disap32 instruction, JARL disp32, reg1 instruction, Bcnd disp9 instruction is the target of this addressing.

Figure 4-7. Relative Addressing (JR disp22/JARL disp22, reg2) [V850ES, V850E1]

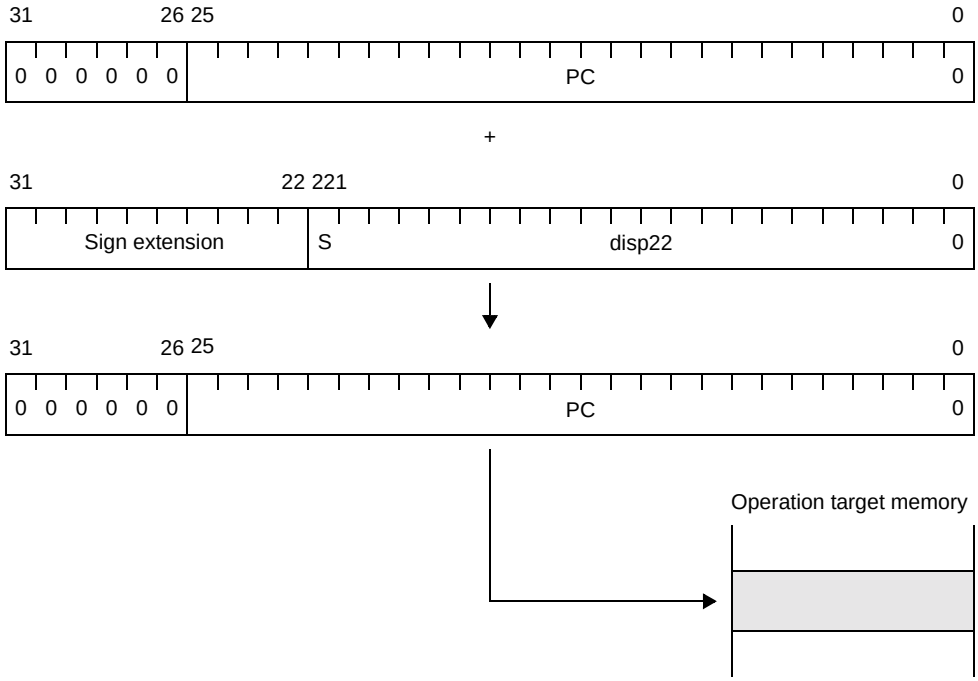


Figure 4-8. Relative Addressing (JR disp22/JARL disp22, reg2) [V850E2]

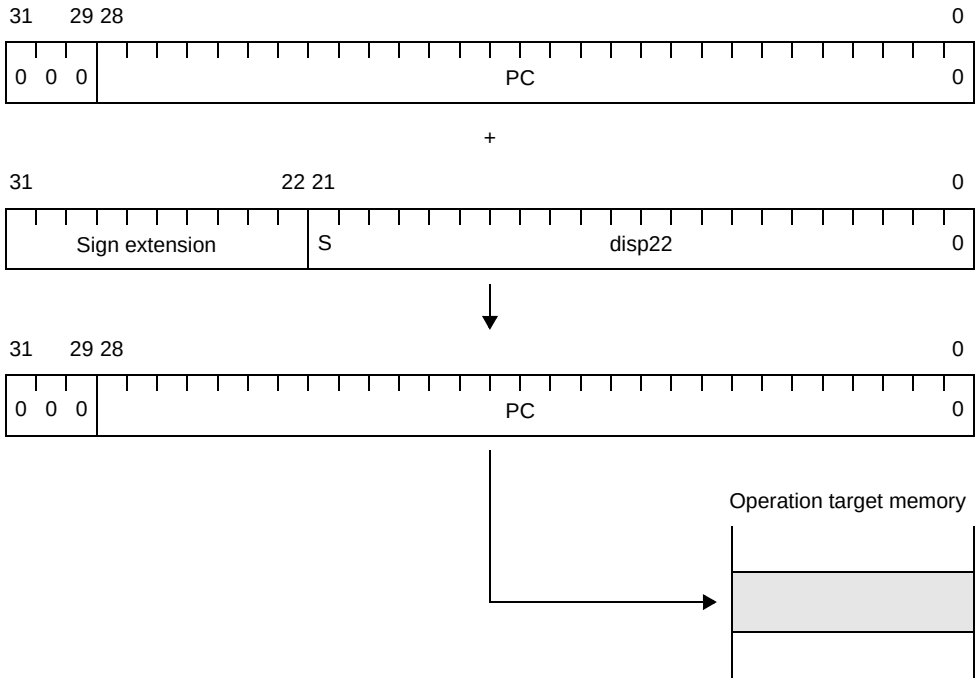


Figure 4-9. Relative Addressing (JR disp32/JARL disp32, reg2) [V850E2]

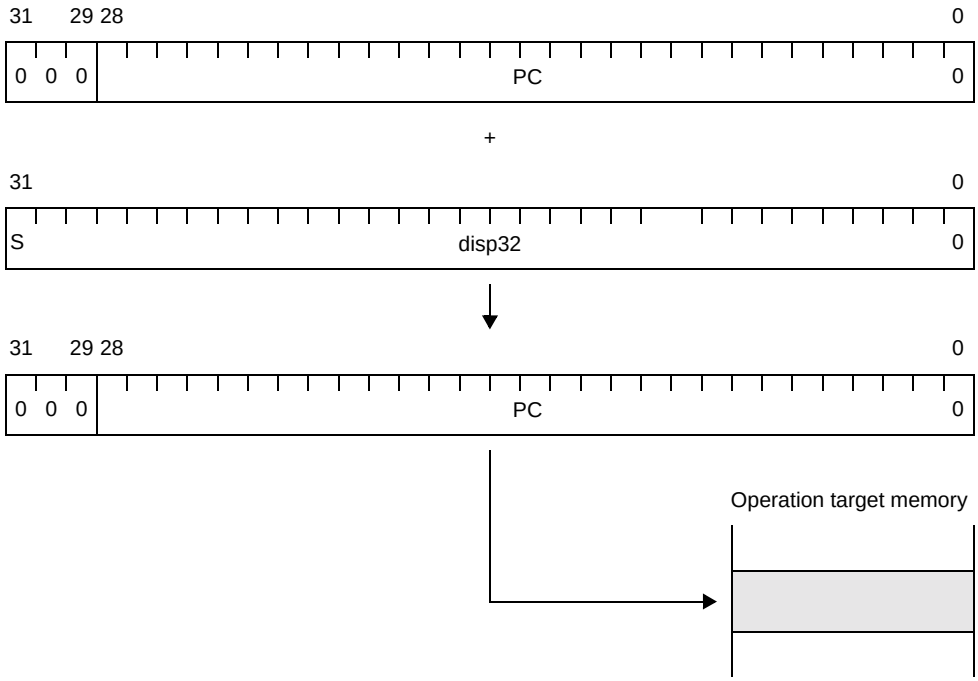


Figure 4-10. Relative Addressing (Bcnd disp9) [V850ES, V850E1]

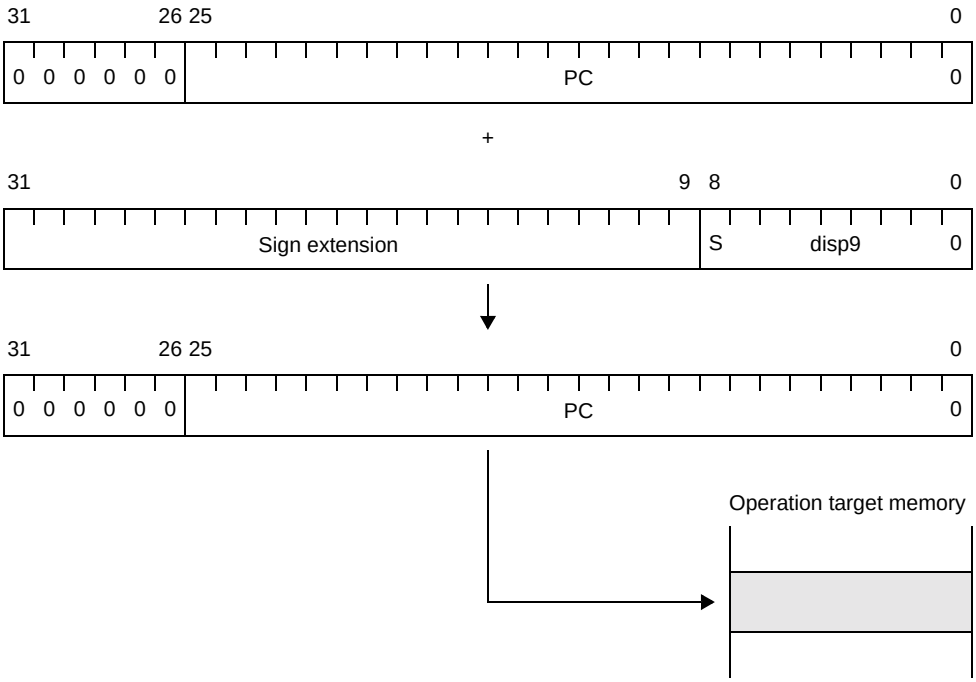
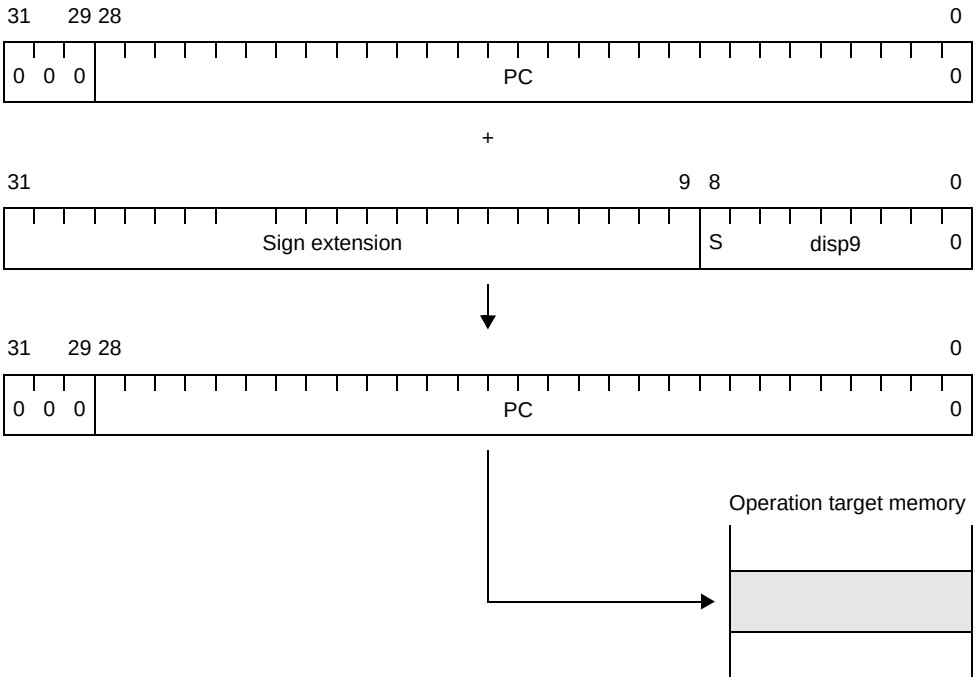


Figure 4-11. Relative Addressing (Bcnd disp9) [V850E2]



(b) Register addressing (Register indirect)

The contents of a general-purpose register (reg1) specified by an instruction are transferred to the program counter (PC).

This addressing is used for the JMP [reg1] instruction.

Figure 4-12. Register Addressing (JMP [reg1]) [V850ES, V850E1]

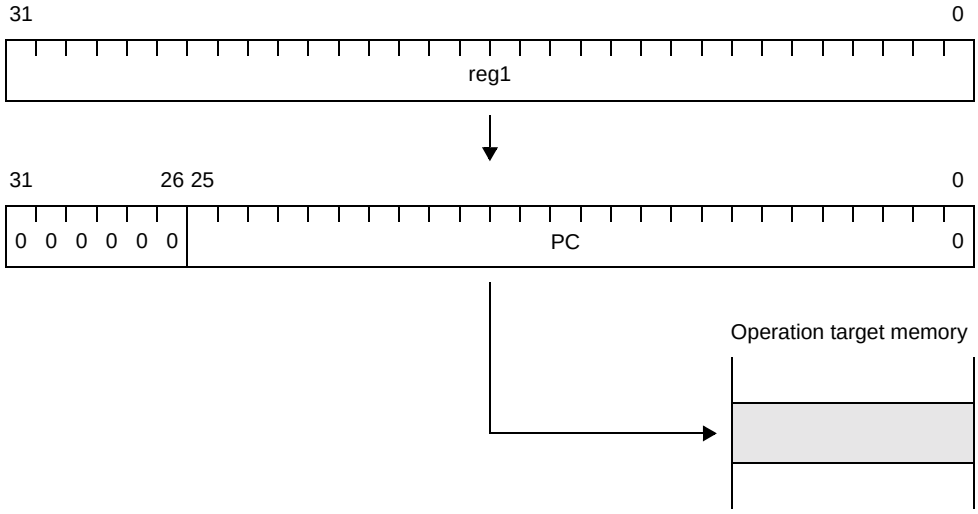
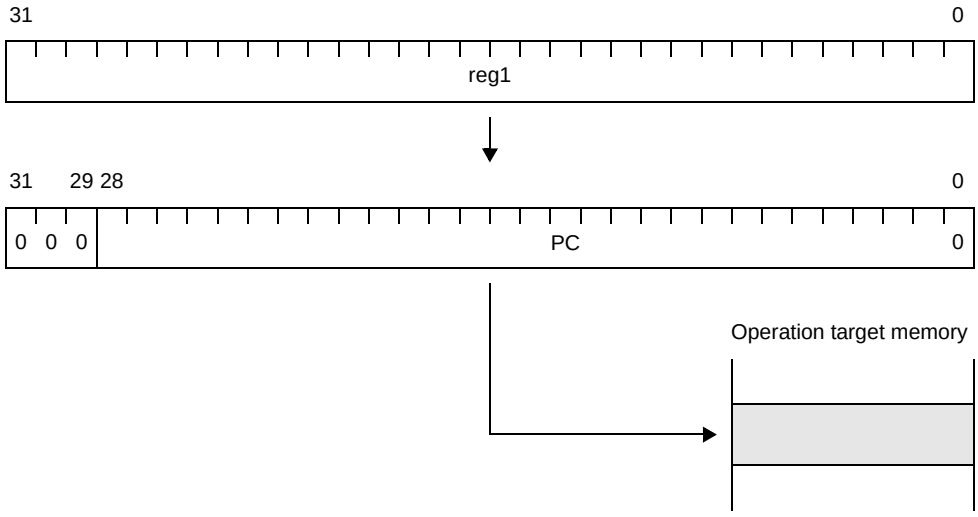


Figure 4-13. Register Addressing (JMP [reg1]) [V850E2]

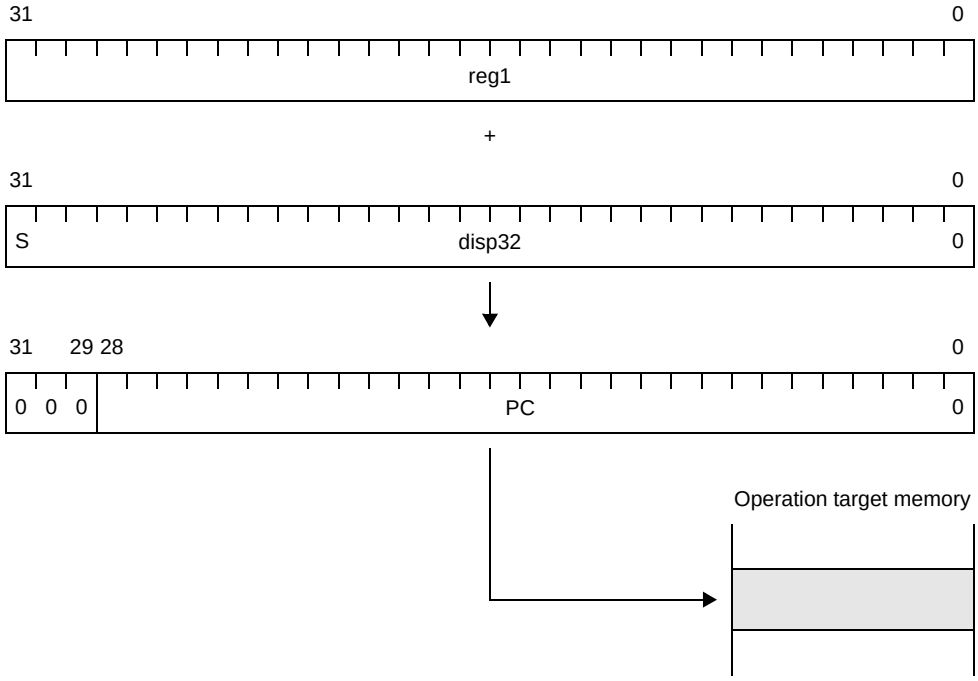


(c) Based addressing

Contents of general purpose register (reg1) specified by command, in which 32 bit data (displacement: disp) is added, are forwarded in program counter (PC).

This addressing is used for the JMP disp32 [reg1] instruction.

Figure 4-14. Register Addressing (JMP disp32[reg1]) [V850E2]



(2) Operand address

When an instruction is executed, the register or memory area to be accessed is specified in one of the following four addressing modes.

(a) Register addressing

The general-purpose register or system register specified in the general-purpose register specification field is accessed as operand.

This addressing mode applies to instructions using the operand format reg1, reg2, reg3, or regID.

(b) Immediate addressing

The 5-bit or 16-bit data for manipulation is contained in the instruction code

This addressing mode applies to instructions using the operand format imm5, imm16, vector, or cccc.

<1> vector

Operand that is 5-bit immediate data for specifying a trap vector (00H to 1FH), and is used in the TRAP instruction.

<2> cccc

Operand consisting of 4-bit data used in the CMOV, SASF, and SETF instructions to specify a condition code. Assigned as part of the instruction code as 5-bit immediate data by appending 1-bit 0 above the highest bit.

(c) Based addressing

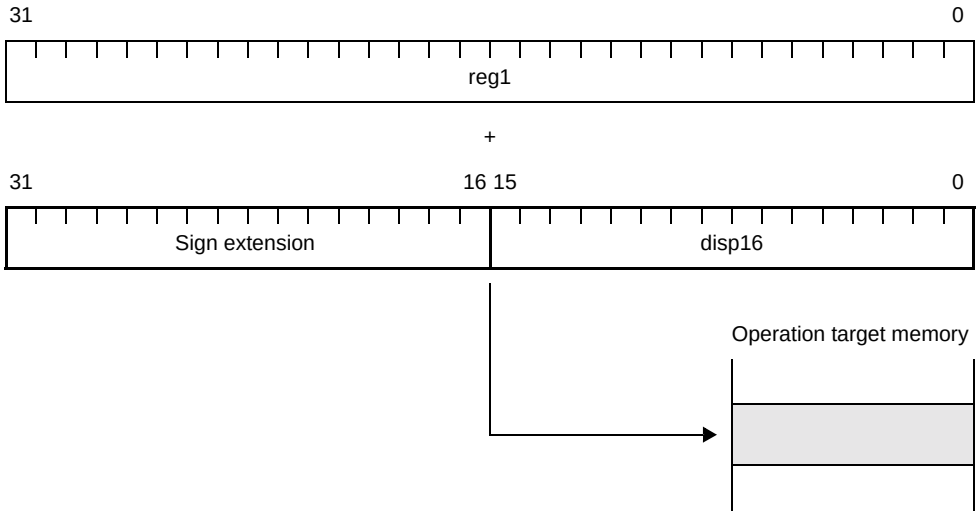
The following two types of based addressing are supported.

<1> Type 1

The address of the data memory location to be accessed is determined by adding the value in the specified general-purpose register (reg1) to the 16-bit displacement value (disp16) contained in the instruction code.

This addressing mode applies to instructions using the operand format disp16 [reg1].

Figure 4-15. Based Addressing (Type1) [V850ES, V850E1, V850E2]

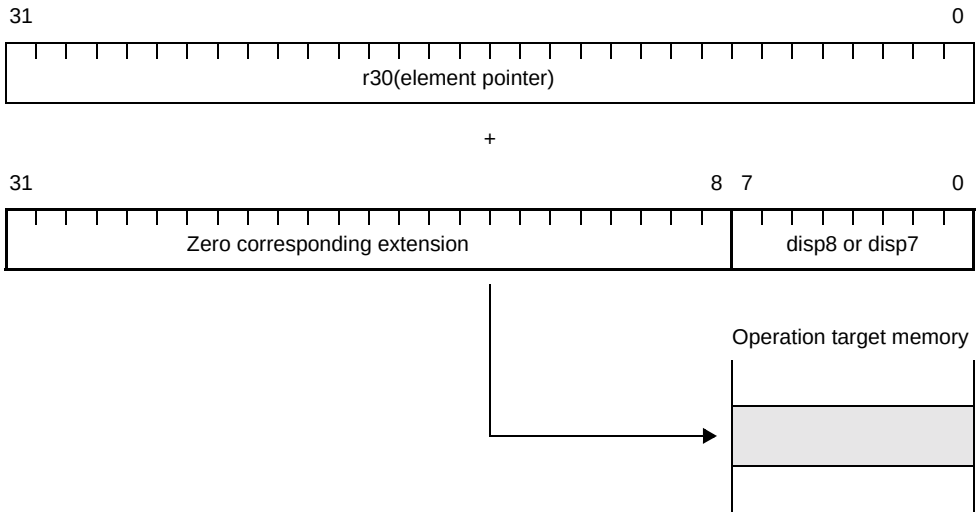


<2> Type 2

The address of the data memory location to be accessed is determined by adding the value in the element pointer (r30) to the 7- or 8-bit displacement value (disp7, disp8).

This addressing mode applies to SLD and SST instructions.

Figure 4-16. Based Addressing (Type2) [V850ES, V850E1, V850E2]



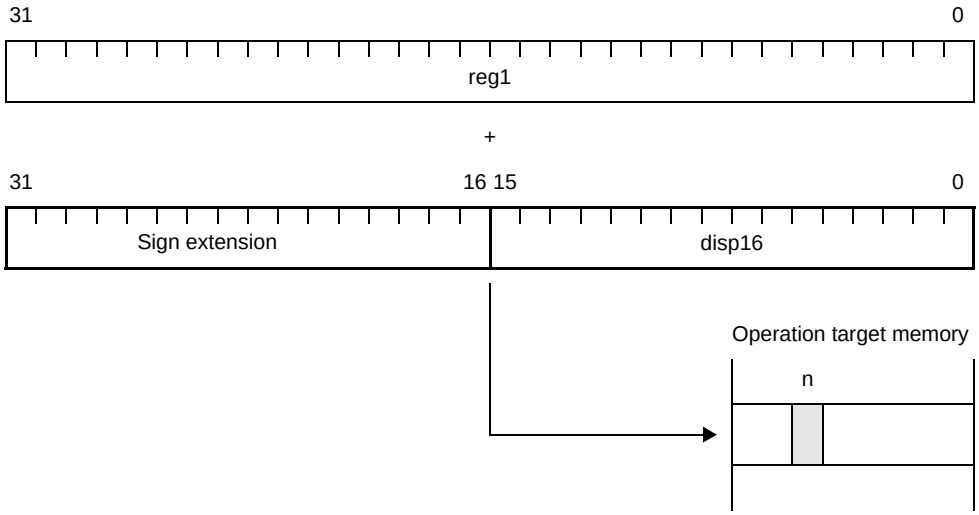
Remark Byte access = disp7
Halfword access and word access: disp8

(d) Bit addressing

This addressing is used to access 1 bit (specified with bit#3 of 3-bit data) among 1 byte of the memory space to be manipulated by using an operand address which is the sum of the contents of a general-purpose register. (reg1) and a 16-bit displacement (disp16) sign-extended to a word length.

This addressing mode applies only to bit manipulation instructions.

Figure 4-17. Bit Addressing [V850ES, V850E1, V850E2]



Remark n: Bit position specified with 3-bit data (bit#3) (n = 0 to 7)

4.7.4 Instruction set

This section explains the instruction set supported by the CX.

(1) Description of symbols

Next table lists the meanings of the symbols used further.

Table 4-23. Meaning of Symbols

Symbols	Meaning
CMD	Instruction
CMDi	Instruction(andi, ori, or xori)
reg, reg1, reg2	Register
r0, R0	Zero register
R1	Assembler-reserved register
gp	Global pointer (r4)
ep	Element pointer (r30)
[reg]	Base register
disp	Displacement (Displacement from the address) 32 bits unless otherwise stated.
disp*n*	*n*-bit displacement
imm	Immediate 32 bits unless otherwise stated.
imm*n*	*n*-bit immediate
bit#3	3-bit data for bit number specification
cc#3	3-bit data for specifying CC0 to CC7 (bits 24 to 31) of the FPSR floating-point system register
#label	Absolute address reference of label
label	Offset reference of label in section or PC offset reference However, for a section allocated to a segment for which a tp symbol is to be generated, offset reference from the tp symbol is referred instead of offset in section
$label	gp offset reference of label
!label	Absolute address reference of label (without instruction expansion)
%label	Offset reference of ep
HIGHW(*value*)	Higher 16 bits of *value*
LOWW(*value*)	Lower 16 bits of *value*
HIGHW1(*value*)	Higher 16 bits of *value* + bit value[Note] of bit number 15 of *value*
HIGH(*value*)	Upper 8 bits of the lower 16 bits of *value*
LOW(*value*)	Lower 8 bits of *value*
addr	Address
PC	Program counter
PSW	Program status word
regID	System register number (0 to 31)
vector	Trap vector (0 to 31)

Symbols	Meaning
BITIO	Peripheral I/O register (for 1-bit manipulation only)

Note The bit number 0 is LSB (Least Significant Bit).

(2) Operand

This section describes the description format of operand in assembler. In assembler, register, constant, symbol, label reference, and expression that composes of constant, symbol, label reference, operator and parentheses can be specified as the operands for instruction, and directives.

(a) Register

The registers that can be specified with the assembler are listed below.[Note]

r0, zero, r1, r2, hp, r3, sp, r4, gp, r5, tp, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30, ep, r31, lp

Note For the ldsr and stsr instructions, the PSW, and system registers are specified by using the numbers. Further, in assembler, PC cannot be specified as an operand.

r0 and zero (Zero register), r2 and hp (Handler stack pointer), r3 and sp (Stack pointer), r4 and gp (Global pointer), r5 and tp (Text pointer), r30 and ep (Element pointer), r31 and lp (Link pointer) shows the same register.

(b) r0

r0 is the register which normally contains 0 value. This register does not substitute the result of an operation even if used as a destination register. When r0 is specified as a destination register, the assembler outputs the following message[Note], and then continues assembling.

Note Output of this message can be suppressed by specifying the warning message suppression option (-Xwarning_level) upon starting the assembler.

```
mov     0x10, r0
    ↓
W0550013: r0 used as destination register
```

(c) r1

The assembler-reserved register (r1) is used as a temporary register when instruction expansion is performed using the assembler. If r1 is specified as a source or destination register, the assembler outputs the following message[Note], then continues assembling.

Note Output of this message can be suppressed by specifying the warning message suppression option (-Xwarning_level) upon starting the assembler.

```
mov     0x10, r1
    ↓
W0550013: r1 used as destination register
```

```
mov     r1, r10
    ↓
W0550013: r1 used as source register
```

(d) Constants

As the constituents of the absolute expressions or relative expressions that can be used to specify the operands of the instructions and pseudo-instruction in the assembler, integer constants and character constants can be used. For the ld/st and bit manipulation instructions, a "peripheral I/O register name", defined in the device file, can also be specified as an operand. Thus enabling input/output of a port address. Moreover, floating-point constants can be used to specify the operand of the .float pseudo-instruction, and string constants can be used to specify the operand of the .set pseudo-instruction.

(e) Symbols

The assembler supports the use of symbols as the constituents of the absolute expressions or relative expressions that can be used to specify the operands of instructions and directives.

(f) Label Reference

In assembler, label reference can be used as a component of available relative value as shown in operand designation of instruction/directive.

- Memory reference instruction (Load/store instruction, and bit manipulation instruction)
- Operation instruction (arithmetic operation instruction, saturated operation instruction, logical operation instruction)
- Branch instruction
- Area reservation directive

In assembler, the meaning of a label reference varies with the reference method and the differences used in the instructions/directives Details are shown below.

Table 4-24. Label Reference

Reference Method	Instructions Used	Meaning
#label	Memory reference instruction, operation instruction and jmp instruction	The absolute address of the position at which the definition of label (label) exists (Offset from address 0[Note 1]). This has a 32-bit address and must be expanded into two instructions except mov instruction.
	Area reservation directive	The absolute address of the position at which the definition of label (label) exists (Offset from address 0[Note 1]). Note that the 32-bit address is a value masked in accordance with the size of the area secured.

Reference Method	Instructions Used	Meaning
!label	Memory reference instruction, operation instruction	The absolute address of the position at which the definition of label (label) exists (Offset from address 0 Note 1). This has a 16-bit address and cannot expand instructions if instructions with 16-bit displacement or immediate are specified. If any other instructions are specified, expansion into appropriate one instruction is possible. If the address defined by label (label) is not within a range expressible by 16 bits, an error will be occur at the time of link.
	Area reservation directive	The absolute address of the position at which the definition of label (label) exists (Offset from address 0 Note 1). Note that the 32-bit address is a value masked in accordance with the size of the area secured.
label	Memory reference instruction, operation instruction	The offset in the section of the position where definition of the label (label) exists (offset from the initial address of the section where the definition of label (label) exists Note 2). This has a 32-bit offset and must be expanded into two instructions. Note that for a section allocated to a segment for which a tp symbol is to be generated, the offset is referred from the tp symbol.
	Branch instruction except jmp instruction	The PC offset at the position where definition of label (label) exists (offset from the initial address of the instruction using the reference of label (label) Note 2).
	Area reservation directive	The offset in the section of the position where definition of the label (label) exists (offset from the initial address of the section where the definition of label (label) exists Note 2). Note that the 32-bit offset is a value masked in accordance with the size of the area secured.
%label	Memory reference instruction, operation instruction	The ep offset at the position where definition of the label (label) exists (offset from the address showing the element pointer). This has a 16-bit offset and cannot expand instructions if instructions with 16-bit displacement or immediate are specified. If any other instructions are specified, expansion into appropriate one instruction is possible. If the address defined by label (label) is not within a range expressible by 16 bits, an error will be occurred at the time of link.
	Area reservation directive	The ep offset at the position where definition of the label (label) exists (offset from the address showing the element pointer). Note that the 32-bit offset is a value masked in accordance with the size of the area secured.
$label	Memory reference instruction, operation instruction	The gp offset at the position where definition of the label (label) exists (offset from the address showing the global pointer Note 3).

Notes 1. The offset from address 0 in object module file after link.

2. The offset from the first address of the section (output section) in which the definition of label (label) exists is allocated in the linked object module file.

3. The offset from the address indicated by the value of text pointer symbol + value of the global pointer symbol for the segment to which the above output section is allocated.

The meanings of label references for memory reference instructions, operation instructions, branch instructions, and area allocation pseudo-instruction are shown below.

Table 4-25. Memory Reference Instruction

Reference Method	Meaning
#label[reg]	The absolute address of label (label) is treated as a displacement. This has a 32-bit value and must be expanded into two instructions. By setting #label[r0], reference by an absolute address can be specified. Part of [reg] can be omitted. If omitted, the assembler assumes that [r0] has been specified.
label[reg]	The offset in the section of label (label) is treated as a displacement. This has a 32-bit value and must be expanded into two instructions. By specifying a register indicating the first address of section as reg and thereby setting label[reg], general register relative reference can be specified. For a section allocated to a segment for which a tp symbol is to be generated, however, the offset from tp symbol is treated as a displacement.
$label[reg]	The gp offset of label (label) is treated as a displacement. This has either a 32-bit or 16-bit value, from the section defined by label (label), and pattern of instruction expansion changes accordingly Note. If an instruction with a 16-bit value is expanded and the offset calculated from the address defined by label (label) is not within a range that can be expressed in 16 bits, an error is output at the time of link. By setting $label [gp], relative reference of the gp register (called a gp offset reference) can be specified. Part of [reg] can be omitted. If omitted, the assembler assumes that [gp] has been specified.
!label[reg]	The absolute address of label (label) is treated as a displacement. This has a 16-bit value and instruction is not expanded. If the address defined by label (label) cannot be expressed in 16 bits, an error is output at the time of link. By setting !lable[r0], reference by an absolute address can be specified. Part of [reg] can be omitted. If omitted, the assembler assumes that [r0] has been specified. However, unlike #label[reg] reference, instruction expansion is not executed.
%label[reg]	The offset from the ep symbol in the position where definition of the label (label) exists is treated as a displacement. This either has a 16-bit value, or depending on the instruction a value lower than this, and if it is not a value that can be expressed within this range, an error is output at the time of link. Part of [reg] can be omitted. If omitted, the assembler assumes that [ep] has been specified.

Note See "(h) gp Offset Reference".

Table 4-26. Operation Instructions

Reference Method	Significance
#label	The absolute address of label (label) is treated as an immediate. This has a 32-bit value and must be expanded into two instructions.
label	The offset in the section of label (label) is treated as an immediate. This has a 32-bit value and must be expanded into two instructions. However, for a section allocated to a segment for which a tp symbol is to be generated, the offset from the tp symbol is treated as an immediate value.
$label	The gp offset of label (label) is treated as an immediate. This either has a 32-bit or 16-bit value, from the section defined by label (label), and pattern of instruction changes accordingly Note 1. If an instruction with a 16-bit value is expanded and the offset calculated from the address defined by label (label) is not within a range that can be expressed in 16 bits, an error is output at the time of link.
!label	The absolute address of label (label) is treated as an immediate. This has a 16-bit value. If operation instruction of an architecture for which a 16-bit value can be specify Note 2 as an immediate are specified, and instruction is not expanded. If the add, mov, and mulh instructions are specified, expansion into appropriate 1-instruction is possible. No other instructions can be specified. If the value is not within a range that can be expressed in 16 bits, an error is output at the time of link.
%label	The offset from the ep symbol in the position where definition of the label (label) exists is treated as an immediate. This has a 16-bit value. If operation instruction of an architecture for which a 16-bit value can be specify Note 2 as an immediate are specified, and instruction is not expanded. This reference method can be specified only for operation instructions of an architecture for which a 16-bit value can be specified as an immediate, and add, mov, and mulh instructions. If the add, mov, and mulh instructions are specified, expansion into appropriate 1-instruction is possible. No other instructions can be specified. If the value is not within a range that can be expressed in 16 bits, an error is output at the time of link.

Notes 1. See "(h) gp Offset Reference".

2. The instructions for which a 16-bit value can be specified as an immediate are the addi, andi, movea, mulhi, ori, satsubi, and xori instructions.

Table 4-27. Branch Instructions

Reference Method	Meaning
#label	In jmp instruction, the absolute address of label (label) is treated as a jump destination address. This has a 32-bit value and must be expanded into two instructions.
label	In branch instructions other than the jmp instruction, PC offset of the label (label) is treated as a displacement. This has a 22-bit value, and if it is not within a range that can be expressed in 22 bits, an error is output at the time of link.

Table 4-28. Area Reservation Directives

Reference Method	Meaning
#label !label	In .db4/.db2/.db directive, the absolute address of the label (label) is treated as a value. This has a 32-bit value, but is masked in accordance with the bit width of each directives
label %label	In .db4/.db2/.db directive, the offset in the section defined by label (label) is treated as a value. This has a 32-bit value, but is masked in accordance with the bit width of each directives
$label	In .db4/.db2/.db directive, the offset from the ep symbol in the position where definition of the label (label) exists is treated as an immediate. This has a 32-bit value, but is masked in accordance with the bit width of each directives

(g) ep Offset Reference

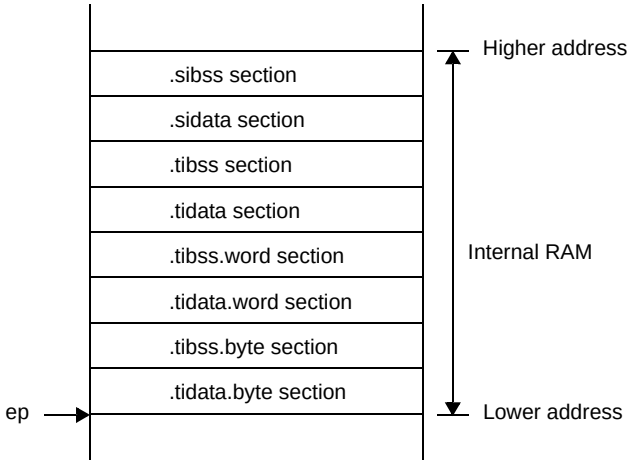
This section describes the ep offset reference. The CX assumes that data explicitly stored in internal RAM is shown below.

Reference through the offset from address indicated by the element pointer (ep).

Data in the internal RAM is divided into the following two groups.

- tidata/.tibss/.tidata.byte/.tibss.byte/.tidata.word/.tibss.word section (Data is referred by memory reference instructions (sld/sst) in a small code size)
- sidata/.sibss section (Data is referred by memory reference instructions (ld/st) in a large code size)

Figure 4-18. Memory Location Image of Internal RAM



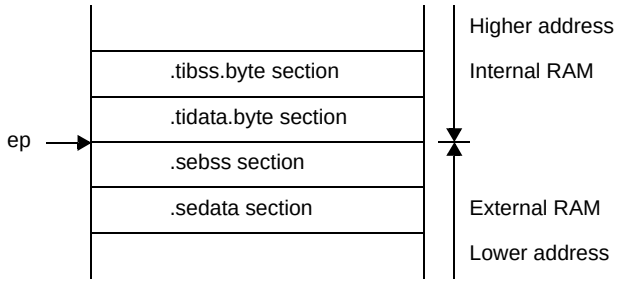
<1> Data Allocation

In internal RAM, data is allocated to the sections as follows:

- When developing a program in C
  Allocate data by specifying the "tidata", "tidata.byte", "tidata.word", or "sidata" section type in the "#pragma section" instruction.

- When developing a program in assembly language
  Data is allocated to the section of .tidata, .tibss, .tidata.byte, .tibss.byte, .tidata.word, .tibss.word, .sidata, or .sibss section type by the section definition directives ep offset reference can also be executed with respect to data in a specific range of external RAM by allocating the data to .sedata or .sebss sections in the same manner as above.

Figure 4-19. Memory Allocation Image for External RAM (.sedata/.sebss Section)



<2> Data Reference

As per the "Data Allocation" method explained above, the assembler generates a machine instruction string as follows.

- Generates a machine instruction by referring ep offset for %label reference to data allocated to the .tidata, .tibss, .tidata.byte, .tibss.byte, .tidata.word, .tibss.word, .sidata, .sibss, .sedata, or .sebss section.
- Generates a machine instruction string by referring offset in the section for %label reference to data allocated to other than that above.

Example

```
        .dseg   SIDATA
sidata: .db2    0xFFF0
        .dseg   DATA
data:   .db2    0xFFF0
        .cseg   TEXT
        ld.h    %sidata, r20    ; (1)
        ld.h    %data, r20      ; (2)
```

The assembler generates a machine instruction string for %label reference because: The assembler regards the code in (1) as being a reference by ep offset because the defined data is allocated to the .sidata section. The assembler regards the code in (2) as being a reference by in-section offset. The assembler performs processing, assuming that the data is allocated to the correct section. If the data is allocated to other than the correct section, it cannot be detected by the assembler.

Example

```
.dseg   TEXT
ld.h    %label[ep], r20
```

Instructions are coded to allocate a label to the .sidata section and to perform reference by ep offset. However, label is allocated to the .data section because of the allocation error. In this case, the assembler loads the data in the base register ep symbol value + offset value in the .data section of label.

(h) gp Offset Reference

This section describes the gp offset reference. The CX assumes that data stored in external RAM (other than .sedata/.sebss section explained on the previous page) is basically shown below.

Referred by the offset from the address indicated by global pointer (gp).

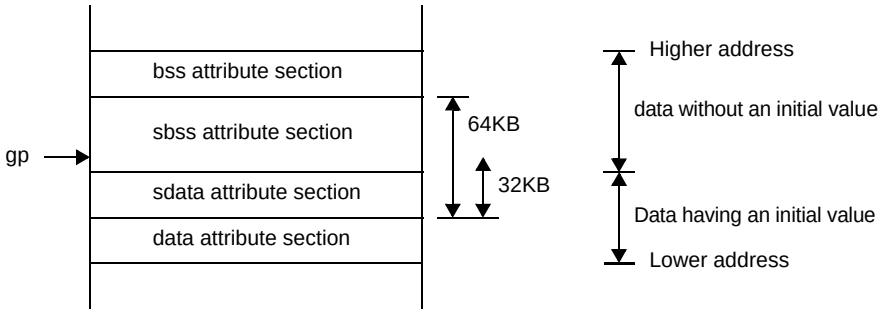
If r0-relative memory allocation for internal ROM or RAM is not done with the "#pragma section" command of C, or an assembly language section definition directive, all data is subject to gp offset reference.

<1> Data Allocation

The memory reference instruction (ld/st) of the machine instruction of the V850 microcontrollers can only accept 16-bit immediate as a displacement. For this reason, the CX classifies data into the following two types. Data of the former type is allocated to the sdata- or sbss-attribute section, while that of the latter type is allocated to the data- or bss-attribute section. Data having an initial value is allocated to the sdata/data-attribute section, while data without an initial value is allocated to the sbss/bss-attribute section. By default, the CX allocates data to the data/sdata/ sbss/bss-attribute sections, starting from the lowest address. Moreover, it is assumed that the global pointer (gp) is set by a startup module to point to the address resulting from addition of 32 KB to the first address of the sdata-attribute section.

- Data allocated to a memory range that can be referred by using the global pointer (gp) and a 16-bit displacement.
- Data allocated to a memory range that can be referred by using the global pointer (gp) and (constructed by many instructions) a 32-bit displacement.

Figure 4-20. Memory Location Image for gp Offset Reference Section



Remark The sum of sdata- and sbss-attribute sections is 64 KB. gp is 32 KB below the first byte of the sdata- attribute section.

Data in the sdata- and sbss-attribute sections can be referred by using a single instruction. To reference data in the data- and bss-attribute sections, however, two or more instructions are necessary. Therefore, the more data allocated to the sdata- and sbss-attribute sections, the higher the execution efficiency and object efficiency of the generated machine instructions. However, the size of the memory range that can be referred with a 16-bit displacement is limited.

If all the data cannot be allocated to the sdata- and sbss-attribute sections, it becomes necessary to determine which data is to be allocated to the sdata- and sbss-attribute sections.

The CX "allocates as much data as possible to the sdata- and sbss-attribute sections". By default, all data items are allocated to the sdata- and sbss-attribute sections. The data to be allocated can be selected as follows:

- When the -Xsdata option is specified.
  By specifying the -Xsdata=*num* option upon starting the C compiler or assembler, data of less than *num* bytes is allocated to the sdata- and sbss-attribute sections.
- When using a program to specify the section to which data will be allocated.
  Explicitly allocate data that will be frequently referred to the sdata- and sbss-attribute sections. For allocation, use a section definition directive when using the assembly language, or the #pragma section command when using C.

<2> Data Reference

Using the data allocation method explained above, the assembler generates a machine instruction string that performs:

- Reference by using a 16-bit displacement for gp offset reference to data allocated to the sdata- and sbss- attribute sections.
- Reference by using a 32-bit displacement (consisting of two or more machine instructions) for gp offset reference to data allocated to the data- and bss-attribute sections.

Example

```
        .dseg   DATA
data:   .db4    0xFFF00010      ; (1)
        .cseg   TEXT
        ld.w    $data[gp], r20  ; (2)
```

The assembler generates a machine instruction string, equivalent to the following instruction string for the ld.w instruction in (2), that performs gp offset reference of the data defined in (1).[Note]

```
movhi   HIGHW1($data), gp, r1
ld.w    LOWW($data)[r1], r20
```

Note See "(j) About HIGH/LOW/HIGHW/LOWW/HIGHW1", for details of HIGHW1/LOWW.

The assembler processes files on a one-by-one basis. Consequently, it can identify to which attribute section data having a definition in a specified file has been allocated, but cannot identify the section to which data not having a definition in a specified file has been allocated. Therefore, the assembler generates machine instructions as follows, when the -Xsdata=*num* option is specified at start-up, assuming that the allocation policy described above (i.e., data smaller than a specific size is allocated to the sdata- and sbss-attribute sections) is observed.

- Generates machine instructions that perform reference by using a 16-bit displacement for gp offset reference to data not having a definition in a specified file and which consists of less than num bytes.
- Generates a machine instruction string that performs reference by using a 32-bit displacement (consisting of two or more machine instructions) for gp offset reference to data having no definition in a specified file and which consists of more than num bytes.

To identify these conditions, however, the size of the data not having a definition in a specified file, and which is referred by a gp offset, must be identified. To develop a program in an assembly language, therefore, specify the size of the data (actually, a label for which there is no definition in a specified file and which is referred by a gp offset) for which there is no definition in a specified file, by using the .extern directives

```
.extern data, 4          ; (1)
.cseg   TEXT
ld.w    $data[gp], r20   ; (2)
```

When the -Xsdata=2 option is specified upon starting the assembler, the assembler generates a machine instruction string, equivalent to the following instruction string, for the ld.w instruction in (2) that performs gp offset reference to the data declared in (1).[Note]

```
movhi   HIGHW1($data), gp, r1
ld.w    LOWW($data)[r1], r20
```

Note See "(j) About HIGH/LOW/HIGHW/LOWW/HIGHW1", for details of HIGHW1/LOWW.

To develop a program in C, the C compiler of the CX automatically generates the .extern directive, thus output the code which specifies the size of data not having a definition in the specified file (actually, a label for which there is no definition in a specified file and which is referred by a gp offset).

Remark The handling of gp offset reference (specifically, memory reference instructions that use a relative expression having the gp offset of a label as their displacement) by the assembler is summarized below.

- If the data has a definition in a specified file.
  - If the data is to be allocated to the sdata- or sbss-attribute section[Note].
    Generates a machine instruction that performs reference by using a16-bit displacement.
  - If the data is not allocated to the sdata- or sbss-attribute section.
    Generates a machine instruction string that performs reference by using a 32-bit displacement.

Note If the value of the constant expression of a relative expression in the form of "label + constant expression" exceeds 16 bits, the assembler generates a machine instruction string that performs reference using a 32-bit displacement.

- If the data does not have a definition in a specified file.
  - If the -Xsdata=*num* option is specified upon starting the assembler.
    If a size of other than 0, but less than *num* bytes is specified for the data (label referred by gp offset) by the .comm/.extern/.globl/.public directives.
    Assumes that the data is to be allocated to the sdata- or sbss-attribute section and generates a machine instruction that performs reference by using a 16-bit displacement.
    Other than above, assumes that the data is not allocated to the sdata- or sbss-attribute section and generates a machine instruction string that performs reference using a 32-bit displacement

- If the -Xsdata option is not specified upon starting the assembler.
  Assumes that the data is to be allocated to the sdata- or sbss-attribute section and generates a machine instruction that performs reference by using a 16-bit displacement.

(i) Label references in multi-core

Below are described the differences between label references for multi-core and for single-core.

<1> If the "-Xmulti=pe*n*" (n: PE number) option is specified

- Data and code can be accessed using the same references as for single-core.

<2> If the "-Xmulti=cmn" option is specified

- Data and code are accessed using absolute addresses (offset from address 0), rather than referencing offset from the gp/ep/tp symbol.
- References offset from the gp/ep/tp symbol will cause an error.

(j) About HIGH/LOW/HIGHW/LOWW/HIGHW1

<1> To refer memory by using 32-bit displacement

The memory reference instruction (Load/store instructions) of the machine instructions of the V850 microcontrollers can take only a 16-bit immediate from displacement. Consequently, the assembler performs instruction expansion to refer the memory by using a 32-bit displacement, and generates an instruction string that performs the reference, by using the movhi and memory reference instructions and thereby constituting a 32-bit displacement from the higher 16 bits and lower 16 bits of the 32-bit displacement.

Example

`ld.w 0x18000[r11], r12`	`movhi HIGHW1(0x18000), r11, r1` `ld.w LOWW(0x18000)[r1], r12`

At this time, the memory reference instruction of machine instructions that uses the lower 16 bits as a displacement sign-extends the specified 16-bit displacement to a 32-bit value. To adjust the sign-extended bits, the assembler does not merely configure the displacement of the higher 16 bits by using the movhi instruction, instead it configures the following displacement.

Higher 16 bits + the most significant bit (bit of bit number 15) of the lower 16 bits

<2> HIGHW/LOWW/HIGHW1/HIGH/LOW

In the next table, the assembler can specify the higher 16 bits of a 32-bit value, the lower 16 bits of a 32-bit value, the value of the higher 16 bits + bit 15 of a 32-bit value, the higher 8 bits of a 16-bit value, and the lower 8 bits of a 16-bit value by using HIGHW, LOWW, HIGHW, HIGH, and LOW.[Note]

Note If this information cannot be internally resolved by the assembler, it is reflected in the relocation information and subsequently resolved by the link editor.

Table 4-29. Area Reservation Directives

HIGHW/LOWW/ HIGHW1/HIGH/LOW	Meaning
HIGHW (*value*)	Higher 16 bits of *value*
LOWW (*value*)	Lower 16 bits of *value*
HIGHW1 (*value*)	Higher 16 bits of *value* + bit value of bit number 15 of *value*
HIGH (*value*)	Upper 8 bits of the lower 16 bits of *value*
LOW (*value*)	Lower 8 bits of *value*

Example

```
        .dseg   DATA
L1:
            :
        .cseg   TEXT
        movhi   HIGHW $L1, r0, r10  ; Stores the higher 16 bits of the gp
                                    ; offset value of L1 in the higher 16 bits
                                    ; of r10, and the lower 16 bits to 0
        movea   LOWW $L1, r0, r10   ; Sign-extends the lower 16 bits of the gp
                                    ; offset of L1 and stores to r10
            :
        movhi   HIGHW1 $L1, r0, r1  ; Stores the gp offset value of L1 in r10
        movea   LOWW $L1, r1, r10
```

4.7.5 Description of instructions

This section describes the instructions of the assembly language supported by the assembler.

For details of the machine instructions generated by the assembler, see the "Each Device User Manual".

Instruction

Indicates the meaning of instruction.

[Syntax]

Indicates the syntax of instruction.

[Function]

Indicates the function of instruction.

[Description]

Indicates the operating method of instruction.

[Flag]

Indicates the operation of flag (PSW) by the execution of instruction.

However, in (set1, clr1, not1) bit operation instruction, indicates the flag value before execution.

"---" of table indicates that the flag value is not changed.

[Caution]

Indicates the caution in instruction.

4.7.6 Load/Store instructions

This section describes the load/store instructions. Next table lists the instructions described in this section.

Table 4-30. Load/Store Instructions

Instruction		Meaning
ld	ld.b	Byte data load
	ld.h	Halfword data load
	ld.w	Word data load
	ld.bu	Unsigned byte data load
	ld.hu	Unsigned halfword data load
sld	sld.b	Byte data load (short format)
	sld.h	Halfword data load (short format)
	sld.w	Word data load (short format)
	sld.bu	Unsigned byte data load (short format)
	sld.hu	Unsigned halfword data load (short format)
ld23	ld23.b	Byte data load
	ld23.h	Halfword data load
	ld23.w	Word data load
	ld23.bu	Unsigned byte data load
	ld23.hu	Unsigned halfword data load
st	st.b	Byte data store
	st.h	Halfword data store
	st.w	Word data store
sst	sst.b	Byte data store (short format)
	sst.h	Halfword data store (short format)
	sst.w	Word data store (short format)
st23	st.b	Byte data store
	st.h	Halfword data store
	st.w	Word data store

ld

Data is loaded.

[Syntax]

- ld.b disp[reg1], reg2
- ld.h disp[reg1], reg2
- ld.w disp[reg1], reg2
- ld.bu disp[reg1], reg2
- ld.hu disp[reg1], reg2

The following can be specified for displacement (disp):

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

The ld.b, ld.bu, ld.h, ld.hu, and ld.w instructions load data of 1 byte, 1 halfword, and 1 word, from the address specified by the first operand, int the register specified by the second operand.

[Description]

- If any of the following is specified for disp, the assembler generates one ld machine instruction[Note]. In the following explanations, ld denotes the ld.b/ld.h/ld.w/ld.bu/ld.hu instructions.

(a) Absolute expression having a value in the range of -32,768 to +32,767

`ld disp16[reg1], reg2`	`ld disp16[reg1], reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`ld $label[reg1], reg2`	`ld $label[reg1], reg2`

(c) Relative expression having !label or %label

`ld !label[reg1], reg2`	`ld !label[reg1], reg2`
`ld %label[reg1], reg2`	`ld %label[reg1], reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`ld disp16[reg1], reg2`	`ld disp16[reg1], reg2`

(e) Internal register name defined in the device file

`ld` *`register-name`*`[reg1], reg2`	`ld` *`register-name`*`[reg1], reg2`

Note The ld machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement

- If any of the following is specified for disp, the assembler performs instruction expansion to generate multiple machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`ld disp[reg1], reg2`	`movhi HIGHW1(disp), reg1, r1` `ld LOWW(disp)[r1], reg2`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`ld #label[reg1], reg2`	`movhi HIGHW1(#label), reg1, r1` `ld LOWW(#label)[r1], reg2`
`ld label[reg1], reg2`	`movhi HIGHW1(label), reg1, r1` `ld LOWW(label)[r1], reg2`
`ld $label[reg1], reg2`	`movhi HIGHW1($label), reg1, r1` `ld LOWW($label)[r1], reg2`

- If disp is omitted, the assembler assumes 0.
- If a relative expression having #label, or a relative expression having #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression having $label, or a relative expression having $label and with HIGHW, LOWW, or HIGHW1 applied, is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- ld.b and ld.h sign-extend the data of 1 byte and 1 halfword, respectively, and load the data into a register as 1 word.
- If a value that is not a multiple of 2 is specified as disp of ld.h, ld.w, or ld.hu, the assembler and link editor aligns disp with 2 and generates a code. Then, the assembler and link editor outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

- If r0 is specified as the second operand of ld.bu and ld.hu, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

sld

Data is loaded (short format).

[Syntax]

- sld.b disp7[ep], reg2
- sld.h disp8[ep], reg2
- sld.w disp8[ep], reg2
- sld.bu disp4[ep], reg2
- sld.hu disp5[ep], reg2

The following can be specified for displacement (disp4/5/7/8):

- Absolute expression having a value of up to 7 bits for sld.b, 8 bits for sld.h and sld.w, 4 bits for sld.bu, and 5 bits for sld.hu.
- Relative expression

[Function]

The sld.b, sld.bu, sld.h, sld.hu, and sld.w instructions load the data of 1 byte, 1 halfword, and 1 word, from the address obtained by adding the displacement specified by the first operand to the contents of register ep, to the register specified by the second operand.

[Description]

The assembler generates one sld machine instruction. Base register specification "[ep]" can be omitted.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- sld.b and sld.h sign-extend and store data of 1 byte and 1 halfword, respectively, in the register as 1 word.
- If a value that is not a multiple of 2 is specified as disp8 of sld.h or disp5 of sld.hu, and if a value that is not a multiple of 4 is specified as disp8 of sld.w, the assembler aligns disp8 or disp5 with multiples of 2 and 4, respectively, and generates a code. Then, the assembler and link editor outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

- If a value exceeding 127 is specified for disp7 of sld.b, a value exceeding 255 is specified for disp8 of sld.h and sld.w, a value exceeding 16 is specified for disp4 of sld.bu, and a value exceeding 32 is specified for disp5 of sld.hu, the assembler outputs the following message, and generates code in which disp7, disp8, disp4, and disp5 are masked with 0x7F, 0xFF, 0xF, and 0x1F, respectively.

```
W0550010: Illegal displacement in ld instruction.
```

- If r0 is specified as the second operand of the sld.bu and sld.hu, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

ld23

Data is loaded.

[Syntax]

- ld23.b disp23[reg1], reg2
- ld23.h disp23[reg1], reg2
- ld23.w disp23[reg1], reg2
- ld23.bu disp23[reg1], reg2
- ld23.hu disp23[reg1], reg2

The following can be specified for displacement (disp):

- Absolute expression having a value of up to 23 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

The ld23.b, ld23.bu, ld23.h, ld23.hu, and ld23.w instructions load data of 1 byte, 1 halfword, and 1 word, from the address specified by the first operand, int the register specified by the second operand.

[Description]

- If any of the following is specified for disp, the assembler generates one ld machine instruction[Note]. In the following explanations, ld denotes the ld.b/ld.h/ld.w/ld.bu/ld.hu instructions.

(a) Absolute expression having a value in the range of -4,194,304 to +4,194,303

`ld23 disp23[reg1], reg2`	`ld23 disp23[reg1], reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`ld23 $label[reg1], reg2`	`ld23 $label[reg1], reg2`

(c) Relative expression having !label or %label

`ld23 !label[reg1], reg2`	`ld23 !label[reg1], reg2`
`ld23 %label[reg1], reg2`	`ld23 %label[reg1], reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`ld23 disp16[reg1], reg2`	`ld23 disp16[reg1], reg2`

(e) Internal register name defined in the device file

`ld23 `*`register-name`*`[reg1], reg2`	`ld23 `*`register-name`*`[reg1], reg2`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`ld23 #label[reg1], reg2`	`ld23 #label[reg1], reg2`
`ld23 label[reg1], reg2`	`ld23 label[reg1], reg2`
`ld23 $label[reg1], reg2`	`ld23 $label[reg1], reg2`

Note The ld machine instruction takes an immediate value in the range of -4,194,304 to +4,194,303 (0xFFC00000 to 0x3FFFFF) as the displacement

- If disp23 is omitted, the assembler assumes 0.
- If a relative expression having #label, or a relative expression having #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression having $label, or a relative expression having $label and with HIGHW, LOWW, or HIGHW1 applied, is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- ld23.b and ld23.h sign-extend the data of 1 byte and 1 halfword, respectively, and load the data into a register as 1 word.
- If a value that is not a multiple of 2 is specified as disp of ld23.h, ld23.w, or ld23.hu, the assembler and link editor aligns disp with 2 and generates a code. Then, the assembler and link editor outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

- If r0 is specified as the second operand of ld.bu and ld.hu, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

st

Data is stored.

[Syntax]

- st.b reg2, disp[reg1]
- st.h reg2, disp[reg1]
- st.w reg2, disp[reg1]

The following can be specified as a displacement (disp):

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

The st.b, st.h, and st.w instructions store the data of the lower 1 byte, lower 1 halfword, and 1 word, respectively, of the register specified by the first operand to the address specified by the second operand.

[Description]

- If any of the following is specified as disp, the assembler generates one st machine instruction[Note]. In the following explanations, st denotes the st.b/st.h/st.w instructions.

(a) Absolute expression having a value in the range of -32,768 to +32,767

`st reg2, disp16[reg1]`	`st reg2, disp16[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`st reg2, $label[reg1]`	`st reg2, $label[reg1]`

(c) Relative expression having !label or %label

`st reg2, !label[reg1]`	`st reg2, !label[reg1]`
`st reg2, %label[reg1]`	`st reg2, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`st reg2, disp16[reg1]`	`st reg2, disp16[reg1]`

(e) Internal register name defined in the device file

`st reg2, register-name[reg1]`	`st reg2, register-name[reg1]`

Note The st machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement.

- If any of the following is specified as disp, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`st reg2, disp[reg1], reg2`	`movhi HIGHW1(disp), reg1, r1` `st reg2, LOWW(disp)[r1], reg2`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`st reg2, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `st reg2, LOWW(#label)[r1]`
`st reg2, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `st reg2, LOWW(label)[r1]`
`st reg2, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `st reg2, LOWW($label)[r1]`

- If disp is omitted, the assembler assumes 0.
- If a relative expression with #label, or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If a value that is not a multiple of 2 is specified as the disp of st.h or st.w, the assembler aligns disp with 2 and generates a code. Then, the assembler outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

sst

Data is stored (short format).

[Syntax]

- sst.b reg2, disp7[ep]
- sst.h reg2, disp8[ep]
- sst.w reg2, disp8[ep]

The following can be specified for displacement (disp7/8):

- Absolute expression having a value of up to 7 bits for sst.b or 8 bits for sst.h and sst.w
- Relative expression

[Function]

The sst.b, sst.h, and sst.w instructions store the data of the lower 1 byte, lower 1 halfword, and 1 word, respectively, of the register specified by the first operand to the address obtained by adding the displacement specified by the second operand to the contents of register ep.

[Description]

The assembler generates one sst machine instruction. Base register specification "[ep]" can be omitted.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If a value that is not a multiple of 2 is specified as disp8 of sst.h, and if a value that is not a multiple of 4 is specified as disp8 of sst.w, the assembler aligns disp8 with multiples of 2 and 4, respectively, and generates a code. Then, the assembler outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

- If a value exceeding 127 is specified as disp7 of sst.b, and if a value exceeding 255 is specified as disp8 of sst.h and sst.w, the assembler outputs the following message, and generates codes disp7 and disp8, masked with 0x7F and 0xFF, respectively.

W0550010: Illegal displacement in ld instruction.

st23

Data is stored.

[Syntax]

- st23.b reg2, disp23[reg1]
- st23.h reg2, disp23[reg1]
- st23.w reg2, disp23[reg1]

The following can be specified as a displacement (disp):

- Absolute expression having a value of up to 23 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

The st23.b, st23.h, and st23.w instructions store the data of the lower 1 byte, lower 1 halfword, and 1 word, respectively, of the register specified by the first operand to the address specified by the second operand.

[Description]

- If any of the following is specified as disp, the assembler generates one st machine instruction[Note]. In the following explanations, st denotes the st23.b/st23.h/st23.w instructions.

(a) Absolute expression having a value in the range of -4,194,304 to +4,194,303

`st23 reg2, disp23[reg1]`	`st23 reg2, disp23[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`st23 reg2, $label[reg1]`	`s23 reg2, $label[reg1]`

(c) Relative expression having !label or %label

`st23 reg2, !label[reg1]`	`st23 reg2, !label[reg1]`
`st23 reg2, %label[reg1]`	`st23 reg2, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`st23 reg2, disp16[reg1]`	`st23 reg2, disp16[reg1]`

(e) Internal register name defined in the device file

`st23 reg2, register-name[reg1]`	`st23 reg2, register-name[reg1]`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`st23 reg2, #label[reg1]`	`st23 reg2, #label[reg1]`
`st23 reg2, label[reg1]`	`st23 reg2, label[reg1]`
`st23 reg2, $label[reg1]`	`st23 reg2, $label[reg1]`

Note The st machine instruction takes an immediate value in the range of -4,194,304 to +4,194,303 (0xFFC00000 to 0x3FFFFF) as the displacement.

- If disp23 is omitted, the assembler assumes 0.
- If a relative expression with #label, or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp23, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If a value that is not a multiple of 2 is specified as the disp of st.h or st.w, the assembler aligns disp with 2 and generates a code. Then, the assembler outputs either one of the following messages.

W0550010: Illegal displacement in ld instruction.
W0560413: Relocated value(*value*) of relocation entry(file:*file*, section:*section*, offset:*offset*, type:*relocation type*) for load/store command become odd value.

4.7.7 Arithmetic operation instructions

This section describes the arithmetic operation instructions. Next table lists the instructions described in this section.

Table 4-31. Arithmetic Operation Instructions

Instruction	Meaning
add	Adds
addi	Adds (immediate)
adf	Adds with condition [V850E2]
sub	Subtracts
subr	Subtracts reverse
sbf	Subtracts with condition [V850E2]
mulh	Multiplies signed data (halfword)
mulhi	Multiplies signed data (halfword immediate)
mul	Multiplies signed data (word)
mulu	Multiplies unsigned data
mac	Multiplies and adds signed word data [V850E2]
macu	Multiplies and adds unsigned word data [V850E2]
divh	Divides signed data (halfword)
div	Divides signed data (word)
divhu	Divides unsigned data (halfword)
divu	Divides unsigned data (word)
divq	Divides signed word data (variable step) [V850E2V3]
divqu	Divides unsigned word data (variable step) [V850E2V3]
cmp	Compares
mov	Moves data
movea	Moves execution address
movhi	Moves higher half-word
mov32	Moves 32-bit data
cmov	Moves data depending on the flag condition
setf	Sets flag condition
sasf	Sets the flag condition after a logical left shift

See the device with an instruction set of V850E2V3 product user's manual and architecture edition for details about the device with an instruction set of V850E2V3.

add

Adds.

[Syntax]

- add reg1, reg2
- add imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "add reg1, reg2"
  Adds the value of the register specified by the first operand to the value of the register specified by the second operand, and stores the result into the register specified by the second operand.
- Syntax "add imm, reg2"
  Adds the value of the absolute expression or relative expression specified by the first operand to the value of the register specified by the second operand, and stores the result in the register specified by the second operand.

[Description]

- If this instruction is executed in syntax "add reg1, reg2", the assembler generates one add machine instruction.
- If the following is specified as imm in syntax "add imm, reg2", the assembler generates one add machine instruction[Note].

(a) Absolute expression having a value in the range of -16 to +15

`add imm5, reg`	`add imm5, reg`

Note The add machine instruction takes a register or immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the first operand.

- If the following is specified for imm in syntax "add imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`add imm16, reg`	`addi imm16, reg, reg`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`add imm, reg`	`movhi HIGHW(imm), r0, r1` `add r1, reg`

Else

`add imm, reg`	`mov imm, r1` `add r1, reg`

(c) Relative expression having !label or %label, or that having $label for a label with a definition in the sdata/sbss-attribute section

`add !label, reg`	`addi !label, reg, reg`
`add %label, reg`	`addi %label, reg, reg`
`add $label, reg`	`addi $label, reg, reg`

(d) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`add #label, reg`	`mov #label, r1` `add r1, reg`
`add label, reg`	`mov label, r1` `add r1, reg`
`add $label, reg`	`mov $label, r1` `add r1, reg`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

addi

Adds immediate.

[Syntax]

- addi imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

Adds the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied, specified by the first operand, to the value of the register specified by the second operand, and stores the result into the register specified by the third operand.

[Description]

- If the following is specified for imm, the assembler generates one addi machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`addi imm16, reg1, reg2`	`addi imm16, reg1, reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`addi $label, reg1, reg2`	`addi $label, reg1, reg2`

(c) Relative expression having !label or %label

`addi !label, reg1, reg2`	`addi !label, reg1, reg2`
`addi %label, reg1, reg2`	`addi %label, reg1, reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`addi imm16, reg1, reg2`	`addi imm16, reg1, reg2`

Note The addi machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF)as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`addi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `add reg1, reg2`

If all the lower 16 bits of the value of imm are 0

`addi imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `add reg1, r1`

Else

`addi imm, reg1, reg2`	`mov imm, reg2` `add reg1, reg2`

Other than above and when reg2 is r0

`addi imm, reg1, r0`	`mov imm, r1` `add reg1, r1`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

If reg2 is r0

`addi #label, reg1, r0`	`mov #label, r1` `add reg1, r1`
`addi label, reg1, r0`	`mov label, r1` `add reg1, r1`
`addi $label, reg1, r0`	`mov $label, r1` `add reg1, r1`

Else

`addi #label, reg1, reg2`	`mov #label, reg2` `add reg1, reg2`
`addi label, reg1, reg2`	`mov label, reg2` `add reg1, reg2`
`addi $label, reg1, reg2`	`mov $label, reg2` `add reg1, reg2`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

adf

Adds on condition flag. [V850E2]

[Syntax]

- adf imm4, reg1, reg2, reg3
- adf*cnd* reg1, reg2, reg3

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits (0xD cannot be specified)

[Function]

- Syntax "adf imm4, reg1, reg2, reg3"
  It compares the current flag condition with the flag condition indicated by the value of the lower 4 bits of the absolute expression (see "Table 4-32. adfcnd Instruction List") specified by the first operand.
  If the values match, adds the word data of the register specified by the second operand to the word data of the register specified by the third operand. And 1 is added to the addition result and that result is stored in the register specified by the fourth operand.
  If the values not match, adds the word data of the register specified by the second operand to the word data of the register specified by the third operand. And that result is stored in the register specified by the fourth operand.
- Syntax "adf*cnd* reg1, reg2, reg3"
  It compares the current flag condition with the flag condition indicated by the string in the *cnd*"part.
  If the values match, adds the word data of the register specified by the first operand to the word data of the register specified by the second operand. And 1 is added to the addition result and that result is stored in the register specified by the third operand.
  If the values not match, adds the word data of the register specified by the first operand to the word data of the register specified by the second operand. And that result is stored in the register specified by the third operand.

[Description]

- For the adf instruction, the assembler generates one adf machine instruction.
- For the adf*cnd* instruction, the assembler generates the corresponding adf instruction (see "Table 4-32. adfcnd Instruction List") and expands it to syntax "adf imm4, reg1, reg2, reg3".

Table 4-32. adf*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
adfgt	((S xor OV) or Z) = 0	Greater than (signed)	adf 0xF
adfge	(S xor OV) = 0	Greater than or equal (signed)	adf 0xE
adflt	(S xor OV) = 1	Less than (signed)	adf 0x6
adfle	((S xor OV) or Z) = 1	Less than or equal (signed)	adf 0x7
adfh	(CY or Z) = 0	Higher (Greater than)	adf 0xB
adfnl	CY = 0	Not lower (Greater than or equal)	adf 0x9
adfl	CY = 1	Lower (Less than)	adf 0x1
adfnh	(CY or Z) = 1	Not higher (Less than or equal)	adf 0x3
adfe	Z = 1	Equal	adf 0x2
adfne	Z = 0	Not equal	adf 0xA

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
adfv	OV = 1	Overflow	adf 0x0
adfnv	OV = 0	No overflow	adf 0x8
adfn	S = 1	Negative	adf 0x4
adfp	S = 0	Positive	adf 0xC
adfc	CY = 1	Carry	adf 0x1
adfnc	CY = 0	No carry	adf 0x9
adfz	Z = 1	Zero	adf 0x2
adfnz	Z = 0	Not zero	adf 0xA
adft	always 1	Always 1	adf 0x5

[Flag]

CY	1 if there is carry from MSB (Most Significant Bit), 0 if not
OV	1 if overflow occurred, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the adf instruction, the following message is output, and assembly continues using the lower 4 bits of the specified value.

```
W0550011: illegal operand (range error in immediate).
```

- If 0xD is specified as imm4 of the adf instruction, the following message is output, and assembly is stopped.

```
E0550261: illegal condition code.
```

sub

Subtracts.

[Syntax]

- sub reg1, reg2
- sub imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "sub reg1, reg2"
  Subtracts the value of the register specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the second operand.
- Syntax "sub imm, reg2"
  Subtracts the value of the absolute expression or relative expression specified by the first operand from the value of the register specified by the second operand, and stores the result into the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "sub reg1, reg2", the assembler generates one sub machine instruction.
- If the instruction is executed in syntax "sub imm, reg2", the assembler executes instruction expansion and generates one or more machine instructions[Note].

(a) 0

`sub 0, reg`	`sub r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`sub imm5, reg`	`mov imm5, r1` `sub r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`sub imm16, reg`	`movea imm16, r0, r1` `sub r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`sub imm, reg`	`movhi HIGHW(imm), r0, r1` `sub r1, reg`

Else

`sub imm, reg`	`mov imm, r1` `sub r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`sub $label, reg`	`movea $label, r0, r1` `sub r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`sub #label, reg`	`mov #label, r1` `sub r1, reg`
`sub label, reg`	`mov label, r1` `sub r1, reg`
`sub $label, reg`	`mov $label, r1` `sub r1, reg`

Note The sub machine instruction does not take an immediate value as an operand.

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit),0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

subr

Subtracts reverse.

[Syntax]

- subr reg1, reg2
- subr imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "subr reg1, reg2"
  Subtracts the value of the register specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the second operand.
- Syntax "subr imm, reg2"
  Subtracts the value of the absolute expression or relative expression specified by the first operand from the value of the register specified by the second operand, and stores the result into the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "subr reg1, reg2", the assembler generates one subr machine instruction.
- If the instruction is executed in syntax "subr imm, reg2", the assembler executes instruction expansion and generates one or more machine instructions[Note].

(a) 0

`subr 0, reg`	`subr r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`subr imm5, reg`	`mov imm5, r1` `subr r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`subr imm16, reg`	`movea imm16, r0, r1` `subr r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`subr imm, reg`	`movhi HIGHW(imm), r0, r1` `subr r1, reg`

Else

`subr imm, reg`	`mov imm, r1` `subr r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`subr $label, reg`	`movea $label, r0, r1` `subr r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`subr #label, reg`	`mov #label, r1` `subr r1, reg`
`subr label, reg`	`mov label, r1` `subr r1, reg`
`subr $label, reg`	`mov $label, r1` `subr r1, reg`

Note The subr machine instruction does not take an immediate value as an operand.

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	1 if a borrow occurs from MSB (Most Significant Bit),0 if not

sbf

Subtracts on condition flag. [V850E2]

[Syntax]

- sbf imm4, reg1, reg2, reg3
- sbf*cnd* reg1, reg2, reg3

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits (0xD cannot be specified)

[Function]

- Syntax "sbf imm4, reg1, reg2, reg3"
  It compares the current flag condition with the flag condition indicated by the value of the lower 4 bits of the absolute expression (see "Table 4-33. sbfcnd Instruction List") specified by the first operand.
  If the values match, subtracts the word data of the register specified by the second operand from the word data of the register specified by the third operand. And 1 is subtracted from the subtraction result and that result is stored in the register specified by the fourth operand.
  If the values not match, subtracts the word data of the register specified by the second operand from the word data of the register specified by the third operand. And that result is stored in the register specified by the fourth operand.
- Syntax "sbf*cnd* reg1, reg2, reg3"
  It compares the current flag condition with the flag condition indicated by the string in the "*cnd*" part.
  If the values match, subtracts the word data of the register specified by the first operand from the word data of the register specified by the second operand. And 1 is subtracted from the subtraction result and that result is stored in the register specified by the third operand.
  If the values not match, subtracts the word data of the register specified by the first operand from the word data of the register specified by the second operand. And that result is stored in the register specified by the third operand.

[Description]

- For the sbf instruction, the assembler generates one sbf machine instruction.
- For the adcond instruction, the assembler generates the corresponding sbf instruction (see "Table 4-33. sbfcnd Instruction List") and expands it to syntax "sbf imm4, reg1, reg2, reg3".

Table 4-33. sbf*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
sbfgt	((S xor OV) or Z) = 0	Greater than (signed)	sbf 0xF
sbfge	(S xor OV) = 0	Greater than or equal (signed)	sbf 0xE
sbflt	(S xor OV) = 1	Less than (signed)	sbf 0x6
sbfle	((S xor OV) or Z) = 1	Less than or equal (signed)	sbf 0x7
sbfh	(CY or Z) = 0	Higher (Greater than)	sbf 0xB
sbfnl	CY = 0	Not lower (Greater than or equal)	sbf 0x9
sbfl	CY = 1	Lower (Less than)	sbf 0x1
sbfnh	(CY or Z) = 1	Not higher (Less than or equal)	sbf 0x3

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
sbfe	Z = 1	Equal	sbf 0x2
sbfne	Z = 0	Not equal	sbf 0xA
sbfv	OV = 1	Overflow	sbf 0x0
sbfnv	OV = 0	No overflow	sbf 0x8
sbfn	S = 1	Negative	sbf 0x4
sbfp	S = 0	Positive	sbf 0xC
sbfc	CY = 1	Carry	sbf 0x1
sbfnc	CY = 0	No carry	sbf 0x9
sbfz	Z = 1	Zero	sbf 0x2
sbfnz	Z = 0	Not zero	sbf 0xA
sbft	always 1	Always 1	sbf 0x5

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit), 0 if not
OV	1 if overflow occurred, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the sbf instruction, the following message is output, and assembly continues using the lower 4 bits of the specified value.

```
W0550011: illegal operand (range error in immediate).
```

- If 0xD is specified as imm4 of the sbf instruction, the following message is output, and assembly is stopped.

```
E0550261: illegal condition code.
```

mulh

Multiplies half-word.

[Syntax]

- mulh reg1, reg2
- mulh imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 16 bits[Note]
- Relative expression

Note The assembler does not check whether the value of the expression exceeds 16 bits. The generated mulh instruction performs the operation by using the lower 16 bits.

[Function]

- Syntax "mulh reg1, reg2"
  Multiplies the value of the lower halfword data of the register specified by the first operand by the value of the lower halfword data of the register specified by the second operand as a signed value, and stores the result in the register specified by the second operand.
- Syntax "mulh imm, reg2"
  Multiplies the value of the lower halfword data of the absolute expression or relative expression specified by the first operand by the value of the lower halfword data of the register specified by the second operand as a signed value, and stores the result in the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "mulh reg1, reg2", the assembler generates one mulh machine instruction.
- If the following is specified as imm in syntax "mulh imm, reg2", the assembler generates one mulh machine instruction[Note].

(a) Absolute expression having a value in the range of -16 to +15

`mulh imm5, reg`	`mulh imm5, reg`

Note The mulh machine instruction takes a register or immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the first operand.

- If the following is specified for imm in syntax "mulh imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`mulh imm16, reg`	`mulhi imm16, reg, reg`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`mulh imm, reg`	`movhi HIGHW(imm), r0, r1` `mulh r1, reg`

Else

`mulh imm, reg`	`mov imm, r1` `mulh r1, reg`

(c) Relative expression having !label or %label, or that having $label for a label with a definition in the sdata/sbss-attribute section

`mulh !label, reg`	`mulhi !label, reg, reg`
`mulh %label, reg`	`mulhi %label, reg, reg`
`mulh $label, reg`	`mulhi $label, reg, reg`

(d) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`mulh #label, reg`	`mov #label, r1` `mulh r1, reg`
`mulh label, reg`	`mov label, r1` `mulh r1, reg`
`mulh $label, reg`	`mov $label, r1` `mulh r1, reg`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If r0 is specified by the second operand, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

mulhi

Multiplies half-word Immediate.

[Syntax]

- mulhi imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 16 bits[Note]
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

Note The assembler does not check whether the value of the expression exceeds 16 bits. The generated mulhi machine instruction performs the operation by using the lower 16 bits.

[Function]

Multiplies the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied specified by the first operand by the value of the register specified by the second operand, and stores the result in the register specified by the third operand.

[Description]

- If the following is specified for imm, the assembler generates one mulhi machine instruction[Noe].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`mulhi $label, reg1, reg2`	`mulhi $label, reg1, reg2`

(c) Relative expression having !label or %label

`mulhi !label, reg1, reg2`	`mulhi !label, reg1, reg2`
`mulhi %label, reg1, reg2`	`mulhi %label, reg1, reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

Note The mulhi machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`mulhi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `mulh reg1, reg2`

Else

`mulhi imm, reg1, reg2`	`mov imm, reg2` `mulh reg1, reg2`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`mulhi #label, reg1, reg2`	`mov #label, reg2` `mulhi reg1, reg2`
`mulhi label, reg1, reg2`	`mov label, reg2` `mulh reg1, reg2`
`mulhi $label, reg1, reg2`	`mov $label, reg2` `mulh reg1, reg2`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If r0 is specified by the second operand, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

mul

Multiplies word.

[Syntax]

- mul reg1, reg2, reg3
- mul imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "mul reg1, reg2, reg3"
  Multiplies the register value specified by the first operand by the register value specified by the second operand as a signed value and stores the lower 32 bits of the result in the register specified by the second operand, and the higher 32 bits in the register specified by the third operand. If the same register is specified by the second and third operands, the higher 32 bits of the multiplication result are stored in that register.
- Syntax "mul imm, reg2, reg3"
  Multiplies the value of the absolute or relative expression specified by the first operand by the register value specified by the second operand as a signed value and stores the lower 32 bits of the result in the register specified by the second operand, and the higher 32 bits in the register specified by the third operand. If the same register is specified by the second and third operands, the higher 32 bits of the multiplication result are stored in that register.

[Description]

- If the instruction is executed in syntax "mul reg1, reg2, reg3", the assembler generates one mul machine instruction.
- If the instruction is executed in syntax "mul imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions.

(a) 0

`mul 0, reg2, reg3`	`mul r0, reg2, reg3`

(b) Absolute expression having a value of other than 0 whithin the range of -256 to +255

`mul imm9, reg2, reg3`	`mul imm9, reg2, reg3`

(c) Absolute expression exceeding the range of -256 to +255, but within the range of -32,768 to +32,767

`mul imm16, reg2, reg3`	`movea imm16, r0, r1` `mul r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`mul imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mul r1, reg2, reg3`

Else

`mul imm, reg2, reg3`	`mov imm, r1` `mul r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`mul $label, reg2, reg3`	`movea $label, r0, r1` `mul r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`mul #label, reg2, reg3`	`mov #label, r1` `mul r1, reg2, reg3`
`mul label, reg2, reg3`	`mov label, r1` `mul r1, reg2, reg3`
`mul $label, reg2, reg3`	`mov $label, r1` `mul r1, reg2, reg3`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If these three conditions for the instructions in syntax "mul reg1, reg2, reg3" are met: reg1 and reg3 are the same register, reg2 is a different register from reg1 and reg3, and reg1 and reg3 are neither r0 nor r1, the assembler performs instruction expansion and generates multiple machine-language instructions.

```
mov     reg1, r1
mul     r1, reg2, reg3
```

- If these three conditions for the instructions in syntax "mul reg1, reg2, reg3" are met: reg1 and reg3 are the same register, reg2 is a different register from reg1 and reg3, and reg1 and reg3 are r1, the assembler outputs the following messages and stops assembling.

W0550013: register r1 used as source register
W0550013: register r1 used as destination register
E0550259: Cannot use r1 as destination in mul/mulu.

- If an instruction with the format "mul imm, reg2, reg3" meets the conditions that "reg3 is r1" and "reg2 is a different register than reg3", and it is possible that multiple machine-language instructions will be generated via instruction expansion, then the assembler outputs the following message, and assembly stops.

W0550013: register r1 used as destination register
E0550259: Cannot use r1 as destination in mul/mulu.

- If the warning message suppressing option -Xno_warning= W0550013 is specified, the assembler outputs the following message and stops assembling.

E0550259: Cannot use r1 as destination in mul/mulu.

mulu

Multiplies unsigned word.

[Syntax]

- mulu reg1, reg2, reg3
- mulu imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "mulu reg1, reg2, reg3"
  Multiplies the register value specified by the first operand by the register value specified by the second operand as an unsigned value and stores the lower 32 bits of the result in the register specified by the second operand, and the higher 32 bits in the register specified by the third operand. If the same register is specified by the second and third operands, the higher 32 bits of the multiplication result are stored in that register.
- Syntax "mulu imm, reg2, reg3"
  Multiplies the value of the absolute or relative expression specified by the first operand by the register value specified by the second operand as an unsigned value and stores the lower 32 bits of the result in the register specified by the second operand, and the higher 32 bits in the register specified by the third operand. If the same register is specified by the second and third operands, the higher 32 bits of the multiplication result are stored in that register.

[Description]

- If the instruction is executed in syntax "mulu reg1, reg2, reg3", the assembler generates one mulu machine instruction.
- If the instruction is executed in syntax "mulu imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions.

(a) 0

`mulu 0, reg2, reg3`	`mulu r0, reg2, reg3`

(b) Absolute expression having a value in the range of -256 to +255

`mulu imm9, reg2, reg3`	`mulu imm9, reg2, reg3`

(c) Absolute expression exceeding the range of -256 to +255, but within the range of -32,768 to +32,767

`mulu imm16, reg2, reg3`	`movea imm16, r0, r1` `mulu r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`mulu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mulu r1, reg2, reg3`

Else

`mulu imm, reg2, reg3`	`mov imm, r1` `mulu r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`mulu $label, reg2, reg3`	`movea $label, r0, r1` `mulu r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`mulu #label, reg2, reg3`	`mov #label, r1` `mulu r1, reg2, reg3`
`mulu label, reg2, reg3`	`mov label, r1` `mulu r1, reg2, reg3`
`mulu $label, reg2, reg3`	`mov $label, r1` `mulu r1, reg2, reg3`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If these three conditions for the instructions in syntax "mulu reg1, reg2, reg3" are met: reg1 and reg3 are the same register, reg2 is a different register from reg1 and reg3, and reg1 and reg3 are neither r0 nor r1, the assembler performs instruction expansion and generates multiple machine-language instructions.

```
mov     reg1, r1
mulu    r1, reg2, reg3
```

- If these three conditions for the instructions in syntax "mulu reg1, reg2, reg3" are met: reg1 and reg3 are the same register, reg2 is a different register from reg1 and reg3, and reg1 and reg3 are r1, the assembler outputs the following messages and stops assembling.

W0550013: register r1 used as source register
W0550013: register r1 used as destination register
E0550259: Cannot use r1 as destination in mul/mulu.

- If an instruction with the format "mulu imm, reg2, reg3" meets the conditions that "reg3 is r1" and "reg2 is a different register than reg3", and it is possible that multiple machine-language instructions will be generated via instruction expansion, then the assembler outputs the following message, and assembly stops.

W0550013: register r1 used as destination register
E0550259: Cannot use r1 as destination in mul/mulu.

- If the warning message suppressing option -Xno_warning=W0550013 is specified, the assembler outputs the following message and stops assembling.

E0550259: Cannot use r1 as destination in mul/mulu.

mac

Multiplies and adds signed word data. [V850E2]

[Syntax]

- mac reg1, reg2, reg3, reg4

[Function]

Adds the multiplication result of the general-purpose register reg2 word data and the general-purpose register reg1 word data with the 64-bit data made up of general-purpose register reg3 as the lower 32 bits and general-purpose register reg3+1 (for example, if reg3 were r6, "reg3+1" would be r7) as the upper 32 bits, and stores the upper 32 bits of that result (64-bit data) in general-purpose register reg4+1 and the lower 32 bits in general-purpose register reg4.

The contents of general-purpose registers reg1 and reg2 are treated as 32-bit signed integers.

General-purpose registers reg1, reg2, reg3, and reg3+1 are unaffected.

[Description]

The assembler generates one mac machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- The general-purpose registers that can be specified to reg3 or reg4 are limited to even numbered registers (r0, r2, r4, ..., r30). When specifying an odd numbered register, the following message is output, and assembly continues, specifying the register as an even numbered register (r0, r2, r4, ..., r30).

```
W0550026: illegal register number, aligned odd register(rXX) to be even register(rYY).
```

macu

Multiply and adds unsigned word data. [V850E2]

[Syntax]

- macu reg1, reg2, reg3, reg4

[Function]

Adds the multiplication result of the general-purpose register reg2 word data and the general-purpose register reg1 word data with the 64-bit data made up of general-purpose register reg3 as the lower 32 bits and general-purpose register reg3+1 (for example, if reg3 were r6, "reg3+1" would be r7) as the upper 32 bits, and stores the upper 32 bits of that result (64-bit data) in general-purpose register reg4+1 and the lower 32 bits in general-purpose register reg4.

The contents of general-purpose registers reg1 and reg2 are treated as 32-bit unsigned integers.

General-purpose registers reg1, reg2, reg3, and reg3+1 are unaffected.

[Description]

The assembler generates one macu machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- The general-purpose registers that can be specified to reg3 or reg4 are limited to even numbered registers (r0, r2, r4, ..., r30). When specifying an odd numbered register, the following message is output, and assembly continues, specifying the register as an even numbered register (r0, r2, r4, ..., r30).

```
W0550026: illegal register number, aligned odd register(rXX) to be even register(rYY).
```

divh

Divides half-word.

[Syntax]

- divh reg1, reg2
- divh imm, reg2
- divh reg1, reg2, reg3
- divh imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 16 bits[Note]
- Relative expression

Note The assembler does not check whether the value of the expression exceeds 16 bits. The generated machine instruction performs execution using the lower 16 bits.

[Function]

- Syntax "divh reg1, reg2"
  Divides the register value specified by the second operand by the value of the lower halfword data of the register specified by the first operand as a signed value, and stores the quotient in the register specified by the second operand.
- Syntax "divh imm, reg2"
  Divides the register value specified by the second operand by the value of the lower halfword data of the absolute or relative expression specified by the first operand as a signed value and stores the quotient in the register specified by the second operand.
- Syntax "divh reg1, reg2, reg3"
  Divides the register value specified by the second operand by the value of the lower halfword data of the register specified by the first operand as a signed value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.
- Syntax "divh imm, reg2, reg3"
  Divides the register value specified by the second operand by the value of the lower halfword data of the absolute or relative expression specified by the first operand as a signed value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.

[Description]

- If the instruction is executed in syntaxes "divh reg1, reg2" and "divh reg1, reg2, reg3", the assembler generates one divh machine instruction.
- If the instruction is executed in syntax "divh imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`divh 0, reg`	`divh r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`divh imm5, reg`	`mov imm5, r1` `divh r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`divh imm16, reg`	`movea imm16, r0, r1` `divh r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`divh imm, reg`	`movhi HIGHW(imm), r0, r1` `divh r1, reg`

Else

`divh imm, reg`	`mov imm, r1` `divh r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`divh $label, reg`	`movea $label, r0, r1` `divh r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`divh #label, reg`	`mov #label, r1` `divh r1, reg`
`divh label, reg`	`mov label, r1` `divh r1, reg`
`divh $label, reg`	`mov $label, r1` `divh r1, reg`

Note The divh machine instruction does not take an immediate value as an operand.

- If the instruction is executed in syntax "divh imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions.

(a) 0

`divh 0, reg2, reg3`	`divh r0, reg2, reg3`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`divh imm5, reg2, reg3`	`mov imm5, r1` `divh r1, reg2, reg3`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`divh imm16, reg2, reg3`	`movea imm16, r0, r1` `divh r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`divh imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `divh r1, reg2, reg3`

Else

`divh imm, reg2, reg3`	`mov imm, r1` `divh r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`divh $label, reg2, reg3`	`movea $label, r0, r1` `divh r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`divh #label, reg2, reg3`	`mov #label, r1` `divh r1, reg2, reg3`
`divh label, reg2, reg3`	`mov label, r1` `divh r1, reg2, reg3`
`divh $label, reg2, reg3`	`mov $label, r1` `divh r1, reg2, reg3`

[Flag]

CY	---
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If r0 is specified by the first operand in syntax "divh reg1, reg2", the CX outputs the following message and stops assembling.

E0550239: Illegal operand (cannot use r0 as source in V850E mode).
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

- If r0 is specified by the second operand (reg2) in syntaxes "divh imm, reg2", the assembler outputs the message and stops assembling.

E0550239: Illegal operand (cannot use r0 as source in V850E mode).
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

- If 0 is specified by the second operand (imm) in syntaxes "divh imm, reg2", the assembler outputs the message and stops assembling.

E0550239: Illegal operand (cannot use r0 as source in V850E mode).

div

Divides word.

[Syntax]

- div reg1, reg2, reg3
- div imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "div reg1, reg2, reg3"
  Divides the register value specified by the second operand by the register value specified by the first operand as a signed value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.
- Syntax "div imm, reg2, reg3"
  Divides the register value specified by the second operand by the value of the absolute or relative expression specified by the first operand as a signed value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.

[Description]

- If the instruction is executed in syntax "div reg1, reg2, reg3", the assembler generates one div machine instruction.
- If the instruction is executed in syntax "div imm, reg2, reg3", the assembler executes instruction expansion to generate two or more machine instructions[Note].

(a) 0

`div 0, reg2, reg3`	`div r0, reg2, reg3`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`div imm5, reg2, reg3`	`mov imm5, r1` `div r1, reg2, reg3`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`div imm16, reg2, reg3`	`movea imm16, r0, r1` `div r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`div imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `div r1, reg2, reg3`

Else

`div imm, reg2, reg3`	`mov imm, r1` `div r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`div $label, reg2, reg3`	`movea $label, r0, r1` `div r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`div #label, reg2, reg3`	`mov #label, r1` `div r1, reg2, reg3`
`div label, reg2, reg3`	`mov label, r1` `div r1, reg2, reg3`
`div $label, reg2, reg3`	`mov $label, r1` `div r1, reg2, reg3`

Note The div machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

divhu

Divides unsigned half-word.

[Syntax]

- divhu reg1, reg2, reg3
- divhu imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 16 bits[Note]
- Relative expression

Note The assembler does not check whether the value of the expression exceeds 16 bits. The generated machine instruction uses only the lower 16 bits for execution.

[Function]

- Syntax "divhu reg1, reg2, reg3"
  Divides the register value specified by the second operand by the value of the lower halfword data of the register value specified by the first operand as an unsigned value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.
- Syntax "divhu imm, reg2, reg3"
  Divides the register value specified by the second operand by the value of the lower halfword data of the absolute or relative expression specified by the first operand as an unsigned value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.

[Description]

- If the instruction is executed in syntax "divhu reg1, reg2, reg3", the assembler generates one divhu machine instruction.
- If the instruction is executed in syntax "divhu imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`divhu 0, reg2, reg3`	`divhu r0, reg2, reg3`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`divhu imm5, reg2, reg3`	`mov imm5, r1` `divhu r1, reg2, reg3`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`divhu imm16, reg2, reg3`	`movea imm16, r0, r1` `divhu r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`divhu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `divhu r1, reg2, reg3`

Else

`divhu imm, reg2, reg3`	`mov imm, r1` `divhu r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`divhu $label, reg2, reg3`	`movea $label, r0, r1` `divhu r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`divhu #label, reg2, reg3`	`mov #label, r1` `divhu r1, reg2, reg3`
`divhu label, reg2, reg3`	`mov label, r1` `divhu r1, reg2, reg3`
`divhu $label, reg2, reg3`	`mov $label, r1` `divhu r1, reg2, reg3`

Note The divhu machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

divu

Divides unsigned word.

[Syntax]

- divu reg1, reg2, reg3
- divu imm, reg2, reg3

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "divu reg1, reg2, reg3"
  Divides the register value specified by the second operand by the register value specified by the first operand as an unsigned value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.
- Syntax "divu imm, reg2, reg3"
  Divides the register value specified by the second operand by the value of the absolute or relative expression specified by the first operand as an unsigned value and stores the quotient in the register specified by the second operand, and the remainder in the register specified by the third operand. If the same register is specified by the second and third operands, the remainder is stored in that register.

[Description]

- If the instruction is executed in syntax "divu reg1, reg2, reg3", the assembler generates one divu machine instruction.
- If the instruction is executed in syntax "divu imm, reg2, reg3", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`divu 0, reg2, reg3`	`divu r0, reg2, reg3`

(b) Absolute expression having a value of other than 0 whithin the range of -16 to +15

`divu imm5, reg2, reg3`	`mov imm5, r1` `divu r1, reg2, reg3`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`divu imm16, reg2, reg3`	`movea imm16, r0, r1` `divu r1, reg2, reg3`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`divu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `divu r1, reg2, reg3`

Else

`divu imm, reg2, reg3`	`mov imm, r1` `divu r1, reg2, reg3`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`divu $label, reg2, reg3`	`movea $label, r0, r1` `divu r1, reg2, reg3`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`divu #label, reg2, reg3`	`mov #label, r1` `divu r1, reg2, reg3`
`divu label, reg2, reg3`	`mov label, r1` `divu r1, reg2, reg3`
`divu $label, reg2, reg3`	`mov $label, r1` `divu r1, reg2, reg3`

Note The divu machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

cmp

Compares.

[Syntax]

- cmp reg1, reg2
- cmp imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "cmp reg1, reg2"
  Compares the value of the register specified by the first operand with the value of the register specified by the second operand, and indicates the result using a flag. Comparison is performed by subtracting the value of the register specified by the first operand from the value of the register specified by the second operand.
- Syntax "cmp imm, reg2"
  Compares the value of the absolute expression or relative expression specified by the first operand with the value of the register specified by the second operand, and indicates the result using a flag. Comparison is performed by subtracting the value of the register specified by the first operand from the value of the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "cmp reg1, reg2", the assembler generates one cmp machine instruction.
- If the following is specified as imm in syntax "cmp imm, reg2", the assembler generates one cmp machine instruction[Note].

(a) Absolute expression having a value in the range of -16 to +15

`cmp imm5, reg`	`cmp imm5, reg`

Note The cmp machine instruction takes a register or immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the first operand.

- If the following is specified as imm in syntax "cmp imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`cmp imm16, reg`	`movea imm16, r0, r1` `cmp r1, reg`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`cmp imm, reg`	`movhi HIGHW(imm), r0, r1` `cmp r1, reg`

Else

`cmp imm, reg`	`mov imm, r1` `cmp r1, reg`

(c) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`cmp $label, reg`	`movea $label, r0, r1` `cmp r1, reg`

(d) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`cmp #label, reg`	`mov #label, r1` `cmp r1, reg`
`cmp label, reg`	`mov label, r1` `cmp r1, reg`
`cmp $label, reg`	`mov $label, r1` `cmp r1, reg`

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit),0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

mov

Moves.

[Syntax]

- mov reg1, reg2
- mov imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "mov reg1, reg2"
  Stores the value of the register specified by the first operand in the register specified by the second operand.
- Syntax "mov imm, reg2"
  Stores the value of the absolute expression or relative expression specified by the first operand in the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "mov reg1, reg2", the assembler generates one mov machine instruction.
- If the following is specified as imm in syntax "mov imm, reg2", the assembler generates one mov machine instruction[Note].

(a) Absolute expression having a value in the range of -16 to +15

`mov imm5, reg`	`mov imm5, reg`

Note The mov machine instruction takes a register or immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the first operand.

- If the following is specified as imm in syntax "mov imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`mov imm16, reg`	`movea imm16, r0, reg`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`mov imm, reg`	`movhi HIGHW(imm), r0, reg`

Else[Note]

`mov imm, reg`	`mov imm, reg`

Note A 16-bit mov instruction is replaced by a 48-bit mov instruction.

(c) Relative expression having !label or %label, or that having $label for a label with a definition in the sdata/sbss-attribute section

`mov !label, reg`	`movea !label, r0, reg`
`mov %label, reg`	`movea %label, r0, reg`
`mov $label, reg`	`movea $label, r0, reg`

(d) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section[Note]

`mov #label, reg`	`mov #label, reg`
`mov label, reg`	`mov label, reg`
`mov $label, reg`	`mov $label, reg`

Note A 16-bit mov instruction is replaced by a 48-bit mov instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value in the range between -16 and 15 is specified by the first operand and r0 is specified by the second operand of syntax "mov imm, reg2", or r0 is specified by the second operand of syntax "mov reg1, reg2", the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

movea

Moves execution address.

[Syntax]

- movea imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

Adds the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied, specified by the first operand, to the value of the register specified by the second operand, and stores the result in the register specified by the third operand.

[Description]

- If the following is specified for imm, the assembler generates one movea machine instruction[Note].
- If r0 is specified by reg1, the assembler recognizes specified syntax "mov imm, reg2".

(a) Absolute expression having a value in the range of -32,768 to +32,767

`movea imm16, reg1, reg2`	`movea imm16, reg1, reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`movea $label, reg1, reg2`	`movea $label, reg1, reg2`

(c) Relative expression having !label or %label

`movea !label, reg1, reg2`	`movea !label, reg1, reg2`
`movea %label, reg1, reg2`	`movea %label, reg1, reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`movea imm16, reg1, reg2`	`movea imm16, reg1, reg2`

Note The movea machine instruction takes an immediate value in a range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`movea imm, reg1, reg2`	`movhi HIGHW(imm), reg1, reg2`

Else

`movea imm, reg1, reg2`	`movhi HIGHW1(imm), reg1, r1` `movea LOWW(imm), r1, reg2`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`movea #label, reg1, reg2`	`movhi HIGHW1(#label), reg1, r1` `movea LOWW(#label), r1, reg2`
`movea label, reg1, reg2`	`movhi HIGHW1(label), reg1, r1` `movea LOWW(label), r1, reg2`
`movea $label, reg1, reg2`	`movhi HIGHW1($label), reg1, r1` `movea LOWW($label), r1, reg2`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If r0 is specified by the third operand, the assembler outputs the message and stops assembling.

movhi

Moves higher half-word.

[Syntax]

- movhi imm16, reg1, reg2

The following can be specified for imm16:

- Absolute expression having a value of up to 16 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

Adds word data for which the higher 16 bits are specified by the first operand and the lower 16 bits are 0, to the value of the register specified by the second operand, and stores the result in the register specified by the third operand.

[Description]

The assembler generates one movhi machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 65,535 is specified as imm16, the assembler outputs the following message and stops assembling.

```
E0550231: illegal operand (range error in immediate)
```

- If r0 is specified by the third operand, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

mov32

Moves 32-bit data.

[Syntax]

- mov32 imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

Stores the value of the absolute or relative expression specified as the first operand in the register specified as the second operand.

[Description]

The assembler generates one 48-bit machine language mov instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

cmov

Moves data depending on the flag condition.

[Syntax]

- cmov imm4, reg1, reg2, reg3
- cmov imm4, imm, reg2, reg3
- cmov*cnd* reg1, reg2, reg3
- cmov*cnd* imm, reg2, reg3

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits[Note]

Note The cmov machine instruction takes an immediate value in the range of 0 to 15 (0x0 to 0xF) as the first operand.

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "cmov imm4, reg1, reg2, reg3"
  Compares the flag condition indicated by the value of the lower 4 bits of the value of the constant expression specified by the first operand with the current flag condition. If a match is found, the register value specified by the second operand is stored in the register specified by the fourth operand; otherwise, the register value specified by the third operand is stored in the register specified by the fourth operand.
- Syntax "cmov imm4, imm, reg2, reg3"
  Compares the flag condition indicated by the value of the lower 4 bits of the constant expression specified by the first operand with the current flag condition. If a match is found, the value of the absolute expression specified by the second operand is stored in the register specified by the fourth operand; otherwise, the register value specified by the third operand is stored in the register specified by the fourth operand.
- Syntax "cmov*cnd* reg1, ret2, reg3"
  Compares the flag condition indicated by string *cnd* with the current flag condition. If a match is found, the register value specified by the first operand is stored in the register specified by the third operand; otherwise, the register value specified by the second operand is stored in the register specified by the third operand.
- Syntax "cmov*cnd* imm, reg2, reg3"
  Compares the flag condition indicated by string *cnd* with the current flag condition. If a match is found, the value of the absolute expression specified by the first operand is stored in the register specified by the third operand; otherwise, the register value specified by the second operand is stored in the register specified by the third operand.

Table 4-34. cmov*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
`cmovgt`	((S xor OV) or Z) = 0	Greater than (signed)	`cmov 0xF`
`cmovge`	(S xor OV) = 0	Greater than or equal (signed)	`cmov 0xE`
`cmovlt`	(S xor OV) = 1	Less than (signed)	`cmov 0x6`
`cmovle`	((S xor OV) or Z) = 1	Less than or equal (signed)	`cmov 0x7`
`cmovh`	(CY or Z) = 0	Higher (Greater than)	`cmov 0xB`

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
cmovnl	CY = 0	Not lower (Greater than or equal)	cmov 0x9
cmovl	CY = 1	Lower (Less than)	cmov 0x1
cmovnh	(CY or Z) = 1	Not higher (Less than or equal)	cmov 0x3
cmove	Z = 1	Equal	cmov 0x2
cmovne	Z = 0	Not equal	cmov 0xA
cmovv	OV = 1	Overflow	cmov 0x0
cmovnv	OV = 0	No overflow	cmov 0x8
cmovn	S = 1	Negative	cmov 0x4
cmovp	S = 0	Positive	cmov 0xC
cmovc	CY = 1	Carry	cmov 0x1
cmovnc	CY = 0	No carry	cmov 0x9
cmovz	Z = 1	Zero	cmov 0x2
cmovnz	Z = 0	Not zero	cmov 0xA
cmovt	always 1	Always 1	cmov 0x5
cmovsa	SAT = 1	Saturated	cmov 0xD

[Description]

- If the instruction is executed in syntax "cmov imm4, reg1, reg2, reg3", the assembler generates one cmov machine instruction[Note].

Note The cmov machine instruction takes an immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the second operand.

- If the following is specified as imm in syntax "cmov imm4, imm, reg2, reg3", the assembler generates one cmov machine instruction.

(a) Absolute expression having a value in the range of -16 to +15

If all the lower 16 bits of the value of imm are 0

`cmov imm4, imm5, reg2, reg3`	`cmov imm4, imm5, reg2, reg3`

- If the following is specified as imm in syntax "cmov imm4, imm, reg2, reg3", the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`cmov imm4, imm16, reg2, reg3`	`movea imm16, r0, r1` `cmov imm4, r1, reg2, reg3`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`cmov imm4, imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `cmov imm4, r1, reg2, reg3`

Else

`cmov imm4, imm, reg2, reg3`	`mov imm, r1` `cmov imm4, r1, reg2, reg3`

(c) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`cmov imm4, #label, reg2, reg3`	`mov #label, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, label, reg2, reg3`	`mov label, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, $label, reg2, reg3`	`mov $label, r1` `cmov imm4, r1, reg2, reg3`

(d) Relative expression having !label or %label, or that having $label for a label with a definition in the sdata/sbss-attribute section

`cmov imm4, !label, reg2, reg3`	`movea !label, r0, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, %label, reg2, reg3`	`movea %label, r0, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, $label, reg2, reg3`	`movea $label, r0, r1` `cmov imm4, r1, reg2, reg3`

- If the instruction is executed in syntax "cmov*cnd* reg1, ret2, reg3", the assembler generates the corresponding cmov instruction (see "Table 4-34. cmovcnd Instruction List") and expands it to syntax "cmov imm4, reg1, reg2, reg3".
- If the following is specified as imm in syntax "cmov*cnd* imm, reg2, reg3", the assembler generates the corresponding cmov instruction (see "Table 4-34. cmovcnd Instruction List") and expands it to syntax "cmov imm4, imm, reg2, reg3".

(a) Absolute expression having a value in the range of -16 to +15

- If the following is specified as imm in syntax "cmov*cnd* imm, reg2, reg3", the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`cmov`*`cnd`* `imm16, reg2, reg3`	`movea imm16, r0, r1` `cmov`*`cnd`* `r1, reg2, reg3`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

cmov*cnd* imm, reg2, reg3	movhi HIGHW(imm), r0, r1 cmov*cnd* r1, reg2, reg3

Else

cmov*cnd* imm, reg2, reg3	mov imm, r1 cmov*cnd* r1, reg2, reg3

(c) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

cmov*cnd* #label, reg2, reg3	mov #label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* label, reg2, reg3	mov label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* $label, reg2, reg3	mov $label, r1 cmov*cnd* r1, reg2, reg3

(d) Relative expression having !label or %label, or that having $label for a label with a definition in the sdata/sbss-attribute section

cmov*cnd* !label, reg2, reg3	movea !label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* %label, reg2, reg3	movea %label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* $label, reg2, reg3	movea $label, r0, r1 cmov*cnd* r1, reg2, reg3

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If a constant expression having a value exceeding 4 bits is specified as imm4 of the cmov instruction, the assembler outputs the following message.
  If the value exceeds 4 bits, the assembler masks the value with 0xF and continues assembling.

W0550011: illegal operand (range error in immediate)

setf

Sets flag condition.

[Syntax]

- setf imm4, reg
- setf*cnd* reg

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits

[Function]

- Syntax "setf imm4, reg"
  Compares the status of the flag specified by the value of the lower 4 bits of the absolute expression specified by the first operand with the current flag condition. If they are found to match, 1 is stored in the register specified by the second operand; otherwise, 0 is stored in the register specified by the second operand.
- Syntax "setf*cnd* reg"
  Compares the status of the flag indicated by string *cnd* with the current flag condition. If they are found to match, 1 is stored in the register specified by the second operand; otherwise, 0 is stored in the register specified by the second operand.

[Description]

- If the instruction is executed in syntax"setf imm4, reg",the assembler generates one satf machine instruction.
- If the instruction is executed in syntax "setf*cnd* reg", the assembler generates the corresponding setf instruction (see "Table 4-35. setfcnd Instruction List") and expands it to syntax "setf imm4, reg".

Table 4-35. setf*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
setfgt	((S xor OV) or Z) = 0	Greater than (signed)	setf 0xF
setfge	(S xor OV) = 0	Greater than or equal (signed)	setf 0xE
setflt	(S xor OV) = 1	Less than (signed)	setf 0x6
setfle	((S xor OV) or Z) = 1	Less than or equal (signed)	setf 0x7
setfh	(CY or Z) = 0	Higher (Greater than)	setf 0xB
setfnl	CY = 0	Not lower (Greater than or equal)	setf 0x9
setfl	CY = 1	Lower (Less than)	setf 0x1
setfnh	(CY or Z) = 1	Not higher (Less than or equal)	setf 0x3
setfe	Z = 1	Equal	setf 0x2
setfne	Z = 0	Not equal	setf 0xA
setfv	OV = 1	Overflow	setf 0x0
setfnv	OV = 0	No overflow	setf 0x8
setfn	S = 1	Negative	setf 0x4
setfp	S = 0	Positive	setf 0xC
setfc	CY = 1	Carry	setf 0x1

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
setfnc	CY = 0	No carry	setf 0x9
setfz	Z = 1	Zero	setf 0x2
setfnz	Z = 0	Not zero	setf 0xA
setft	always 1	Always 1	setf 0x5
setfsa	SAT = 1	Saturated	setf 0xD

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the setf instruction, the assembler outputs the following message and continues assembling using four low-order bits of a specified value.

W0550011: illegal operand (range error in immediate).

sasf

Sets the flag condition after a logical left shift.

[Syntax]

- sasf imm4, reg
- sasf*cnd* reg

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits

[Function]

- Syntax "sasf imm4, reg"
  Compares the flag condition indicated by the value of the lower 4 bits of the absolute expression specified by the first operand (see "Table 4-36. sasfcnd Instruction List") with the current flag condition. If a match is found, the contents of the register specified by the second operand are shifted logically 1 bit to the left and ORed with 1, and the result stored in the register specified by the second operand; otherwise, the contents of the register specified by the second operand are logically shifted 1 bit to the left and the result stored in the register specified by the second operand.
- Syntax "sasf*cnd* reg"
  Compares the flag condition indicated by string *cnd* with the current flag condition. If a match is found, the contents of the register specified by the second operand are shifted logically 1 bit to the left and ORed with 1, and the result stored in the register specified by the second operand; otherwise, the contents of the register specified by the second operand are shifted logically 1 bit to the left and the result stored in the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "sasf imm4, reg", the assembler generates one sasf machine instruction.
- If the instruction is executed in syntax "sasf*cnd* reg", the assembler generates the corresponding sasf instruction (see "Table 4-36. sasfcnd Instruction List") and expands it to syntax "sasf imm4, reg".

Table 4-36. sasf*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
sasfgt	((S xor OV) or Z) = 0	Greater than (signed)	sasf 0xF
sasfge	(S xor OV) = 0	Greater than or equal (signed)	sasf 0xE
sasflt	(S xor OV) = 1	Less than (signed)	sasf 0x6
sasfle	((S xor OV) or Z) = 1	Less than or equal (signed)	sasf 0x7
sasfh	(CY or Z) = 0	Higher (Greater than)	sasf 0xB
sasfnl	CY = 0	Not lower (Greater than or equal)	sasf 0x9
sasfl	CY = 1	Lower (Less than)	sasf 0x1
sasfnh	(CY or Z) = 1	Not higher (Less than or equal)	sasf 0x3
sasfe	Z = 1	Equal	sasf 0x2
sasfne	Z = 0	Not equal	sasf 0xA
sasfv	OV = 1	Overflow	sasf 0x0

Instruction	Flag Condition	Meaning of Flag Condition	Instruction Expansion
sasfnv	OV = 0	No overflow	sasf 0x8
sasfn	S = 1	Negative	sasf 0x4
sasfp	S = 0	Positive	sasf 0xC
sasfc	CY = 1	Carry	sasf 0x1
sasfnc	CY = 0	No carry	sasf 0x9
sasfz	Z = 1	Zero	sasf 0x2
sasfnz	Z = 0	Not zero	sasf 0xA
sasft	always 1	Always 1	sasf 0x5
sasfsa	SAT = 1	Saturated	sasf 0xD

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the sasf instruction, the assembler outputs the following message and continues assembling using four low-order bits of a specified value.

```
W0550011: illegal operand (range error in immediate).
```

4.7.8 Saturated operation instructions

This section describes the saturated operation instructions. Next table lists the instructions described in this section.

Table 4-37. Saturated Operation Instructions

Instruction	Meaning
satadd	Adds saturated
satsub	Subtracts saturated
satsubi	Subtracts saturated (immediate)
satsubr	Subtracts reverse saturated

satadd

Adda saturated.

[Syntax]

- satadd reg1, reg2
- satadd imm, reg2
- satadd reg1, reg2, reg3 [V850E2]

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "satadd reg1, reg2"
  Adds the value of the register specified by the first operand to the value of the register specified by the second operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1.
- Syntax "satadd imm, reg2"
  Adds the value of the absolute expression or relative expression specified by the first operand to the value of the register specified by the second operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1.
- Syntax "satadd reg1, reg2, reg3"
  Adds the value of the register specified by the first operand to the value of the register specified by the second operand, and stores the result in the register specified by the third operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the third operand. In both cases, the SAT flag is set to 1.

[Description]

- If the instruction is executed in syntax "satadd reg1, reg2" or "satadd reg1, reg2, reg3", the assembler generates one satadd machine instruction.
- If the following is specified for imm in syntax "satadd imm, reg2", the assembler generates one satadd machine instruction[Note].

(a) Absolute expression having a value in the range of -16 to +15

`satadd imm5, reg`	`satadd imm5, reg`

Note The satadd machine instruction takes a register or immediate value in the range of -16 to +15 (0xFFFFFFF0 to 0xF) as the first operand.

- If the following is specified for imm in syntax "satadd imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`satadd imm16, reg`	`movea imm16, r0, r1` `satadd r1, reg`

(b) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`satadd imm, reg`	`movhi HIGHW(imm), r0, r1` `satadd r1, reg`

Else

`satadd imm, reg`	`mov imm, r1` `satadd r1, reg`

(c) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`satadd $label, reg`	`movea $label, r0, r1` `satadd r1, reg`

(d) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`satadd #label, reg`	`mov #label, r1` `satadd r1, reg`
`satadd label, reg`	`mov label, r1` `satadd r1, reg`
`satadd $label, reg`	`mov $label, r1` `satadd r1, reg`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	1 if OV = 1, - if not

[Caution]

- If the instruction is executed in syntax "satadd reg1, reg2" or "satadd imm, reg2", if r0 is specified as the second operand, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

satsub

Subtracts saturated.

[Syntax]

- satsub reg1, reg2
- satsub imm, reg2
- satsub reg1, reg2, reg3 [V850E2]

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "satsub reg1, reg2"
  Subtracts the value of the register specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the third operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1
- Syntax "satsub imm, reg2"
  Subtracts the value of the absolute expression or relative expression specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1.
- Syntax "satsub reg1, reg2, reg3"
  Subtracts the value of the register specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the third operand. In both cases, the SAT flag is set to 1.

[Description]

- If the instruction is executed in syntax "satsub reg1, reg2" or "satsub reg1, reg2, reg3", the assembler generates one satsub machine instruction.
- If the instruction is executed in syntax "satsub imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`satsub 0, reg`	`satsub r0, reg`

(b) Absolute expression having a value in the range of -32,768 to +32,767

`satsub imm16, reg`	`satsubi imm16, reg, reg`

(c) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`satsub imm, reg`	`movhi HIGHW(imm), r0, r1` `satsub r1, reg`

Else

`satsub imm, reg`	`mov imm, r1` `satsub r1, reg`

(d) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`satsub $label, reg`	`satsubi $label, reg, reg`

(e) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`satsub #label, reg`	`mov #label, r1` `satsub r1, reg`
`satsub label, reg`	`mov label, r1` `satsub r1, reg`
`satsub $label, reg`	`mov $label, r1` `satsub r1, reg`

Note The satsub machine instruction does not take an immediate value as an operand.

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	1 if OV = 1, - if not

[Caution]

- If the instruction is executed in syntax "satsub reg1, reg2" or "satsub imm, reg2", if r0 is specified as the second operand, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

satsubi

Subtracts saturated (immediate).

[Syntax]

- satsubi imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

Subtracts the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied specified by the first operand from the value of the register specified by the second operand, and stores the result in the register specified by the third operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the third operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the third operand. In both cases, the SAT flag is set to 1.

[Description]

- If the following is specified for imm, the assembler generates one satsubi machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`satsubi imm16, reg1, reg2`	`satsubi imm16, reg1, reg2`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`satsubi $label, reg1, reg2`	`satsubi $label, reg1, reg2`

(c) Relative expression having !label or %label

`satsubi !label, reg1, reg2`	`satsubi !label, reg1, reg2`
`satsubi %label, reg1, reg2`	`satsubi %label, reg1, reg2`

(d) Expression with HIGHW, LOWW, or HIGHW1

`satsubi imm16, reg1, reg2`	`satsubi imm16, reg1, reg2`

Note The satsubi machine instruction takes an immediate value, in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF), as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`satsubi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `satsubr reg1, reg2`

If all the lower 16 bits of the value of imm are 0

`satsubi imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `satsubr reg1, r1`

Else

`satsubi imm, reg1, reg2`	`mov imm, reg2` `satsubr reg1, reg2`

Other than above and when reg2 is r0

`satsubi imm, reg1, r0`	`mov imm, r1` `satsubr reg1, r1`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

If reg2 is r0

`satsubi #label, reg1, r0`	`movhi #label, r1` `satsubr reg1, r1`
`satsubi label, reg1, r0`	`mov label, r1` `satsubr reg1, r1`
`satsubi $label, reg1, r0`	`mov $label, r1` `satsubr reg1, r1`

Else

`satsubi #label, reg1, reg2`	`movhi #label, reg2` `satsubr reg1, reg2`
`satsubi label, reg1, reg2`	`mov label, reg2` `satsubr reg1, reg2`
`satsubi $label, reg1, reg2`	`mov $label, reg2` `satsubr reg1, reg2`

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	1 if OV = 1, - if not

[Caution]

- If r0 is specified by the second operand, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

satsubr

Subtracts reverse saturated.

[Syntax]

- satsubr reg1, reg2
- satsubr imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "satsubr reg1, reg2"
  Subtracts the value of the register specified by the second operand from the value of the register specified by the first operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1.
- Syntax "satsubr imm, reg2"
  Subtracts the value of the register specified by the second operand from the value of the absolute expression or relative expression specified by the first operand, and stores the result in the register specified by the second operand. If the result exceeds the maximum positive value of 0x7FFFFFFF, however, 0x7FFFFFFF is stored in the register specified by the second operand. Likewise, if the result exceeds the maximum negative value of 0x80000000, 0x80000000 is stored in the register specified by the second operand. In both cases, the SAT flag is set to 1.

[Description]

- If the instruction is executed in syntax "satsubr reg1, reg2", the assembler generates one satsubr machine instruction.
- If the instruction is executed in syntax "satsubr imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`satsubr 0, reg`	`satsubr r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`satsubr imm5, reg`	`mov imm5, r1` `satsubr r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`satsubr imm16, reg`	`movea imm16, r0, r1` `satsubr r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`satsubr imm, reg`	`movhi HIGHW(imm), r0, r1` `satsubr r1, reg`

Else

`satsubr imm, reg`	`mov imm, r1` `satsubr r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`satsubr $label, reg`	`movea $label, r0, r1` `satsubr r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`satsubr #label, reg`	`mov #label, r1` `satsubr r1, reg`
`satsubr label, reg`	`mov label, r1` `satsubr r1, reg`
`satsubr $label, reg`	`mov $label, r1` `satsubr r1, reg`

Note The satsubr machine instruction does not take an immediate value as an operand.

[Flag]

CY	1 if a borrow occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	1 if OV = 1, - if not

[Caution]

- If r0 is specified by the second operand, the assembler outputs the following message and stops assembling.

E0550240: Illegal operand (cannot use r0 as destination in V850E mode).

4.7.9 Logical instructions

This section describes the logical instructions. Next table lists the instructions described in this section.

Table 4-38. Logical Instructions

Instruction	Meanings
or	Logical sum
ori	Logical sum (immediate)
xor	Exclusive OR
xori	Exclusive OR (immediate)
and	Logical product
andi	Logical product (immediate)
not	Logical negation (takes 1's complement)
shr	Logical right shift
sar	Arithmetic right shift
shl	Logical left shift
sxb	Sign extension of byte data
sxh	Sign extension of 2-byte data
zxb	Zero extension of byte data
zxh	Zero extension of 2-byte data
bsh	Byte swap of half-word data
bsw	Byte swap of word data
hsh	Half-word swap of half-word data [V850E2]
hsw	Half-word swap of word data
tst	Test
sch0l	Bit (0) search from MSB side [V850E2]
sch0r	Bit (0) search from LSB side [V850E2]
sch1l	Bit (1) search from MSB side [V850E2]
sch1r	Bit (1) search from LSB side [V850E2]

or

Logical sum.

[Syntax]

- or reg1, reg2
- or imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "or reg1, reg2"
  ORs the value of the register specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operand.
- Syntax "or imm, reg2"
  ORs the value of the absolute expression or relative expression specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operand.

[Description]

- When this instruction is executed in syntax "or reg1, reg2", the assembler generates one or machine instruction.
- When this instruction is executed in syntax "or imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`or 0, reg`	`or r0, reg`

(b) Absolute expression having a value in the range of 1 to 65,535

`or imm16, reg`	`ori imm16, reg, reg`

(c) Absolute expression having a value in the range of -16 to -1

`or imm5, reg`	`mov imm5, r1` `or r1, reg`

(d) Absolute expression having a value in the range of -32,768 to -17

`or imm16, reg`	`movea imm16, r0, r1` `or r1, reg`

(e) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`or imm, reg`	`movhi HIGHW(imm), r0, r1` `or r1, reg`

Else

`or imm, reg`	`mov imm, r1` `or r1, reg`

(f) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`or $label, reg`	`movea $label, r0, r1` `or r1, reg`

(g) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`or #label, reg`	`mov #label, r1` `or r1, reg`
`or label, reg`	`mov label, r1` `or r1, reg`
`or $label, reg`	`mov $label, r1` `or r1, reg`

Note The or machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

ori

Logical sum (immediate).

[Syntax]

- ori imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

ORs the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the third operand.

[Description]

- If the following is specified for imm, the assembler generates one ori machine instruction[Note].

(a) Absolute expression having a value in the range of 0 to 65,535

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

(b) Relative expression having !label or %label

`ori !label, reg1, reg2`	`ori !label, reg1, reg2`
`ori %label, reg1, reg2`	`ori %label, reg1, reg2`

(c) Expression with HIGHW, LOWW, or HIGHW1

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

Note The ori machine instruction takes an immediate value of 0 to 65,535 (0 to 0xFFFF) as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression having a value in the range of -16 to -1

If reg2 is r0

`ori imm5, reg1, r0`	`mov imm5, r1` `or reg1, r1`

Else

`ori imm5, reg1, reg2`	`mov imm5, reg2` `or reg1, reg2`

(b) Absolute expression having a value in the range of -32,768 to -17

If reg2 is r0

`ori imm16, reg1, r0`	`movea imm16, r0, r1` `or reg1, r1`

Else

`ori imm16, reg1, reg2`	`movea imm16, r0, reg2` `or reg1, reg2`

(c) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`ori imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `or reg1, reg2`

If all the lower 16 bits of the value of imm are 0

`ori imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `or reg1, r1`

Else

`ori imm, reg1, reg2`	`mov imm, reg2` `or reg1, reg2`

Other than above and when reg2 is r0

`ori imm, reg1, r0`	`mov imm, r1` `or reg1, r1`

(d) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

If reg2 is r0

`ori $label, reg1, r0`	`movea $label, r0, r1` `or reg1, r1`

Else

`ori $label, reg1, reg2`	`movea $label, r0, reg2` `or reg1, reg2`

(e) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

If reg2 is r0

`ori #label, reg1, r0`	`mov #label, r1` `or reg1, r1`
`ori label, reg1, r0`	`mov label, r1` `or reg1, r1`
`ori $label, reg1, r0`	`mov $label, r1` `or reg1, r1`

Else

`ori #label, reg1, reg2`	`mov #label, reg2` `or reg1, reg2`
`ori label, reg1, reg2`	`mov label, reg2` `or reg1, reg2`
`ori $label, reg1, reg2`	`mov $label, reg2` `or reg1, reg2`

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

xor

Exclusive OR.

[Syntax]

- xor reg1, reg2
- xor imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "xor reg1, reg2"
  Exclusive-ORs the value of the register specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operand.
- Syntax "xor imm, reg2"
  Exclusive-ORs the value of the absolute expression or relative expression specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operan.

[Description]

- When this instruction is executed in syntax "xor reg1, reg2", the assembler generates one xor machine instruction.
- When this instruction is executed in syntax "xor imm, reg2", the assembler executes instruction expansion to generate two or more machine instructions[Note].

(a) 0

`xor 0, reg`	`xor r0, reg`

(b) Absolute expression having a value in the range of 1 to 65,535

`xor imm16, reg`	`xori imm16, reg, reg`

(c) Absolute expression having a value in the range of -16 to -1

`xor imm5, reg`	`mov imm5, r1` `xor r1, reg`

(d) Absolute expression having a value in the range of -32,768 to -17

`xor imm16, reg`	`movea imm16, r0, r1` `xor r1, reg`

(e) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`xor imm, reg`	`movhi HIGHW(imm), r0, r1` `xor r1, reg`

Else

`xor imm, reg`	`mov imm, r1` `xor r1, reg`

(f) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`xor $label, reg`	`movea $label, r0, r1` `xor r1, reg`

(g) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`xor #label, reg`	`mov #label, r1` `xor r1, reg`
`xor label, reg`	`mov label, r1` `xor r1, reg`
`xor $label, reg`	`mov $label, r1` `xor r1, reg`

Note The xor machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

xori

Exclusive OR (Immediate).

[Syntax]

- xori imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

Exclusive-ORs the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the third operand.

[Description]

- If the following is specified for imm, the assembler generates one xori machine instruction[Note].

(a) Absolute expression having a value in the range of 0 to 65,535

`xori imm16, reg1, reg2`	`xori imm16, reg1, reg2`

(b) Relative expression having !label or %label

`xori !label, reg1, reg2`	`xori !label, reg1, reg2`
`xori %label, reg1, reg2`	`xori %label, reg1, reg2`

(c) Expression with HIGHW, LOWW, or HIGHW1

`xori imm16, reg1, reg2`	`xori imm16, reg1, reg2`

Note The xori machine instruction takes an immediate value of 0 to 65,535 (0 to 0xFFFF) as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression having a value in the range of -16 to -1

If reg2 is r0

`xori imm5, reg1, r0`	`mov imm5, r1` `xor reg1, r1`

Else

`xori imm5, reg1, reg2`	`mov imm5, reg2` `xor reg1, reg2`

(b) Absolute expression having a value in the range of -32,768 to -17

If reg2 is r0

`xori imm16, reg1, r0`	`movea imm16, r0, r1` `xor reg1, r1`

Else

`xori imm16, reg1, reg2`	`movea imm16, r0, reg2` `xor reg1, reg2`

(c) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`xori imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `xor reg1, reg2`

If all the lower 16 bits of the value of imm are 0

`xori imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `xor reg1, r1`

Else

`xori imm, reg1, reg2`	`mov imm, reg2` `xor reg1, reg2`

Other than above and when reg2 is r0

`xori imm, reg1, r0`	`mov imm, r1` `xor reg1, r1`

(d) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

If reg2 is r0

`xori $label, reg1, r0`	`movea $label, r0, r1` `xor reg1, r1`

Else

`xori $label, reg1, reg2`	`movea $label, r0, reg2` `xor reg1, reg2`

(e) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

If reg2 is r0

`xori #label, reg1, r0`	`mov #label, r1` `xor reg1, r1`
`xori label, reg1, r0`	`mov label, r1` `xor reg1, r1`
`xori $label, reg1, r0`	`mov $label, r1` `xor reg1, r1`

Else

`xori #label, reg1, reg2`	`mov #label, reg2` `xor reg1, reg2`
`xori label, reg1, reg2`	`mov label, reg2` `xor reg1, reg2`
`xori $label, reg1, reg2`	`mov $label, reg2` `xor reg1, reg2`

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

and

Logical product.

[Syntax]

- and reg1, reg2
- and imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "and reg1, reg2"
  ANDs the value of the register specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operand.
- Syntax "and imm, reg2"
  ANDs the value of the absolute expression or relative expression specified by the first operand with the value of the register specified by the second operand, and stores the result in the register specified by the second operand.

[Description]

- When this instruction is executed in syntax "and reg1, reg2", the assembler generates one and machine instruction.
- When this instruction is executed in syntax "and imm, reg2", the assembler executes instruction expansion to generate one or more machine instruction[Note].

(a) 0

`and 0, reg`	`and r0, reg`

(b) Absolute expression having a value in the range of 1 to 65,535

`and imm16, reg`	`andi imm16, reg, reg`

(c) Absolute expression having a value in the range of -16 to -1

`and imm5, reg`	`mov imm5, r1` `and r1, reg`

(d) Absolute expression having a value in the range of -32,768 to -17

`and imm16, reg`	`movea imm16, r0, r1` `and r1, reg`

(e) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`and imm, reg`	`movhi HIGHW(imm), r0, r1` `and r1, reg`

Else

`and imm, reg`	`mov imm, r1` `and r1, reg`

(f) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`and $label, reg`	`movea $label, r0, r1` `and r1, reg`

(g) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`and #label, reg`	`mov #label, r1` `and r1, reg`
`and label, reg`	`mov label, r1` `and r1, reg`
`and $label, reg`	`mov $label, r1` `and r1, reg`

Note The and machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

andi

Logical product (immediate).

[Syntax]

- andi imm, reg1, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

[Function]

ANDs the value of the absolute expression, relative expression, or expression with HIGHW, LOWW, or HIGHW1 applied specified by the first operand with the value of the register specified by the second operand, and stores the result into the register specified by the third operand.

[Description]

- If the following is specified as imm, the assembler generates one andi machine instruction[Note].

(a) Absolute expression having a value in the range of 0 to 65,535

`andi imm16, reg1, reg2`	`andi imm16, reg1, reg2`

(b) Relative expression having !label or %label

`andi !label, reg1, reg2`	`andi !label, reg1, reg2`
`andi %label, reg1, reg2`	`andi %label, reg1, reg2`

(c) Expression with HIGHW, LOWW, or HIGHW1

`andi imm16, reg1, reg2`	`andi imm16, reg1, reg2`

Note The andi machine instruction takes an immediate value of 0 to 65,535 (0 to 0xFFFF) as the first operand.

- If the following is specified for imm, the assembler executes instruction expansion to generate one or more machine instructions.

(a) Absolute expression having a value in the range of -16 to -1

If reg2 is r0

`andi imm5, reg1, r0`	`mov imm5, r1` `and reg1, r1`

Else

`andi imm5, reg1, reg2`	`mov imm5, reg2` `and reg1, reg2`

(b) Absolute expression having a value in the range of -32,768 to -17

If reg2 is r0

`andi imm16, reg1, r0`	`movea imm16, r0, r1` `and reg1, r1`

Else

`andi imm16, reg1, reg2`	`movea imm16, r0, reg2` `and reg1, reg2`

(c) Absolute expression exceeding the above ranges

If all the lower 16 bits of the value of imm are 0

`andi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `and reg1, reg2`

If all the lower 16 bits of the value of imm are 0

`andi imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `and reg1, r1`

Else

`andi imm, reg1, reg2`	`mov imm, reg2` `and reg1, reg2`

Other than above and when reg2 is r0

`andi imm, reg1, reg2`	`mov imm, r1` `and reg1, r1`

(d) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

If reg2 is r0

`andi $label, reg1, r0`	`movea $label, r0, r1` `and reg1, r1`

Else

`andi $label, reg1, reg2`	`movea $label, r0, reg2` `and reg1, reg2`

(e) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

If reg2 is r0

`andi #label, reg1, r0`	`mov #label, r1` `and reg1, r1`
`andi label, reg1, r0`	`mov label, r1` `and reg1, r1`
`andi $label, reg1, r0`	`mov $label, r1` `and reg1, r1`

Else

`andi #label, reg1, reg2`	`mov #label, reg2` `and reg1, reg2`
`andi label, reg1, reg2`	`mov label, reg2` `and reg1, reg2`
`andi $label, reg1, reg2`	`mov $label, reg2` `and reg1, reg2`

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

not

Logical negation (takes 1's complement).

[Syntax]

- not reg1, reg2
- not imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "not reg1, reg2"
  NOTs (1's complement) the value of the register specified by the first operand, and stores the result in the register specified by the second operand.
- Syntax "not imm, reg2"
  NOTs (1's complement) the value of the absolute expression or relative expression specified by the first operand, and stores the result in the register specified by the second operand.

[Description]

- When this instruction is executed in syntax "not reg1, reg2", the assembler generates one not machine instruction.
- When this instruction is executed in syntax "not imm, reg2", the assembler executes instruction expansion to generate one or more machine instructions[Note].

(a) 0

`not 0, reg`	`not r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`not imm5, reg`	`mov imm5, r1` `not r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`not imm16, reg`	`movea imm16, r0, r1` `not r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`not imm, reg`	`movhi HIGHW(imm), r0, r1` `not r1, reg`

Else

`not imm, reg`	`mov imm, r1` `not r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`not $label, reg`	`movea $label, r0, r1` `not r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`not #label, reg`	`mov #label, r1` `not r1, reg`
`not label, reg`	`mov label, r1` `not r1, reg`
`not $label, reg`	`mov $label, r1` `not r1, reg`

Note The not machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

shr

Logical right shift.

[Syntax]

- shr reg1, reg2
- shr imm5, reg2
- shr reg1, reg2, reg3 [V850E2]

The following can be specified for imm5:

- Absolute expression having a value of up to 5 bits

[Function]

- Syntax "shr reg1, reg2"
  Logically shifts to the right the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "shr imm5, reg2"
  Logically shifts to the right the value of the register specified by the second operand by the number of bits specified by the value of the absolute expression specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "shr reg1, reg2, reg3"
  Logically shifts to the right the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the third operand.

[Description]

The assembler generates one shr machine instruction.

[Flag]

CY	1 if the value of the bit shifted out last is 1, 0 if not (0 if the specified number of bits is 0)
OV	0
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 31 is specified as imm5 in syntax "shr imm5, reg2", the assembler outputs the following message, and continues assembling by using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate).
```

Note The shr machine instruction takes an immediate value of 0 to 31 (0x0 to 0x1F) as the first operand.

sar

Arithmetic right shift.

[Syntax]

- sar reg1, reg2
- sar imm5, reg2
- sar reg1, reg2, reg3 [V850E2]

The following can be specified for imm5:

- Absolute expression having a value of up to 5 bits

[Function]

- Syntax "sar reg1, reg2"
  Arithmetically shifts to the right the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "sar imm5, reg2"
  Arithmetically shifts to the right the value of the register specified by the second operand by the number of bits specified by the value of the absolute expression specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "sar reg1, reg2, reg3"
  Arithmetically shifts to the right the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the third operand.

[Description]

The assembler generates one sar machine instruction.

[Flag]

CY	1 if the value of the bit shifted out last is 1, 0 if not (0 if the specified number of bits is 0)
OV	0
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 31 is specified for imm5 in syntax "sar imm5, reg2", the assembler outputs the following message, and continues assembling using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate).
```

Note The sar machine instruction takes an immediate value of 0 to 31 (0x0 to 0x1F) as the first operand.

shl

Logical left shift.

[Syntax]

- shl reg1, reg2
- shl imm5, reg2
- shl reg1, reg2, reg3 [V850E2]

The following can be specified for imm5:

- Absolute expression having a value of up to 5 bits

[Function]

- Syntax "shl reg1, reg2"
  Logically shifts to the left the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "shl imm5, reg2"
  Logically shifts to the left the value of the register specified by the second operand by the number of bits specified by the value of the absolute expression specified by the first operand, then stores the result in the register specified by the second operand.
- Syntax "shl reg1, reg2, reg3"
  Logically shifts to the left the value of the register specified by the second operand by the number of bits indicated by the lower 5 bits of the register value specified by the first operand, then stores the result in the register specified by the third operand.

[Description]

The assembler generates one shl machine instruction.

[Flag]

CY	1 if the value of the bit shifted out last is 1, 0 if not (0 if the specified number of bits is 0)
OV	0
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 31 is specified for imm5 in syntax "shl imm5, reg2", the assembler outputs the following message, and continues assembling by using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate).
```

Note The shl machine instruction takes an immediate value of 0 to 31 (0x0 to 0x1F) as the first operand.

sxb

Sign extension of byte data.

[Syntax]

- sxb reg

[Function]

Sign-extends the data of the lowermost byte of the register specified by the first operand to word length.

[Description]

The assembler generates one sxb machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

sxh

Sign extension of 2-byte data.

[Syntax]

- sxh reg

[Function]

Sign-extends the data of the lower 2 bytes of the register specified by the first operand to word length.

[Description]

The assembler generates one sxh machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

zxb

Zero extension of byte data.

[Syntax]

- zxb reg

[Function]

Zero-extends the data of the lowermost byte of the register specified by the first operand to word length.

[Description]

The assembler generates one zxb machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

zxh

Zero extension of 2-byte data

[Syntax]

- zxh reg

[Function]

Zero-extends the data of the lower 2 bytes of the register specified by the first operand to word length.

[Description]

The assembler generates one zxh machine instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

bsh

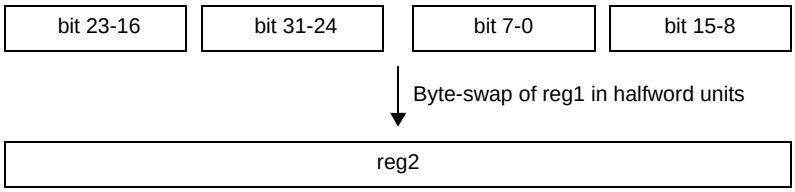
Byte swap of half-word data.

[Syntax]

- bsh reg1, reg2

[Function]

Byte-swaps the register value specified by the first operand in halfword units and stores the result in the register specified by the second operand.



[Description]

The assembler generates one bsh machine instruction.

[Flag]

CY	1 if either or both of the bytes in the lower halfword of the register is 0, 0 if not
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the lower half-word data of the result is 0, 0 if not
SAT	---

bsw

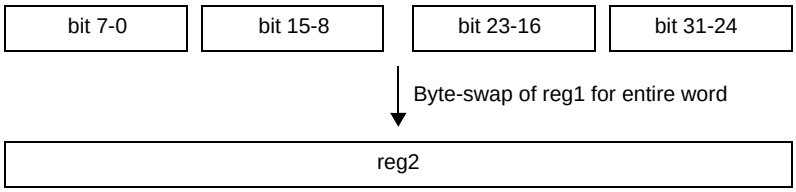
Byte swap of word data.

[Syntax]

- bsw reg1, reg2

[Function]

Byte-swaps the register value specified by the first operand and stores the result in the register specified by the second operand.



[Description]

The assembler generates one bsw machine instruction.

[Flag]

CY	1 if one or more bytes of the word in the register is 0, 0 if not
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the word data of the result is 1, 0 if not
SAT	---

hsh

Half-word swap of half-word data. [V850E2]

[Syntax]

- hsh reg2, reg3

[Function]

Stores the register value specified by the first operand in the register specified by the second operand, and stores the flag assessment result in the PSW register.

[Description]

The assembler generates one hsh machine instruction.

[Flag]

CY	1 if the lower half-word data of the result is 0, 0 if not
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the lower half-word data of the result is 0, 0 if not
SAT	---

hsw

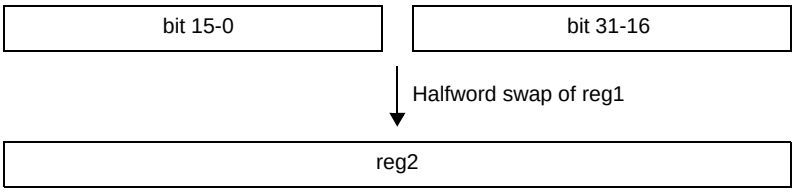
Half-word swap of word data.

[Syntax]

- hsw reg1, reg2

[Function]

Halfword-swaps the register value specified by the first operand and stores the result in the register specified by the second operand.



[Description]

The assembler generates one hsw machine instruction.

[Flag]

CY	1 if one or more halfwords in the word of the register is 0, 0 if not
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the word data of the result is 1, 0 if not
SAT	---

tst

Test.

[Syntax]

- tst reg1, reg2
- tst imm, reg2

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits
- Relative expression

[Function]

- Syntax "tst reg1, reg2"
  ANDs the value of the register specified by the second operand with the value of the register specified by the first operand, and sets only the flags without storing the result.
- Syntax "tst imm, reg2"
  ANDs the value of the register specified by the second operand with the value of the absolute expression or relative expression specified by the first operand, and sets only the flags without storing the result.

[Description]

- When this instruction is executed in syntax "tst reg1, reg2", the assembler generates one tst machine instruction.
- When this instruction is executed in syntax "tst imm, reg2", the assembler executes instruction expansion to generate two or more machine instructions[Note].

(a) 0

`tst 0, reg`	`tst r0, reg`

(b) Absolute expression having a value of other than 0 within the range of -16 to +15

`tst imm5, reg`	`mov imm5, r1` `tst r1, reg`

(c) Absolute expression exceeding the range of -16 to +15, but within the range of -32,768 to +32,767

`tst imm16, reg`	`movea imm16, r0, r1` `tst r1, reg`

(d) Absolute expression having a value exceeding the range of -32,768 to +32,767

If all the lower 16 bits of the value of imm are 0

`tst imm, reg`	`movhi HIGHW(imm), r0, r1` `tst r1, reg`

Else

`tst imm, reg`	`mov imm, r1` `tst r1, reg`

(e) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`tst $label, reg`	`movea $label, r0, r1` `tst r1, reg`

(f) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`tst #label, reg`	`mov #label, r1` `tst r1, reg`
`tst label, reg`	`mov label, r1` `tst r1, reg`
`tst $label, reg`	`mov $label, r1` `tst r1, reg`

Note The tst machine instruction does not take an immediate value as an operand.

[Flag]

CY	---
OV	0
S	1 if the word data MSB of the result is 1, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

sch0l

Bit (0) search from MSB side (search zero from left). [V850E2]

[Syntax]

- sch0l reg1, reg2

[Function]

Searches the word data of the register specified by the first operand, from the left (MSB side), and stores the position of the first bit (0) found in the register specified by the second operand in hexadecimal. (For example, if bit 31 of the register specified by the first operand is 0, 01H is stored in the register specified by the second operand.)

If no bit (0) is found, 0 is written into the register specified by the second operand, and the Z flag is simultaneously set (1). If a bit (0) is found at the end, the CY flag is set (1).

[Description]

The assembler generates one sch0l machine instruction.

[Flag]

CY	1 if a bit (0) is found at the end, 0 if not
OV	0
S	0
Z	1 if a bit (0) is not found, 0 if not
SAT	---

sch0r

Bit (0) search from LSB side (search zero from right). [V850E2]

[Syntax]

- sch0r reg1, reg2

[Function]

Searches the word data of the register specified by the first operand, from the right (LSB side), and stores the position of the first bit (0) found in the register specified by the second operand in hexadecimal. (For example, if bit 0 of the register specified by the first operand is 0, 01H is stored in the register specified by the second operand.)

If no bit (0) is found, 0 is written into the register specified by the second operand, and the Z flag is simultaneously set (1). If a bit (0) is found at the end, the CY flag is set (1).

[Description]

The assembler generates one sch0r machine instruction.

[Flag]

CY	1 if a bit (0) is found at the end, 0 if not
OV	0
S	0
Z	1 if a bit (0) is not found, 0 if not
SAT	---

sch1l

Bit (1) search from MSB side (search one from left). [V850E2]

[Syntax]

- sch1l reg1, reg2

[Function]

Searches the word data of the register specified by the first operand, from the left (MSB side), and stores the position of the first bit (1) found in the register specified by the second operand in hexadecimal. (For example, if bit 31 of the register specified by the first operand is 1, 01H is stored in the register specified by the second operand.)

If no bit (1) is found, 0 is written into the register specified by the second operand, and the Z flag is simultaneously set (1). If a bit (0) is found at the end, the CY flag is set (1).

[Description]

The assembler generates one sch1l machine instruction.

[Flag]

CY	1 if a bit (1) is found at the end, 0 if not
OV	0
S	0
Z	1 if a bit (1) is not found, 0 if not
SAT	---

sch1r

Bit (1) search from LSB side (search zero from right). [V850E2]

[Syntax]

- sch1r reg2, reg3

[Function]

Searches the word data of the register specified by the first operand, from the right (LSB side), and stores the position of the first bit (1) found in the register specified by the second operand in hexadecimal. (For example, if bit 0 of the register specified by the first operand is 1, 01H is stored in the register specified by the second operand.)

If no bit (1) is found, 0 is written into the register specified by the second operand, and the Z flag is simultaneously set (1). If a bit (1) is found at the end, the CY flag is set (1).

[Description]

The assembler generates one sch1r machine instruction.

[Flag]

CY	1 if a bit (1) is found at the end, 0 if not
OV	0
S	0
Z	1 if a bit (1) is not found, 0 if not
SAT	---

4.7.10 Branch instructions

This section describes the branch instructions. Next table lists the instructions described in this section.

Table 4-39. Branch Instructions

Instruction	Meanings
jmp	Unconditional branch
jmp32	Unconditional branch [V850E2]
jr	Unconditional branch (PC relative)
jr22	Unconditional branch (PC relative) [V850E2]
jr32	Unconditional branch (PC relative) [V850E2]
jcnd	Conditional branch
jarl	Jump and register link
jarl22	Jump and register link [V850E2]
jarl32	Jump and register link [V850E2]

jmp

Unconditional branch.

[Syntax]

- jmp [reg]
- jmp addr
- jmp disp32[reg] [V850E2]

The following can be specified for addr:

- Relative expression having the absolute address reference of a label

The following can be specified for disp32:

- Absolute expression having a value of up to 32 bits

[Function]

- Syntax "jmp [reg]"
  Transfers control to the address indicated by the value of the register specified by the operand.
- Syntax "jmp disp32[reg]"
  Transfers control to the address attained by adding the displacement specified by the operand and the register content.
- Syntax "jmp addr"
  Transfers control to the address indicated by the value of the relative expression specified by the operand.

[Description]

- When this instruction is executed in syntax "jmp [reg]", the assembler generates one jmp machine instruction.
- When this instruction is executed in syntax "jmp addr", the assembler executes instruction expansion and generates two or more machine instruction.

`jmp #label`	`mov #label, r1` `jmp [r1]`

- If the instruction is executed in syntax "jmp addr", when the V850E2 operate, the assembler generates one jmp machine instruction (6-byte long instruction).
- When this instruction is executed in syntax "jmp disp32[reg]", the assembler generates one jmp (6-byte long instruction) machine instructions.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an expression other than a relative expression having the absolute address reference of a label is specified as addr in syntax "jmp addr", the assembler outputs the following message and stops assembling.

E0550224: Illegal operand (label reference for jmp must be string).

jmp32

Unconditional branch. [V850E2]

[Syntax]

- jmp32 disp32[reg]
- jmp32 addr

The following can be specified for addr:

- Relative expression having the absolute address reference of a label

The following can be specified for disp32:

- Absolute expression having a value of up to 32 bits

[Function]

- Syntax "jmp32 disp32[reg]"
  Transfers control to the address attained by adding the displacement specified by the operand and the register content.
- Syntax "jmp32 addr"
  Transfers control to the address indicated by the value of the relative expression specified by the operand.

[Description]

The assembler generates one jmp machine instruction (6-byte long instruction).

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an expression other than a relative expression having the absolute address reference of a label is specified as addr in syntax "jmp32 addr", the assembler outputs the following message and stops assembling.

```
E0550224: Illegal operand (label reference for jmp must be string).
```

jr

Unconditional branch (PC relative).

[Syntax]

- jr disp22
- jr disp32 [V850E2]

The following can be specified for disp22:
- Absolute expression having a value of up to 22 bits
- Relative expression having a PC offset reference of label

The following can be specified for disp32:
- Absolute expression having a value of up to 32 bits
- Relative expression having a PC offset reference of label

[Function]

- Syntax "jr disp22"
  Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand.
- Syntax "jr disp32"
  Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand.

[Description]

- If the instruction is executed in syntax "jr disp22", the assembler generates one jr machine instruction[Note] if any of the following expressions are specified for disp22.

  (a) Absolute expression having a value in the range of -2,097,152 to +2,097,151

  (b) Relative expression that has a PC offset reference of label having a definition in the same section of the same file as this instruction, and having a value in the range of -2,097,152 to +2,097,151

  (c) Relative expression having a PC offset reference of a label with no definition in the same file or section as this instruction

  Note The jr machine instruction takes an immediate value in the range of -2,097,152 to +2,097,151 (0xFE00000 to 0x1FFFFF) as the displacement.

- If the instruction is executed in syntax "jr disp32", the assembler generates one jr machine instruction (6-byte long instruction).

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of -2,097,152 to +2,097,151, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction, and having a value exceeding the range of -2,097,152 to +2,097,151, is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550230: illegal operand (range error in displacement)
```

- If an absolute expression having an odd-numbered value or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction, and having an odd-numbered value, is specified as disp22/disp32, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

- When the assembler option -Xfar_jump is not specified, and an absolute expression outside of the range -2,097,152 to +2,097,151 or a relative expression outside of the range -2,097,152 to +2,097,151, having a label PC offset reference with a definition in the same file and same section as this instruction, is specified as disp32, the following message is output and assembly is stopped.

```
E0550230: illegal operand (range error in displacement)
```

jr22

Unconditional branch (PC relative). [V850E2]

[Syntax]

- jr22 disp22

The following can be specified for disp22:

- Absolute expression having a value of up to 22 bits
- Relative expression having a PC offset reference of label

[Function]

Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the operand.

[Description]

- If the following is specified for disp22, the assembler generates one jr machine instruction[Note].

(a) Absolute value in the range of -2,097,152 to +2,097,151

(b) Relative expression that has a PC offset reference of label having a definition in the same section and the same file as this instruction, and which has a value in the range of -2,097,152 to +2,097,151

(c) Relative expression having a PC offset reference of a label having no definition in the same file or section as this instruction

Note The jr machine instruction takes an immediate value in the range of -2,097,152 to +2,097,151 (0xFE00000 to 0x1FFFFF) as the displacement.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression that exceeds the range of -2,097,152 to +2,097,151, or a relative expression having a PC offset reference of label with a definition in the same section and the same file as this instruction and having a value that falls outside the range of -2,097,152 to +2,097,151 is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550230: illegal operand (range error in displacement)
```

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction and having an odd-numbered value, is specified as disp22, the assembler outputs the following message and stops assembling.

E0550226: illegal operand (must be even displacement)

jr32

Unconditional branch (PC relative). [V850E2]

[Syntax]

- jr32 disp32

The following can be specified for disp32:

- Absolute expression having a value of up to 32 bits
- Relative expression having a PC offset reference of label

[Function]

Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand.

[Description]

The assembler generates one jr machine instruction (6-byte long instruction).

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction and having an odd-numbered value, is specified as disp32, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

jcnd

Conditional branch.

[Syntax]

- j*cnd* disp22

The following can be specified for disp22:

- Absolute expression having a value of up to 22 bits
- Relative expression having a PC offset reference of label

[Function]

Compares the flag condition indicated by string *cnd* (see "Table 4-40. jcnd Instruction List") with the current flag condition. If they are found to be the same, transfers control to the address obtained by adding the value of the absolute expression or relative expression specified by the operand to the current value of the program counter (PC)[Note].

Note For a j*cnd* instruction other than jbr, the mnemonic "b*cnd*" can be used, and the "br" machine-language instruction can be used for the jbr instruction (there is no functional difference).

Table 4-40. j*cnd* Instruction List

Instruction	Flag Condition	Meaning of Flag Condition
jgt	((S xor OV) or Z) = 0	Greater than (signed)
jge	(S xor OV) = 0	Greater than or equal (signed)
jlt	(S xor OV) = 1	Less than (signed)
jle	((S xor OV) or Z) = 1	Less than or equal (signed)
jh	(CY or Z) = 0	Higher (Greater than)
jnl	CY = 0	Not lower (Greater than or equal)
jl	CY = 1	Lower (Less than)
jnh	(CY or Z) = 1	Not higher (Less than or equal)
je	Z = 1	Equal
jne	Z = 0	Not equal
jv	OV = 1	Overflow
jnv	OV = 0	No overflow
jn	S = 1	Negative
jp	S = 0	Positive
jc	CY = 1	Carry
jnc	CY = 0	No carry
jz	Z = 1	Zero
jnz	Z = 0	Not zero
jbr	---	Always (Unconditional)
jsa	SAT = 1	Saturated

[Description]

- If the following is specified for disp22, the assembler generates one bcond machine instruction[Note].

(a) Absolute expression having a value in the range of -256 to +255

(b) Relative expression having a PC offset reference for a label with a definition in the same section and the same file as this instruction and having a value in the range of -256 to +255

j*cnd* disp9	b*cnd* disp9

Note The b*cnd* machine instruction takes an immediate value in the range of -256 to +255 (0xFFFFFF00 to 0xFF) as the displacement.

- If the following is specified as disp22, the assembler executes instruction expansion and generates two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -256 to +255 but within the range of -2,097,150 to +2,097,153[Note 1]

(b) Relative expression having a PC offset reference of label with a definition in the same section of the same file as this instruction and having a value exceeding the range of -256 to +255 but within the range of -2,097,150 to +2,097,153

(c) Relative expression having a PC offset reference of label without a definition in the same file or section as this instruction

jbr disp22	jr disp22
jsa disp22	bsa Label1 br Label2 Label1: jr disp22 - 4 Label2:
j*cnd* disp22	bn*cnd* Label[Note 2] jr disp22 - 2 Label:

Notes 1. The range of -2,097,150 to +2,097,153 applies to instructions other than jbr and jsa. The range for the jbr instruction is from -2,097,152 to +2,097,151, and that for the jsa instruction is from -2,097,148 to +2,097,155.

2. bn*cnd* denotes an instruction that effects control branches under opposite conditions, for example, bnz for bz or ble for bgt.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of -2,097,150 to +2,097,153, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction, and having a value exceeding the range of -2,097,150 to +2,097,153, is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550230: illegal operand (range error in displacement)
```

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction, and having an odd-numbered value, is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

jarl

Jump and register link.

[Syntax]

- jarl disp22, reg2
- jarl disp32, reg1 [V850E2]

The following can be specified for disp22:
- Absolute expression having a value of up to 22 bits
- Relative expression having a PC offset reference of label

The following can be specified for disp32:
- Absolute expression having a value of up to 32 bits
- Relative expression having a PC offset reference of label

[Function]

- Syntax "jarl disp22, reg2"
  Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand. The return address is stored in the register specified by the second operand.
- Syntax "jarl disp32, reg1"
  Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand. The return address is stored in the register specified by the second operand.

[Description]

- If the instruction is executed in syntax "jarl disp22, reg2", the assembler generates one jarl machine instruction[Note] if any of the following expressions are specified for disp22.

  (a) Absolute value in the range of -2,097,152 to +2,097,151

  (b) Relative expression that has a PC offset reference of label having a definition in the same section and the same file as this instruction, and which has a value in the range of -2,097,152 to +2,097,151

  (c) Relative expression having a PC offset reference of a label having no definition in the same file or section as this instruction

  Note The jarl machine instruction takes an immediate value in the range of -2,097,152 to +2,097,151 (0xFE00000 to 0x1FFFFF) as the operand.

- If the instruction is executed in syntax "jarl disp32, reg1", the assembler generates one jarl machine instruction (6-byte long instruction).

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression that exceeds the range of -2,097,152 to +2,097,151, or a relative expression having a PC offset reference of label with a definition in the same section and the same file as this instruction and having a value that falls outside the range of -2,097,152 to +2,097,151 is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550230: illegal operand (range error in displacement)
```

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction and having an odd-numbered value, is specified as disp22/disp32, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

- When the assembler option -Xfar_jump is not specified, and an absolute expression outside of the range -2,097,152 to +2,097,151 or a relative expression outside of the range -2,097,152 to +2,097,151, having a label PC offset reference with a definition in the same file and same section as this instruction, is specified as disp32, the following message is output and assembly is stopped.

```
E0550230: illegal operand (range error in displacement)
```

- If r0 is specified as reg1/reg2, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

jarl22

Jump and register link. [V850E2]

[Syntax]

- jarl22 disp22, reg1

The following can be specified for disp22:

- Absolute expression having a value of up to 22 bits
- Relative expression having a PC offset reference of label

[Function]

Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand. The return address is stored in the register specified by the second operand.

[Description]

- If the following is specified for disp22, the assembler generates one jarl machine instruction[Note].

(a) Absolute value in the range of -2,097,152 to +2,097,15

(b) Relative expression that has a PC offset reference of label having a definition in the same section and the same file as this instruction, and which has a value in the range of -2,097,152 to +2,097,151

(c) Relative expression having a PC offset reference of a label having no definition in the same file or section as this instruction

Note The jarl machine instruction takes an immediate value in the range of -2,097,152 to +2,097,151 (0xFE00000 to 0x1FFFFF) as the operand.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression that exceeds the range of -2,097,152 to +2,097,151, or a relative expression having a PC offset reference of label with a definition in the same section and the same file as this instruction and having a value that falls outside the range of -2,097,152 to +2,097,151 is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550230: illegal operand (range error in displacement)
```

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction and having an odd-numbered value, is specified as disp22, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

- If r0 is specified as reg2, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

jarl32

Jump and register link. [V850E2]

[Syntax]

- jarl32 disp32, reg1

The following can be specified for disp32:

- Absolute expression having a value of up to 32 bits
- Relative expression having a PC offset reference of label

[Function]

Transfers control to the address attained by adding the current program counter (PC) value and the relative or absolute expression value specified by the first operand. The return address is stored in the register specified by the second operand.

[Description]

The assembler generates one jarl machine instruction (6-byte long instruction).

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having an odd-numbered value, or a relative expression having a PC offset reference of a label with a definition in the same section and the same file as this instruction, and having an odd-numbered value, is specified as disp32, the assembler outputs the following message and stops assembling.

```
E0550226: illegal operand (must be even displacement)
```

- If r0 is specified as reg1, the assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

4.7.11 Bit manipulation instructions

This section describes the bit manipulation instructions. Next table lists the instructions described in this section.

Table 4-41. Bit Manipulation Instructions

Instruction	Meanings
set1	Sets bit
clr1	Clears bit
not1	Inverts bit
tst1	Tests bit

set1

Set s bit.

[Syntax]

- set1 bit#3, disp[reg1]
- set1 reg2, [reg1]
- set1 BITIO

The following can be specified for disp:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

Caution The disp cannot be specified in syntax "set1 reg2, [reg1]".

[Function]

- Syntax "set1 bit#3, disp[reg1]"
  Sets the bit specified by the first operand of the data indicated by the address specified by the second operand. The bits other than the one specified are not affected.
- Syntax "set1 reg2, [reg1]"
  Sets the bit specified by the lower 3 bits of the register value specified by the first operand of the data indicated by the address specified by the register value of the second operand. The bits other than the one specified are not affected.
- Syntax "set1 BITIO"
  Sets the bit specified by the peripheral I/O register bit name (only reserved words defined in the device file) in the data indicated by the address specified by the first operand

[Description]

- If the following is specified for disp, the assembler generates one set1 machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`set1 #bit3, $label[reg1]`	`set1 #bit3, $label[reg1]`

(c) Relative expression having !label or %label

`set1 #bit3, !label[reg1]`	`set1 #bit3, !label[reg1]`
`set1 #bit3, %label[reg1]`	`set1 #bit3, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

(e) Internal register name defined in the device file

`set1 reg2, register-name[reg1]`	`set1 reg2, register-name[reg1]`

Note The set1 machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement.

- If any of the following is specified as disp, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`set1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `set1 #bit3, LOWW(disp)[r1]`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`set1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `set1 #bit3, LOWW(#label)[r1]`
`set1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `set1 #bit3, LOWW(label)[r1]`
`set1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `set1 #bit3, LOWW($label)[r1]`

- If disp is omitted, the assembler assumes 0.
- If a relative expression with #label, or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	1 if the specified bit is 0, 0 if not[Note]
SAT	---

Note The flag values shown here are those existing prior to the execution of this instruction, not those after the execution.

clr1

Clears bit.

[Syntax]

- clr1 bit#3, disp[reg1]
- clr1 reg2, [reg1]
- clr1 BITIO

The following can be specified for disp:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

Caution The disp cannot be specified in syntax "clr1 reg2, [reg1]".

[Function]

- Syntax "clr1 bit#3, disp[reg1]"
  Clears the bit specified by the first operand of the data indicated by the address specified by the second operand. The bits other than the one specified are not affected.
- Syntax "clr1 reg2, [reg1]"
  Clears the bit specified by the lower 3 bits of the register value specified by the first operand of the data indicated by the address specified by the register value of the second operand. The bits other than the one specified are not affected.
- Syntax "clr1 BITIO"
  Clears the bit specified by the peripheral I/O register bit name (only reserved words defined in the device file) in the data indicated by the address specified by the first operand.

[Description]

- If the following is specified as disp, the assembler generates one clr1 machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`clr1 #bit3, disp16[reg1]`	`clr1 #bit3, disp16[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`clr1 #bit3, $label[reg1]`	`clr1 #bit3, $label[reg1]`

(c) Relative expression having !label or %label

`clr1 #bit3, !label[reg1]`	`clr1 #bit3, !label[reg1]`
`clr1 #bit3, %label[reg1]`	`clr1 #bit3, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`clr1 #bit3, disp16[reg1]`	`clr1 #bit3, disp16[reg1]`

(e) Internal register name defined in the device file

`clr1 reg2, register-name[reg1]`	`clr1 reg2, register-name[reg1]`

Note The clr1 machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement.

- If any of the following is specified as disp, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`clr1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `clr1 #bit3, LOWW(disp)[r1]`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`clr1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `clr1 #bit3, LOWW(#label)[r1]`
`clr1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `clr1 #bit3, LOWW(label)[r1]`
`clr1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `clr1 #bit3, LOWW($label)[r1]`

- If disp is omitted, the assembler assumes 0.
- If a relative expression with #label or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] that follows the expression can be omitted. If omitted, the assembler assumes [r0] to be specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	1 if the specified bit is 0, 0 if not[Note]
SAT	---

Note The flag values shown here are those existing prior to the execution of this instruction, not those after the execution.

not1

Inverts bit.

[Syntax]

- not1 bit#3, disp[reg1]
- not1 reg2, [reg1]
- not1 BITIO

The following can be specified for disp:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

Caution The disp cannot be specified in syntax "not1 reg2, [reg1]".

[Function]

- Syntax "not1 bit#3, disp[reg1]"
  Inverts the bit specified by the first operand (0 to 1 or 1 to 0) of the data indicated by the address specified by the second operand. The bits other than the one specified are not affected.
- Syntax "not1 reg2, [reg1]"
  Inverts the bit specified by the lower 3 bits of the register value specified by the first operand (0 to 1 or 1 to 0) of the data indicated by the address specified by the register value of the second operand. The bits other than the one specified are not affected.
- Syntax "not1 BITIO"
  Inverts (from 0 to 1 or 1 to 0) the bit specified by the peripheral I/O register bit name (only reserved words defined in the device file) in the data indicated by the address specified by the first operand.

[Description]

- If the following is specified for disp, the assembler generates one not1 machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`not1 #bit3, disp16[reg1]`	`not1 #bit3, disp16[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`not1 #bit3, $label[reg1]`	`not1 #bit3, $label[reg1]`

(c) Relative expression having !label or %label

`not1 #bit3, !label[reg1]`	`not1 #bit3, !label[reg1]`
`not1 #bit3, %label[reg1]`	`not1 #bit3, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`not1 #bit3, disp16[reg1]`	`not1 #bit3, disp16[reg1]`

(e) Internal register name defined in the device file

`not1 reg2, register-name[reg1]`	`not1 reg2, register-name[reg1]`

Note The not1 machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement.

- If any of the following is specified as disp, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`not1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `not1 #bit3, LOWW(disp)[r1]`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`not1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `not1 #bit3, LOWW(#label)[r1]`
`not1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `not1 #bit3, LOWW(label)[r1]`
`not1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `not1 #bit3, LOWW($label)[r1]`

- If disp is omitted, the assembler assumes 0.
- If a relative expression with #label, or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	1 if the specified bit is 0, 0 if not[Note]
SAT	---

Note The flag values shown here are those existing prior to the execution of this instruction, not those after the execution.

tst1

Tests bit.

[Syntax]

- tst1 bit#3, disp[reg1]
- tst1 reg2, [reg1]
- tst1 BITIO

The following can be specified for disp:

- Absolute expression having a value of up to 32 bits
- Relative expression
- Either of the above expressions with HIGHW, LOWW, or HIGHW1 applied

Caution The disp cannot be specified in syntax "tst1 bit#3, disp[reg1]".

[Function]

- Syntax "tst1 bit#3, disp[reg1]"
  Sets only a flag according to the value of the bit specified by the first operand of the data indicated by the address specified by the second operand. The value of the second operand and the specified bit are not changed.
- Syntax "tst1 reg2, [reg1]"
  Sets only a flag according to the value of the bit of the lower 3 bits of the register value specified by the first operand of the data indicated by the address specified by the second operand. The value of the second operand and the specified bit are not changed.
- Syntax "tst1 BITIO"
  Sets only the flag in accordance with the value of the bit specified by the peripheral I/O register bit name (only reserved words defined in the device file) in the data indicated by the address specified by the first operand. The value of the peripheral I/O register bit is not affected.

[Description]

- If the following is specified for disp, the assembler generates one tst1 machine instruction[Note].

(a) Absolute expression having a value in the range of -32,768 to +32,767

`tst1 #bit3, disp16[reg1]`	`tst1 #bit3, disp16[reg1]`

(b) Relative expression having $label for a label having a definition in the sdata/sbss-attribute section

`tst1 #bit3, $label[reg1]`	`tst1 #bit3, $label[reg1]`

(c) Relative expression having !label or %label

`tst1 #bit3, !label[reg1]`	`tst1 #bit3, !label[reg1]`
`tst1 #bit3, %label[reg1]`	`tst1 #bit3, %label[reg1]`

(d) Expression with HIGHW, LOWW, or HIGHW1

`tst1 #bit3, disp16[reg1]`	`tst1 #bit3, disp16[reg1]`

(e) Internal register name defined in the device file

`tst1 reg2, register-name[reg1]`	`tst1 reg2, register-name[reg1]`

Note The tst1 machine instruction takes an immediate value in the range of -32,768 to +32,767 (0xFFFF8000 to 0x7FFF) as the displacement.

- If any of the following is specified as disp, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression having a value exceeding the range of -32,768 to +32,767

`tst1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `tst1 #bit3, LOWW(disp)[r1]`

(b) Relative expression having #label or label, or that having $label for a label having no definition in the sdata/sbss-attribute section

`tst1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `tst1 #bit3, LOWW(#label)[r1]`
`tst1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `tst1 #bit3, LOWW(label)[r1]`
`tst1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `tst1 #bit3, LOWW($label)[r1]`

- If disp is omitted, the assembler assumes 0.
- If a relative expression with #label, or a relative expression with #label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.
- If a relative expression with $label, or a relative expression with $label and with HIGHW, LOWW, or HIGHW1 applied is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [gp] is specified.
- If a peripheral I/O register name defined in the device file is specified as disp, [reg1] can be omitted. If omitted, the assembler assumes that [r0] is specified.

[Flag]

CY	---
OV	---
S	---
Z	1 if the specified bit is 0, 0 if not
SAT	---

4.7.12 Stack manipulation instructions

This section describes the stack manipulation instructions. Next table lists the instructions described in this section.

Table 4-42. Stack Manipulation Instructions

Instruction	Meanings
push	Pushes to stack area (single register)
pushm	Pushes to stack area (multiple registers)
pop	Pops from stack area (single register)
popm	Pops from stack area (multiple registers)

push

Pushes to stack area (single register).

[Syntax]

push reg

[Function]

Pushes the value of the register specified by the operand to the stack area.

[Description]

- When the push instruction is executed, the assembler executes instruction expansion to generate two or more machine instructions.

`push reg`	`add -4, sp` `st.w reg, [sp]`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

Caution Instruction expansion is performed, and set via an add instruction.

pushm

Pushes to stack area (multiple registers).

[Syntax]

pushm reg1, reg2, ..., regN

[Function]

Pushes the values of the registers specified by the operand to the stack area. Up to 32 registers can be specified by the operand.

[Description]

- When the pushm instruction is executed, the assembler executes instruction expansion to generate two or more machine instructions.
  - When there are four or fewer registers.

`pushm reg1, reg2, …, regN`	`add -4 * N, sp`
	`st.w regN, 4 * (N - 1)[sp]`
	`:`
	`st.w reg2, 4 * 1[sp]`
	`st.w reg1, 4 * 0[sp]`

  - When there are five or more registers.

`pushm reg1, reg2, …, regN`	`addi -4 * N, sp, sp`
	`st.w regN, 4 * (N - 1)[sp]`
	`:`
	`st.w reg2, 4 * 1[sp]`
	`st.w reg1, 4 * 0[sp]`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

Caution Instruction expansion is performed, and set via an add/addi instruction.

pop

Pops from stack area (single register).

[Syntax]

pop reg

[Function]

Pops the value of the register specified by the operand from the stack area.

[Description]

- When the pop instruction is executed, the assembler executes instruction expansion to generate two or more machine instructions.

`pop reg`	`ld.w [sp], reg` `add 4, sp`

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

Caution Instruction expansion is performed, and set via an add instruction.

popm

Pops from stack area (multiple registers).

[Syntax]

popm reg1, reg2, ..., regN

[Function]

Pops the values of the registers specified by the operand from the stack area in the sequence in which the registers are specified. Up to 32 registers can be specified by the operand.

[Description]

- When the popm instruction is executed, the assembler executes instruction expansion to generate two or more machine instructions.
  - When there are three or fewer registers.

popm reg1, ···, regN	ld.w 4 * 0[sp], reg1 : ld.w 4 * (N - 1)[sp], regN add 4 * N, sp

  - When there are four or more registers.

popm reg1, reg2, ···, regN	ld.w 4 * 0[sp], reg1 ld.w 4 * 1[sp], reg2 : ld.w 4 * (N - 1)[sp], regN addi 4 * N, sp, sp

[Flag]

CY	1 if a carry occurs from MSB (Most Significant Bit), 0 if not
OV	1 if Integer-Overflow occurs, 0 if not
S	1 if the result is negative, 0 if not
Z	1 if the result is 0, 0 if not
SAT	---

Caution Instruction expansion is performed, and set via an add/addi instruction.

4.7.13 Special instructions

This section describes the special instructions. Next table lists the instructions described in this section.

Table 4-43. Special Instructions

Instruction	Meanings
ldsr	Loads to system register
stsr	Stores contents of system register
di	Disables maskable interrupt
ei	Enables maskable interrupt
reti	Returns from trap or interrupt routine
eiret	Returns from EI level exception [V850E2V3]
feret	Returns fromFE level exception [V850E2V3]
halt	Stops the processor
trap	Software trap
rmtrap	Runtime monitor trap [V850E2V3]
fetrap	FE level software exception instruction [V850E2V3]
nop	No operation
switch	Table reference branch
callt	Table reference call
ctret	Returns from callt
caxi	Compare and exchange [V850E2V3]
rie	Reserved Instruction exception [V850E2V3]
syncm	Synchronize memory [V850E2V3]
syncp	Synchronize pipline [V850E2V3]
dbtrap	Debug trap
dbret	Returns from debug trap
prepare	Generates stack frame (preprocessing of function)
dispose	Deletes stack frame (post processing of function)
synce	Synchronize exception [V850E2V3]
syscall	System call exception [V850E2V3]

See the device with an instruction set of V850E2V3 product user's manual and architecture edition for details about the device with an instruction set of V850E2V3.

ldsr

Loads to system register.

[Syntax]

- ldsr reg, regID

The following can be specified as regID:

- Absolute expression having a value of up to 5 bits

[Function]

Stores the value of the register specified by the first operand in the system register[Note] indicated by the system register number specified by the second operand.

Note For details of the system registers, see the Relevant Device's Hardware User's Manual provided with the each device.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

Caution If the program status word (PSW) is specified as the system register, the value of the corresponding bit of reg is set as each flag.

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 31 is specified as regID, the assembler outputs the following message, then continues assembling using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate)
```

Note The ldsr machine instruction takes an immediate value in the range of 0 to 31 (0x0 to 0x1F) as the second operand.

- If a reserved register number, the number of a register which cannot be accessed (such as ECR) or the number of a register which can be accessed only in the debug mode is specified as regID, the assembler outputs the following message and continues assembling as is.

```
W0550018: illegal regID for ldsr
```

stsr

Stores contents of system register.

[Syntax]

- stsr regID, reg

The following can be specified as regID:

- Absolute expression having a value of up to 5 bits

[Function]

Stores the value of the system register[Note] indicated by the system register number specified by the first operand, to the register specified by the second operand.

Note For details of the system registers, see the Relevant Device's Hardware User's Manual provided with the each device.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value exceeding the range of 0 to 31 is specified as regID, the assembler outputs the following message, then continues assembling using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate)
```

Note The stsr machine instruction takes an immediate value in the range of 0 to 31 (0x0 to 0x1F) as the first operand.

- If a reserved register number or the number of a register which can be accessed only in the debug mode is specified as regID, the assembler outputs the following message and continues assembling as is.

```
W0550018: illegal regID for ldsr
```

di

Disables maskable interrupt.

[Syntax]

- di

[Function]

Sets the ID bit of the PSW to 1 and disables acknowledgement of maskable interrupts since this instruction has already been executed.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---
ID	1

ei

Enables maskable interrupt.

[Syntax]

- ei

[Function]

Sets the ID bit of the PSW to 0, and enables acknowledgment of maskable interrupt from the next instruction.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---
ID	0

reti

Returns from trap or interrupt routine.

[Syntax]

- reti

[Function]

Returns from a trap or interrupt routine[Note].

Note For details of the function, see the Relevant Device's Architecture User's Manual of each devic

[Flag]

CY	Extracted value
OV	Extracted value
S	Extracted value
Z	Extracted value
SAT	Extracted value

halt

Stops the processor.

[Syntax]

- halt

[Function]

Stops the processor and sets it in the HALT status. The HALT status can be released by a maskable interrupt, NMI, or reset.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

trap

Software trap.

[Syntax]

- trap vector

The following can be specified for vector:

- Absolute expression having a value of up to 5 bits

[Function]

Causes a software trap[Note].

Note For details of the function, see the Relevant Device's Architecture User's Manual of each device.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If an absolute expression having a value falling outside the range of 0 to 31 is specified as vector, the assembler outputs the following message, continuing assembling using the lower 5 bits[Note] of the specified value.

```
W0550011: illegal operand (range error in immediate)
```

Note The trap machine instruction takes an immediate value in the range of 0 to 31 (0x0 to 0x1F) as an operand.

nop

No operation.

[Syntax]

- nop

[Function]

Nothing is executed. This instruction can be used to allocate an area during an instruction sequence or to insert a delay cycle during instruction execution.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

switch

Table reference branch.

[Syntax]

switch reg

[Function]

Performs processing in the following sequence.

(1) Adds the value resulting from logically shifting the value specified by the operand 1 bit to the left to the first address of the table (address following the switch instruction) to generate a table entry address.

(2) Loads signed halfword data from the generated table entry address.

(3) Logically shifts the loaded value 1 bit to the left and sign-extends it to word length. Then adds the first address of the table to it to generate an address

(4) Branches to the generated address.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

[Caution]

- If r0 is specified by reg, the assembler outputs the following message and stops assembling.

E0550239: Illegal operand (cannot use r0 as source in V850E mode).

callt

Table reference call.

[Syntax]

- callt imm6

The following can be specified as imm6:

- Absolute expression having a value of up to 6 bits

[Function]

Performs processing in the following sequence[Note]

(1) Saves the values of the return PC and PSW to CTPC and CTPSW.

(2) Generates a table entry address by shifting the value specified by the operand 1 bit to the left as an offset value from CTBP(CALLT Base Pointer) and by adding it to the CTBP value.

(3) Loads unsigned halfword data from the generated table entry address.

(4) Adds the loaded value to the CTBP value to generate an address.

(5) Branches to the generated address.

Note For details of the system registers, see the Relevant Device's Architecture User's Manual of each device.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

ctret

Returns from callt.

[Syntax]

- ctret

[Function]

Returns from the processing by callt. Performs the processing in the following sequence[Note]:

(1) Extracts the return PC and PSW from CTPC and CTPSW.

(2) Sets the extracted values in the PC and PSW and transfers control.

Note For details of the system registers, see the Relevant Device's Architecture User's Manual of each device.

[Flag]

CY	Extracted value
OV	Extracted value
S	Extracted value
Z	Extracted value
SAT	Extracted value

dbtrap

Debug trap.

[Syntax]

- dbtrap

[Function]

Causes debug trap[Note].

Note For details of the function, see the Relevant Device's Architecture User's Manual of each device.

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

dbret

Returns from debug trap.

[Syntax]

- dbret

[Function]

Returns from debug trap[Note].

Note For details of the function, see the Relevant Device's Architecture User's Manual of each device.

[Flag]

CY	Extracted value
OV	Extracted value
S	Extracted value
Z	Extracted value
SAT	Extracted value

prepare

Generates stack frame (preprocessing of function).

[Syntax]

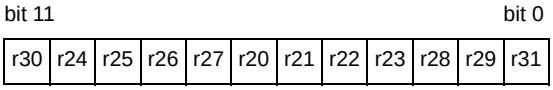
- prepare list, imm1
- prepare list, imm1, imm2
- prepare list, imm1, sp

The following can be specified as imm1/imm2:

- Absolute expression having a value of up to 32 bits

list specifies the 12 registers that can be pushed by the prepare instruction.The following can be specified as list.

- Register
  Specify the registers (r20 to r31) to be pushed, delimiting each with a comma.
- 1Constant expression having a value of up to 12 bits
  The 12 bits and 12 registers correspond as follows:



The following two specifications are equivalent.

`prepare r26, r29, r31, 0x10`	`prepare 0x103, 0x10`

[Function]

The prepare instruction performs the preprocessing of a function.

- Syntax "prepare list, imm1"

(a) Pushes one of the registers specified by the first operand and subtracts 4 from the stack pointer (sp).

(b) Repeatedly performs (a) until all the registers specified by the first operand have been pushed.

(c) Subtracts the value of the absolute expression specified by the second operand from sp[Note] and sets sp in the register saving area.

- Syntax "prepare list, imm1, imm2"

(a) Pushes one of the registers specified by the first operand and subtracts 4 from sp.

(b) Repeatedly performs (a) until all the registers specified by the first operand have been pushed.

(c) Subtracts the value of the absolute expression specified by the second operand from sp[Note] and sets sp to the register saving area.

(d) Sets the value of the absolute expression specified by the third operand in ep.

- Syntax "prepare list, imm1, sp"

(a) Pushes one of the registers specified by the first operand and subtracts 4 from sp.

(b) Repeatedly performs (a) until all the registers specified by the first operand have been pushed.

(c) Subtracts the value of the absolute expression specified by the second operand from sp[Note] and sets sp in the register saving area.

(d) Sets the value of sp specified by the third operand in ep.

Note Since the value actually subtracted from sp by the machine instruction is imm1 shifted 2 bits to the left, the assembler shifts the specified imm1 2 bits to the right in advance and reflects it in the code.

[Description]

- If the following is specified for imm1, the assembler generates one prepare machine instruction.

(a) Absolute expression having a value in the range of 0 to 127

`prepare list, imm1`	`prepare list, imm1`
`prepare list, imm1, imm2`	`prepare list, imm1, imm2`
`prepare list, imm1, sp`	`prepare list, imm1, sp`

- If anything other than a constant expression[Note] is specified as list, the assembler outputs the following message and stops assembling.

E0550249: illegal syntax

Note Undefined symbol and label reference.

- When the following is specified as imm1, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression exceeding the range of 0 to 127, but within the range of 0 to 32,767

`prepare list, imm1`	`prepare list, 0` `movea -imm1, sp, sp`
`prepare list, imm1, imm2`	`prepare list, 0, imm2` `movea -imm1, sp, sp`
`prepare list, imm1, sp`	`prepare list, 0, sp` `movea -imm1, sp, sp`

(b) Absolute expression having a value exceeding the range of 0 to 32,767

`prepare list, imm1`	`prepare list, 0`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, imm2`	`prepare list, 0, imm2`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, sp`	`prepare list, 0, sp`
	`mov imm1, r1`
	`sub r1, sp`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

Caution If a sub instruction is generated as a result of instruction expansion, the flag value may be affected.

[Caution]

- An address consisting of the two lower bits specified by sp is masked to 0 even though misalign access is enabled. In sp, set a value which is aligned with a four-byte boundary.

dispose

Deletes stack frame (post processing of function).

[Syntax]

- dispose imm, list
- dispose imm, list, [reg]

The following can be specified for imm:

- Absolute expression having a value of up to 32 bits

The following can be specified as list. list specifies the 12 registers that can be popped by the dispose instruction.

- Register
  Specify the registers (r20 to r31) to be popped, delimiting each with a comma.
- Constant expression having a value of up to 12 bits
  The 12 bits and 12 registers correspond as follows:

bit 11 bit 0

r30	r24	r25	r26	r27	r20	r21	r22	r23	r28	r29	r31

The following two specifications are equivalent.

`dispose 0x10, r26, r29, r31`	`dispose 0x10, 0x103`

[Function]

The dispose instruction performs the postprocessing of a function.

- Syntax "dispose imm, list"

(a) Adds the value of the absolute expression specified by the first operand to the stack pointer (sp)[Note] and sets sp in the register saving area.

(b) Pops one of the registers specified by the second operand and adds 4 to sp.

(c) Repeatedly executes (b) until all the registers specified by the second operand have been popped.

- Syntax "dispose imm, list, [reg]"

(a) Adds the value of the absolute expression specified by the first operand to the stack pointer (sp)[Note] and sets sp in the register saving area.

(b) Pops one of the registers specified by the second operand and adds 4 to sp.

(c) Repeatedly executes (b) until all the registers specified by the second operand have been popped.

(d) Sets the register value specified by the third operand in the program counter (PC).

Note Since the value actually added to sp by the machine instruction is imm shifted 2 bits to the left, the assembler shifts the specified imm 2 bits to the right in advance and reflects it in the code.

[Description]

- If the following is specified for imm, the assembler generates one dispose machine instruction.

(a) Absolute expression having a value in the range of 0 to 127

`dispose imm, list`	`dispose imm, list`
`dispose imm, list, [reg]`	`dispose imm, list, [reg]`

- If anything other than a constant expression is specified as list, the assembler outputs the following message and stops assembling.

E0550249: illegal syntax

- If the following is specified for imm, the assembler executes instruction expansion to generate two or more machine instructions.

(a) Absolute expression exceeding the range of 0 to 127, but within the range of 0 to 32,767

`dispose imm, list`	`movea imm, sp, sp` `dispose 0, list`
`dispose imm, list, [reg]`	`movea imm, sp, sp` `dispose 0, list, [reg]`

(b) Absolute expression having a value exceeding the range of 0 to 32,767

`dispose imm, list`	`mov imm, r1`
	`add r1, sp`
	`dispose 0, list`

`dispose imm, list, [reg]`	`mov imm, r1`
	`add r1, sp`
	`dispose 0, list, [reg]`

[Flag]

CY	---
OV	---
S	---
Z	---
SAT	---

Caution If the add instruction is generated as a result of instruction expansion, the flag value may be affected.

[Caution]

- An address consisting of the two lower bits specified by sp is masked to 0 even though misalign access is enabled. In sp, set a value which is aligned with a four-byte boundary.
- If r0 is specified by the [reg] in syntax "dispose imm, list, [reg]", tthe assembler outputs the following message and stops assembling.

```
E0550240: Illegal operand (cannot use r0 as destination in V850E mode).
```

4.7.14 Floating-point operation instructions [V850E2V3]

Next table lists the floating-point operation instructions.

See the device with an instruction set of V850E2V3 product user's manual and architecture edition for details.

Table 4-44. Floating-point Operation Instructions (Basic Operation Instructions)

Instruction	Meanings
absf.d	Floating-point absolute value (double)
absf.s	Floating-point absolute value (single)
addf.d	Floating-point add (double)
addf.s	Floating-point add (single)
divf.d	Floating-point divide (double)
divf.s	Floating-point divide (single)
maxf.d	Floating-point maximum (double)
maxf.s	Floating-point maximum (single)
minf.d	Floating-point minimum (double)
minf.s	Floating-point minimum (single)
mulf.d	Floating-point multiply (double)
mulf.s	Floating-point multiply (single)
negf.d	Floating-point negate (double)
negf.s	Floating-point negate (single)
recipf.d	Reciprocal of a floating-point value (double)
recipf.s	Reciprocal of a floating-point value (single)
rsqrtf.d	Reciprocal of the square root of a floating-point value (double)
rsqrtf.s	Reciprocal of the square root of a floating-point value (single)
sqrtf.d	Floating-point square root (double)
sqrtf.s	Floating-point square root (single)
subf.d	Floating-point subtract (double)
subf.s	Floating-point subtract (single)

Table 4-45. Floating-point Operation Instructions (Expansion Basis Operation Instructions)

Instruction	Meanings
maddf.s	Floating-point multiply-add (single)
msubf.s	Floating-point multiply-add (single)
nmaddf.s	Floating-point multiply-add (single)
nmsubf.s	Floating-point multiply-add (single)

Table 4-46. Floating-point Operation Instructions (Exchange Instructions)

Instruction	Meanings
ceilf.dl	Floating-point ceiling to Integer Format (double)
ceilf.dw	Floating-point ceiling to integer format (double)
ceilf.dul	Floating-point ceiling to unsighned integer format (double)
ceilf.duw	Floating-point ceiling to unsighned integer format (double)
ceilf.sl	Floating-point ceiling to integer format (single)
ceilf.sw	Floating-point ceiling to integer format (single)
ceilf.sul	Floating-point ceiling to unsighned integer format (single)
ceilf.suw	Floating-point ceiling to unsighned integer format (single)
cvtf.dl	Floating-point ceiling to integer format (double)
cvtf.ds	Floating-point convert to floating-point format (double)
cvtf.dul	Floating-point ceiling to unsighned integer format (double)
cvtf.duw	Floating-point ceiling to unsighned integer format (double)
cvtf.dw	Floating-point ceiling to integer format (double)
cvtf.ld	Floating-point convert to floating-point format (double)
cvtf.ls	Floating-point convert to floating-point format (single)
cvtf.sd	Floating-point convert to floating-point format (double)
cvtf.sl	Floating-point ceiling to integer format (single)
cvtf.sul	Floating-point ceiling to unsighned integer format (single)
cvtf.suw	Floating-point ceiling to unsighned integer format (single)
cvtf.sw	Floating-point ceiling to integer format (single)
cvtf.uld	Floating-point convert to floating-point format (double)
cvtf.uls	Floating-point convert to floating-point format (single)
cvtf.uwd	Floating-point convert to floating-point format (double)
cvtf.uws	Floating-point convert to floating-point format (single)
cvtf.wd	Floating-point convert to floating-point format (double)
cvtf.ws	Floating-point convert to floating-point format (single)
floorf.d	Floating-point ceiling to integer format (double)
floorf.dw	Floating-point ceiling to integer format (double)
floorf.dul	Floating-point ceiling to unsighned integer format (double)
floorf.duw	Floating-point ceiling to unsighned integer format (double)
floorf.sl	Floating-point ceiling to integer format (single)
floorf.sw	Floating-point ceiling to integer format (single)
floorf.sul	Floating-point ceiling to unsighned integer format (single)
floorf.suw	Floating-point ceiling to unsighned integer format (single)
trncf.dl	Floating-point ceiling to integer format (double)
trncf.dul	Floating-point ceiling to unsighned integer format (double)

Instruction	Meanings
trncf.duw	Floating-point ceiling to unsighned integer format (double)
trncf.dw	Floating-point ceiling to integer format (double)
trncf.sl	Floating-point ceiling to integer format (single)
trncf.sul	Floating-point ceiling to unsighned integer format (single)
trncf.suw	Floating-point ceiling to unsighned integer format (single)
trncf.sw	Floating-point ceiling to integer format (single)

Table 4-47. Floating-point Operation Instructions (Compare Instructions)

Instruction	Meanings
cmpf.s	Floating-point compare (single)
cmpf.d	Floating-point compare (double)

Table 4-48. Floating-point Operation Instructions (Conditional Move Instructions)

Instruction	Meanings
cmovf.s	Floating-point conditional move (single)
cmovf.d	Floating-point conditional move (double)

Table 4-49. Floating-point Operation Instructions (Conditional Bit Move Instructions)

Instruction	Meanings
trfsr	Transfer floating flags

cmpf.s

Floating-point compare (single)

[Syntax]

- cmpf.s imm4, reg1, reg2, cc#3
- cmpf*cnd*.s reg1, reg2

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits

[Function]

- Syntax "cmpf.s imm4, reg1, reg2, cc#3"
  The content in single-precision floating-point format in the register pair specified by reg2 is compared with the content in single-precision floating-point format in the register pair specified by reg1, via the imm4 comparison condition. The result (1 if true; 0 if false) is set in the condition bit (CC(7:0) bits; bits 31-24) in the FPSR register specified via cc#3. If cc#3 is omitted, it is set in the CC0 bit (bit 24).
- Syntax "cmpf*cnd*.s reg1, reg2"
  Via cmpf*cnd*.s, a corresponding "cmpf.s" instruction is generated (see "Table 4-50. cmpfcnd.s Instruction List" for details), and expanded in the format "cmpf.s imm4, reg1, reg2, cc#3". The content in single-precision floating-point format in the register pair specified by reg2 is compared with the content in single-precision floating-point format in the register pair specified by reg1, via the comparison condition. The result (1 if true; 0 if false) is set in the condition bit (CC(7:0) bits; bits 31-24) in the FPSR register specified via cc#3. If cc#3 is omitted, it is set in the CC0 bit (bit 24).

[Description]

- If the instruction is executed in syntax "cmpf.s imm4, reg1, reg2, cc#3", the assembler generates one cmpf.s machine instruction.
- If the instruction is executed in syntax "cmpf*cnd*.s reg1, reg2", the assembler generates the corresponding cmpf.s instruction (see "Table 4-50. cmpfcnd.s Instruction List") and expands it to syntax "cmpf.s imm4, reg1, reg2, cc#3".

Table 4-50. cmpf*cnd*.s Instruction List

Instruction	Condition	Meaning of Condition	Instruction Expansion
cmpff.s	FALSE	Always false	cmpf.s 0x0
cmpfun.s	Unordered	At least one of reg1 and reg2 is a non-number	cmpf.s 0x1
cmpfeq.s	reg2 = reg1	Neither is a non-number, and they are equal	cmpf.s 0x2
cmpfueq.s	reg2 ?= reg1	At least one is a non-number, or they are equal	cmpf.s 0x3
cmpfolt.s	reg2 < reg1	Neither is a non-number, and less than	cmpf.s 0x4
cmpfult.s	reg2 ?< reg1	At least one is a non-number, or less than	cmpf.s 0x5
cmpfole.s	reg2 <= reg1	Neither is a non-number, and less than or equal	cmpf.s 0x6
cmpfule.s	reg2 ?<= reg1	At least one is a non-number, or less than or equal	cmpf.s 0x7
cmpfsf.s	FALSE	Always false	cmpf.s 0x8
cmpfngle.s	Unordered	At least one of reg1 and reg2 is a non-number	cmpf.s 0x9

Instruction	Condition	Meaning of Condition	Instruction Expansion
cmpfseq.s	reg2 = reg1	Neither is a non-number, and they are equal	cmpf.s 0xA
cmpfngl.s	reg2 ?= reg1	At least one is a non-number, or they are equal	cmpf.s 0xB
cmpflt.s	reg2 < reg1	Neither is a non-number, and less than	cmpf.s 0xC
cmpfnge.s	reg2 ?< reg1	At least one is a non-number, or less than	cmpf.s 0xD
cmpfle.s	reg2 <= reg1	Neither is a non-number, and less than or equal	cmpf.s 0xE
cmpfngt.s	reg2 ?<= reg1	At least one is a non-number, or less than or equal	cmpf.s 0xF

Remark ?: Unordered

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the cmpf.s instruction, the following message is output, and assembly continues using the lower 4 bits of the specified value.

W0550011: illegal operand (range error in immediate).

cmpf.d

Floating-point compare (double)

[Syntax]

- cmpf.d imm4, reg1, reg2, cc#3
- cmpf*cnd*.d reg1, reg2

The following can be specified for imm4:

- Absolute expression having a value up to 4 bits

[Function]

- Syntax "cmpf.d imm4, reg1, reg2, cc#3"
  The content in double-precision floating-point format in the register pair specified by reg2 is compared with the content in double-precision floating-point format in the register pair specified by reg1, via the imm4 comparison condition. The result (1 if true; 0 if false) is set in the condition bit (CC(7:0) bits; bits 31-24) in the FPSR register specified via cc#3. If cc#3 is omitted, it is set in the CC0 bit (bit 24).
- Syntax "cmpf*cnd*.d reg1, reg2"
  Via cmpf*cnd*.d, a corresponding "cmpf.d" instruction is generated (see "Table 4-51. cmpfcnd.d Instruction List" for details), and expanded in the format "cmpf.d imm4, reg1, reg2, cc#3". The content in single-precision floating-point format in the register pair specified by reg2 is compared with the content in single-precision floating-point format in the register pair specified by reg1, via the comparison condition. The result (1 if true; 0 if false) is set in the condition bit (CC(7:0) bits; bits 31-24) in the FPSR register specified via cc#3. If cc#3 is omitted, it is set in the CC0 bit (bit 24).

[Description]

- If the instruction is executed in syntax "cmpf.d imm4, reg1, reg2, cc#3", the assembler generates one cmpf.d machine instruction.
- If the instruction is executed in syntax "cmpf*cnd*.d reg1, reg2", the assembler generates the corresponding cmpf.d instruction (see "Table 4-51. cmpfcnd.d Instruction List") and expands it to syntax "cmpf.d imm4, reg1, reg2, cc#3".

Table 4-51. cmpf*cnd*.d Instruction List

Instruction	Condition	Meaning of Condition	Instruction Expansion
cmpff.d	FALSE	Always false	cmpf.d 0x0
cmpfun.d	Unordered	At least one of reg1 and reg2 is a non-number	cmpf.d 0x1
cmpfeq.d	reg2 = reg1	Neither is a non-number, and they are equal	cmpf.d 0x2
cmpfueq.d	reg2 ?= reg1	At least one is a non-number, or they are equal	cmpf.d 0x3
cmpfolt.d	reg2 < reg1	Neither is a non-number, and less than	cmpf.d 0x4
cmpfult.d	reg2 ?< reg1	At least one is a non-number, or less than	cmpf.d 0x5
cmpfole.d	reg2 <= reg1	Neither is a non-number, and less than or equal	cmpf.d 0x6
cmpfule.d	reg2 ?<= reg1	At least one is a non-number, or less than or equal	cmpf.d 0x7
cmpfsf.d	FALSE	Always false	cmpf.d 0x8
cmpfngle.d	Unordered	At least one of reg1 and reg2 is a non-number	cmpf.d 0x9

Instruction	Condition	Meaning of Condition	Instruction Expansion
cmpfseq.d	reg2 = reg1	Neither is a non-number, and they are equal	cmpf.d 0xA
cmpfngl.d	reg2 ?= reg1	At least one is a non-number, or they are equal	cmpf.d 0xB
cmpflt.d	reg2 < reg1	Neither is a non-number, and less than	cmpf.d 0xC
cmpfnge.d	reg2 ?< reg1	At least one is a non-number, or less than	cmpf.d 0xD
cmpfle.d	reg2 <= reg1	Neither is a non-number, and less than or equal	cmpf.d 0xE
cmpfngt.d	reg2 ?<= reg1	At least one is a non-number, or less than or equal	cmpf.d 0xF

Remark ?: Unordered

[Caution]

- If an absolute expression having a value exceeding 4 bits is specified as imm4 of the cmpf.d instruction, the following message is output, and assembly continues using the lower 4 bits of the specified value.

W0550011: illegal operand (range error in immediate).

CHAPTER 5 LINK DIRECTIVE SPECIFICATIONS

This chapter explains the necessary items for link directives and how to write a link directive file.

In an embedded application such as allocating program code from certain address or allocating by division, it is necessary to pay attention in the memory allocation.

To implement the memory allocation as expected, program code or data allocation information should be specified in linker. This information is called as "Link directive" and file describing link directive is called as "Link directive file".

Linker will decide the memory allocation according to this link directive file and will create load module.

5.1 Specification Items

Items specified in the link directive generally fall into the following two categories.

- Segment directives and mapping directives
- Symbol directive

5.1.1 Segment directives and mapping directives

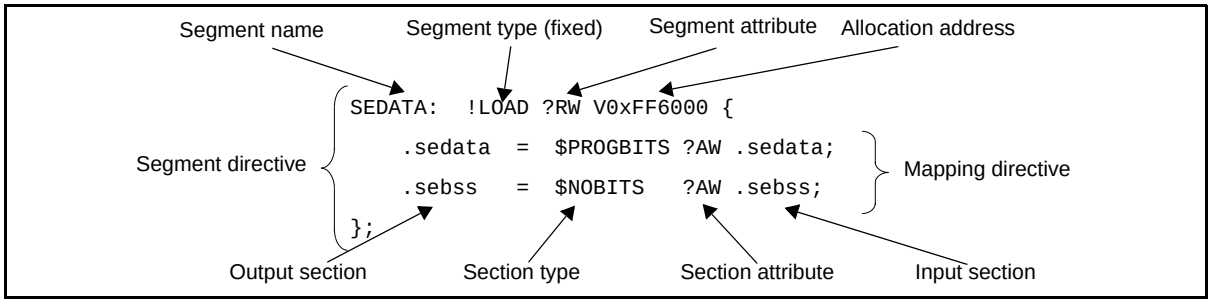
Link directives that gather information on sections where programs and data are allocated into information on segments for certain types and attributes, and that determine the corresponding allocation address.

A link directive that contains description of section information is called a "mapping directive" and a link directive that contains description of segment information is called a "segment directive".

The following shows examples of a segment directive and mapping directives that are contained in a link directive file.

For further description of the link directive format, see "5.4 Coding Method".

Figure 5-1. Segment Directives and Mapping Directives



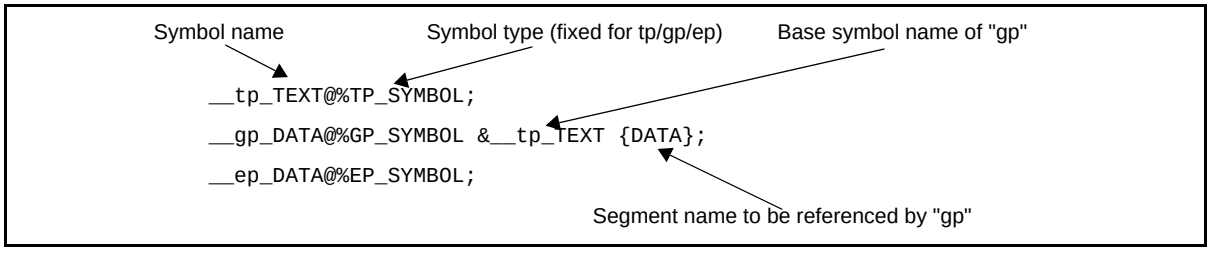
5.1.2 Symbol directive

Link directives that create "symbols" which generate tp (text pointers), gp (global pointers), and ep (element pointers): this symbol-related information is called a "symbol directive".

The following shows an example of a symbol directive that is contained in a link directive file.

For further description of the link directive format, see "5.4 Coding Method".

Figure 5-2. Symbol Directive



5.2 Sections and Segments

This section describes the sections and segments.

5.2.1 Sections

A section is the basic unit making up programs (area to which programs or data are allocated). For example, program code is allocated to a text-attribute section and variables that have initial values are allocated to a data-attribute section. In other words, different types of information are allocated to different sections.

Section names can be specified within application. In C language, they can be specified using a #pragma section directive or #pragma text directive and in assembly language they can be specified using section definition directives.

Even if the #pragma directive is not used to specify a section, however, allocation by the compiler to a particular section may already be set as the default setting in the program code or data (variables).

5.2.2 Segments

A segment is the basic unit in which programs and data are loaded to memory. Sections that have the same attribute or the same type are gathered into one section group which is called segment. In other words, the general idea is that a segment is a collection of similar sections.

A segment name, attribute, and address to which a program is loaded can be freely specified by a link directive.

Caution Some characters cannot be specified in segment names and attributes. For details, see "5.4.3 Segment directive".

The following shows code extracted from a link directive file that allocates the read-enabled (R) and executable (X) segment "TEXT1" to address 0x100000.

```
TEXT1:!LOAD ?RX V0x100000 {
        :
(Mapping directive)
        :
};
```

Since a segment is the basic unit for loading to memory, the segment is also the unit for allocating program code and data. In other words, to allocate a certain section to a specified memory area, the section information is coded in a mapping directive and then a segment that includes the mapping directive is created. Next, the segment's allocation address is determined.

Caution Although the allocation address for a mapping directive can be directly specified in a section, addresses are usually specified with segment units.

Example Allocate variable "i" to the sdata area and function "func1" to 0x120000.

- test1.c

```
#pragma section sdata
i = 10;
#pragma section default

#pragma text "f1" func1

void func1() {
            :
        return;
}
```

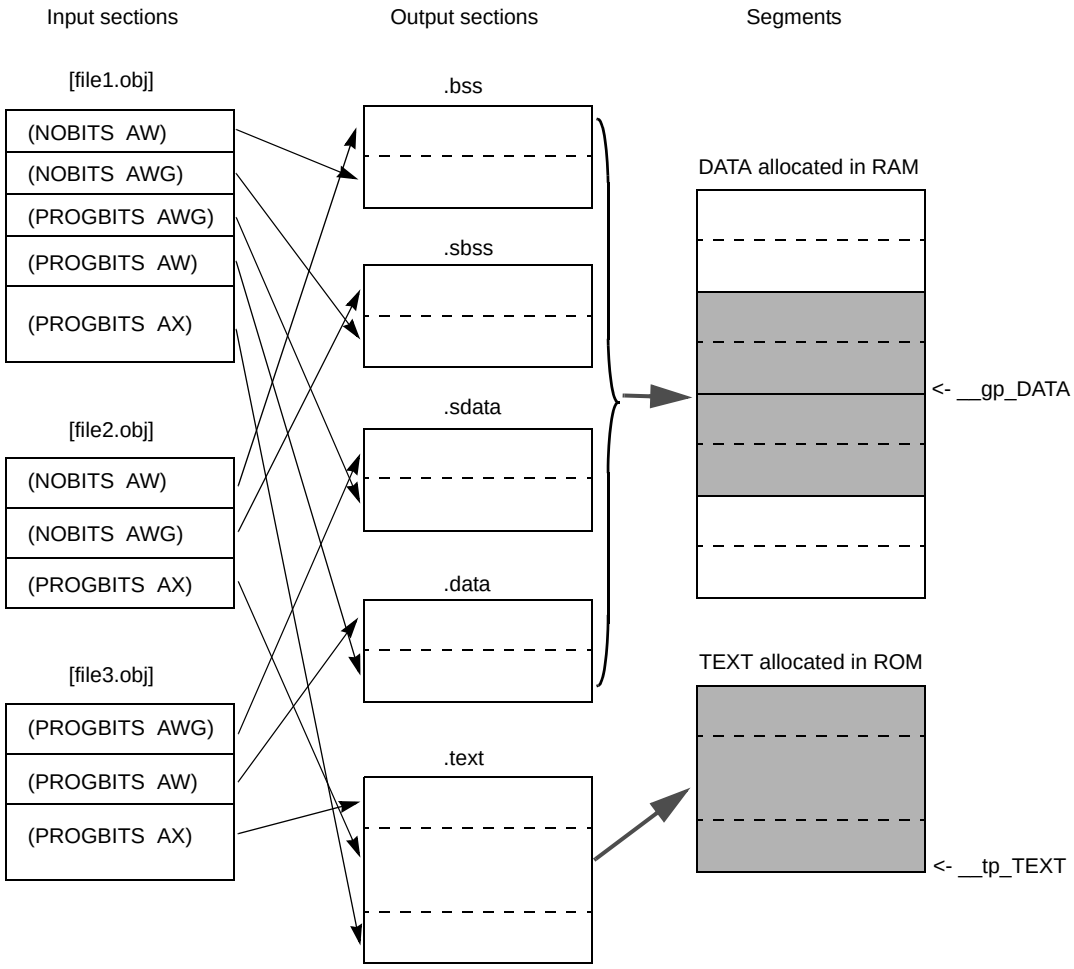
- Link directive (partial)

```
TEXT2:  !LOAD ?RX V0x120000 {
        text1= $PROGBITS ?AX f1.text;
};
DATA :  !LOAD ?R V0x200000 {
        .data   = $PROGBITS ?AW;
        .sdata  = $PROGBITS ?AWG;
        .sbss   = $NOBITS ?AWG;
        .bss    = $NOBITS ?AW;
};
            :
```

5.2.3 Relationship between segments and sections

The following shows a mapping image of the relation between segments and sections.

Figure 5-3. Relation Between Segments and Sections



Sections that are included in objects (file1.obj, file2.obj, file3.obj) are called "input sections". These sections are gathered in the same attribute. Sections that are grouped and output are called "output sections". Output section groups are also gathered in corresponding segments (DATA segment and TEXT segment) and are mapped to appropriate areas (if there is no explicit address specification).

The text pointer (tp) symbol "__tp_TEXT" and the global pointer (gp) symbol "__gp_DATA" are set according to certain rules.

5.2.4 Types of sections

The following describes the types of sections that can be handled by the CX.

"Table 5-1. CX Allocation Section Types" lists the section types that can specify the allocations, and their features.

Data for which allocation to a section is not specified by this format or section file is allocated by the CX to the .sdata section, .data section, .sbss section, or .bss section according to sizes specified by the CX's options settings[Note1].

Data for which the type qualifier const has been specified and character string constants are allocated by the CX to the .const section or .sconst section according to sizes specified by the CX's options settings[Note2].

Allocation to sections can also be specified via section files[Note3].

Notes 1. The default setting is for all data to be allocated to the .sdata or .sbss sections.

2. See "the CX's -Xsconst option" in the "CubeSuite Build for CX Compiler" for details.

3. See "Symbol Information File" in the "CubeSuite Build for CX Compiler" for details.

Table 5-1. CX Allocation Section Types

Type	Feature	Specified Character String
.tidata.byte section .tidata.word section .tibss.byte section .tibss.word section (tiny internal data/ tiny internal bss)	This sections can be referenced from ep (element pointer) with 1 instruction toward higher addresses. These sections are accessed with 1 instruction in the same manner as sidata/sibss attribute sections, but differ in terms of the assemble instruction to be used. sidata/sibss attribute sections use the 4-byte "st/ld" instruction for store/reference, whereas tidata/tibss attribute sections use the 2-byte "sst/sld" instruction to perform access. In other words, their code efficiency is better than that of sidata/sibss attribute sections. However, the range in which sst/sld instruction can be applied is small. So it is not possible to allocate a large number of variables. Data with initial values are allocated to the tidata (tidata.byte, tidata.word) attribute section, and data without initial values are allocated to the tibss (tibss.byte, tibss.word) attribute section. Specify the tidata.byte/tibss.byte attribute to allocate byte data, and specify the tidata.word/tibss.word attribute to allocate word data. To select automatic byte/word judgment by the CX, specify the tidata/tibss attribute.	tidata tidata_byte tidata_word
.data section .bss section (data/bss)	These sections can be reference from gp (global pointer) with 2 instructions. Since access (with ld/st instruction) is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed. In other words, these sections can be allocated anywhere as long as it is in RAM. Data with initial values are allocated to the data attribute section, and data without initial values are allocated to the bss attribute section.	data
.sdata section .sbss section (sdata/sbss)	These sections can be referenced from gp (global pointer) with 1 instruction (ld/st instruction), and must be allocated within +/- 32K-byte from gp (64K-byte total). Data with initial values are allocated to the sdata attribute section, and data without initial values are allocated to the sbss attribute section. The CX first attempts to generate the code to be allocated to these sections. If the code exceeds the upper limit of these attribute sections, however, code to be allocated in data/bss attribute section is generated. To increase the amount of data to be allocated to sdata/sbss attribute section, the upper size limit for the data to be allocated can be specified with the -Xsdata option of the CX so that data in excess of this upper limit is not allocated to the sdata/sbss attribute section.	sdata

Type	Feature	Specified Character String
.sedata section .sebss section (small extended data/ small extended bss)	This sections can be referenced from ep (element pointer) with 1 instruction (ld/st instruction), and they are accessed from ep toward lower addresses. In other words, theses sections are allocated within 32K-byte toward lower addresses from ep. Data with initial values are allocated to the sedata attribute section, and data without initial values are allocated to the sebss attribute section. If variables that exceed the upper limit of sdata/sbss attribute section that can be accessed from gp with 1 instruction, but which one wants to access with 1 instruction still exist, they can be allocated in the range that can be accessed with 1 instruction using ep. sidata/sibss attribute section is section for access toward higher addresses from ep, but sedata/sebss attribute section is section for access toward lower addresses from ep.	sedata
.sidata section .sibss section (small internal data/ small internal bss)	This sections can be referenced from ep (element pointer) with 1 instruction (ld/st instruction), and they are accessed from ep toward higher addresses. In other words, theses sections are allocated within 32K-byte toward higher addresses from ep. Data with initial values are allocated to the sidata attribute section, and data without initial values are allocated to the sibss attribute section. If variables that exceed the upper limit of sdata/sbss attribute section that can be accessed from gp with 1 instruction, but which one wants to access with 1 instruction still exist, they can be allocated in the range that can be accessed with 1 instruction using ep. sidata/sibss attribute section is section for access toward higher addresses from ep, but sedata/sebss attribute section is section for access toward lower addresses from ep.	sidata
.sconst section (small const data)	This section can be referenced from r0 (i.e. address 0) with 1 instruction (ld/st instruction), and must be allocated within +/- 32K-byte from address 0. Basically, data that can be fixed into ROM is allocated to this section. In the case of V850 microcontrollers with internal ROM, in many cases the internal ROM is assigned from address 0, and data that one wishes to reference with 1 instruction and that can be fixed to ROM is allocated as the sconst attribute section. In the case of devices without internal ROM, when the ROM-less mode is specified, such data is allocated to the external memory. Variables/data declared by adding the const modifier are subject to allocation to sconst/const attribute section. If the data exceeds the upper limit of these attribute sections, it is allocated to the const attribute section. To increase the amount of data to be allocated to sconst attribute section, the upper size limit for the data to be allocated can be specified with the -Xsconst option of the CX so that data in excess of this upper limit is not allocated to the sconst attribute section (See the "CubeSuite Build for CX Compiler" for the option details).	sconst
.const section (const data)	This section can be reference from r0 (i.e. address 0) with 2 instructions. Since access (with ld/st instruction) is performed after address generation, the code becomes correspondingly longer and the execution speed also drops, but the entire 32-bit space can be accessed. Data that can be fixed into ROM that exceeds the upper limit of the sconst attribute section, or data that one wishes to allocate in external ROM in the case of ROM-less devices of the V850 microcontrollers, is allocated to the const attribute section.	const

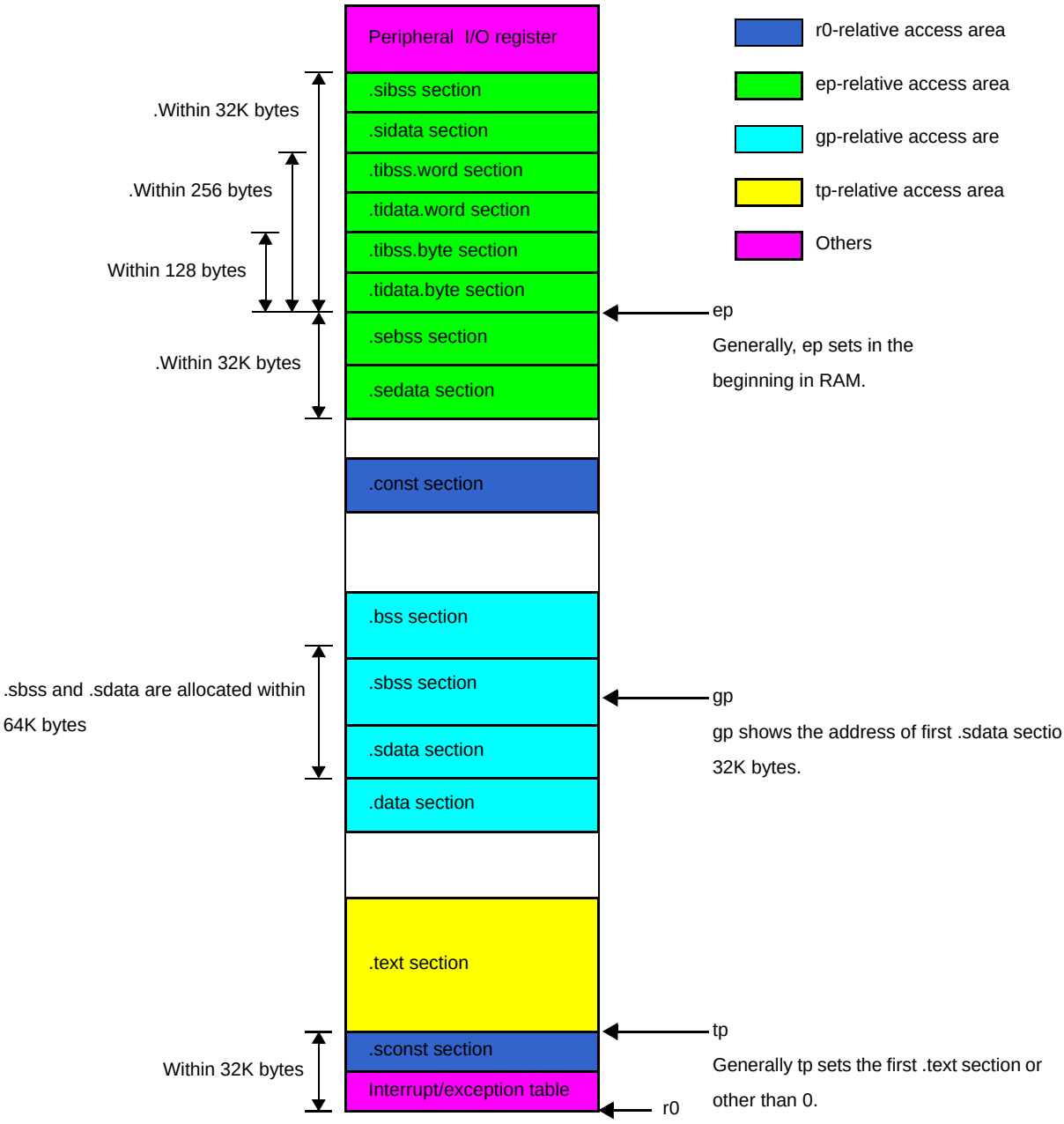
Cautions 1. "2 instructions" refer to the two instructions that are generated by assembler's instruction expansion function.

2. "gp relative" and "r0" relative indicate that the compiler will indicate gp-relative or r0-relative code.

3. Section types that are allocated to "external memory" can be used in cases where external memory has been mounted in the target system.

The following shows an image of memory allocation to various sections.

Figure 5-4. Example of Memory Allocation to Various Sections by CX (With Internal ROM)



5.2.5 Relationship between types and attributes of sections

The following describes the relation between types and attributes of sections.
These types and attributes are needed when coding section information in mapping directives.
The section types are categorized as shown below.

Table 5-2. Section Types

Section Type	Meaning
PROGBITS	Section that has actual values in an object module file --> Text or data (variable) with initial value
NOBITS	Section that does not have actual values in an object module file --> Data (variable) without initial value

The section attributes are categorized as shown below.

Table 5-3. Section Attributes

Section Attribute	Meaning
A	Section that occupies a memory area (corresponds to entire section): memory-resident section
W	Write-enable section (section allocated in RAM)
X	Executable section (mainly text section)
G	Section that is allocated within a memory area that can be referenced using a global pointer (gp) with 16-bit displacement (.sdata and .sbss section)

Sections are categorized into the following six groups according to their types and attributes.

Table 5-4. Classification of Sections

Section Attribute	Section Type/Section Attribute		Corresponding Reserved Section
bss attribute	Section type	NOBITS	.bss .sebss .sibss .tibss.byte .tibss.word
	Section attribute	AW	
const attribute	Section type	PROGBITS	.const .sconst
	Section attribute	A	
data attribute	Section type	PROGBITS	.data .sedata .sidata .tidata.byte .tidata.word
	Section attribute	AW	
sbss attribute	Section type	NOBITS	.sbss
	Section attribute	AWG	
sdata attribute	Section type	PROGBITS	.sdata
	Section attribute	AWG	

Section Attribute	Section Type/Section Attribute		Corresponding Reserved Section
text attribute	Section type	PROGBITS	.pro_epi_runtime .text
	Section attribute	AX	

Caution In cases where a specific section name is created within the application, the user must check the attribute for that section as shown in "Table 5-4. Classification of Sections", and specify the section type and section attribute in the mapping directive.
Section names that start with "V/H/A" which is followed by numeric characters cannot be created due to link directive format restrictions.

5.3 Symbols

The CX uses the following pointers for operation of applications.

- Text pointer (tp)
- Global pointer (gp)
- Element pointer (ep)

Each pointer value relates to the position of a segment and a means to determine these pointer values is required in the link directive.

A link directive contains symbol definitions that are used to determine pointer values. A defined symbol's value is determined by the linker and that value is copied to the pointer in the application to determine the pointer value. A link directive is sometimes called a "symbol directive" because it defines symbols used for pointers.

This section describes the role of each pointer and how pointer values are determined.

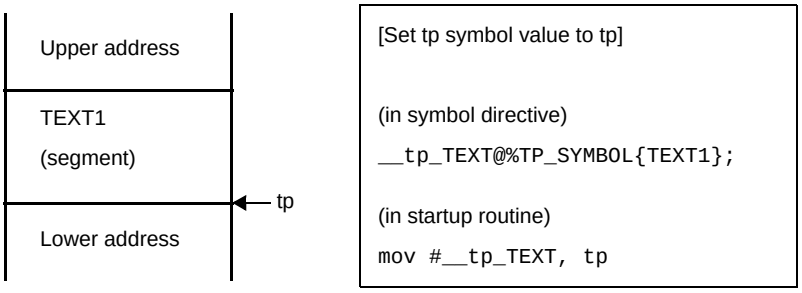
5.3.1 Text pointer (tp)

When referencing a text area in an application, the text pointer (tp) is provided to enable access independent of the allocation position (PIC: Position Independent Code). In other words, the text is referenced with tp-relative. Since the compiler outputs the code on the assumption that the tp has correctly set to the start of the text, the pointer value must be correctly.

In addition to creating a single tp for an application, several tps can be created for various segments.

When several tps have been created, however, the switching of tps must be explicitly performed by the application.

Figure 5-5. Example of tp Setting



In the above example, the link directive is used to set so that the tp symbol value specifies the start of TEXT1 segment. Since the tp symbol name is "__tp_TEXT", the start address of TEXT1 segment which is determined when linking is set to the symbol "__tp_TEXT".

To set this value to the tp, a startup routine (or other means) includes code (format: mov #__tp_TEXT, tp) that assigns the value of "__tp_TEXT" to the variable "tp". This correctly sets the text pointer value to the tp.

5.3.2 Global pointer (gp)

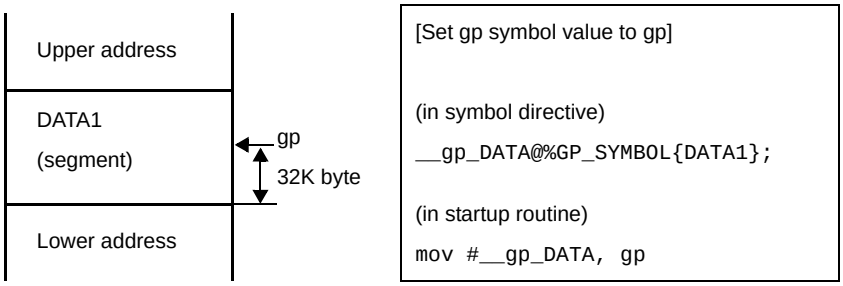
Data that is globally declared in an application is allocated to memory. When referencing (loading or storing) this data that has been allocated to memory, the global pointer (gp) is provided to enable access independent of the allocation position (PID: Position Independent Data).

Globally declared data is referenced with gp-relative. In V850 core devices, such data can be referenced using either "gp and one instruction" or "gp and two instructions". Compared to the "gp and two instructions" method, the "gp and one instruction" method speeds up applications and reduces the code size.

The sections that can be referenced using the gp and one instruction (ld/st instruction) method are the sections that have either the sdata attribute or the sbss attribute, while those that can be referenced using the gp and two instructions (movhi+ld/st instruction) method are the sections that have either the data attribute or the bss attribute. This means there are a total four attributes of sections that can be referenced with the gp-relative. The sections that have either the sdata attribute or sbss attribute are allocated within 32K-byte higher and lower the gp position, so that data (variables) allocated this range can be accessed using only one instruction, which is high-speed access with more reduced code size.

In addition to creating a single gp for an application, several gps can be created for various segments. When several gps have been created, however, the switching of gps must be explicitly performed by the application program.

Figure 5-6. Example of gp Setting (When Specifying Segment)



In the above example, the link directive is used to set so that the gp symbol value references the DATA1 segment. Since the gp symbol name is "__gp_DATA ", the address that is 32K-byte away from the start of the DATA1 segment which is determined when linking is set to the symbol "__gp_DATA" (see "Figure 5-6. Example of gp Setting (When Specifying Segment)").

To set this value to the gp, a startup routine (or other means) includes code (format: mov #__gp_DATA, gp) that assigns the value of "__gp_DATA" to the variable "gp". This correctly sets the global pointer value to the gp.

In addition to address, a gp symbol can also be specified by using an offset address value from tp symbol.

Offset specification for gp symbol values is described next.

(1) Offset specification for gp symbol values

As was described in the above, a typical method for specifying gp symbol values is the method that specifies the target segment for gp referencing.

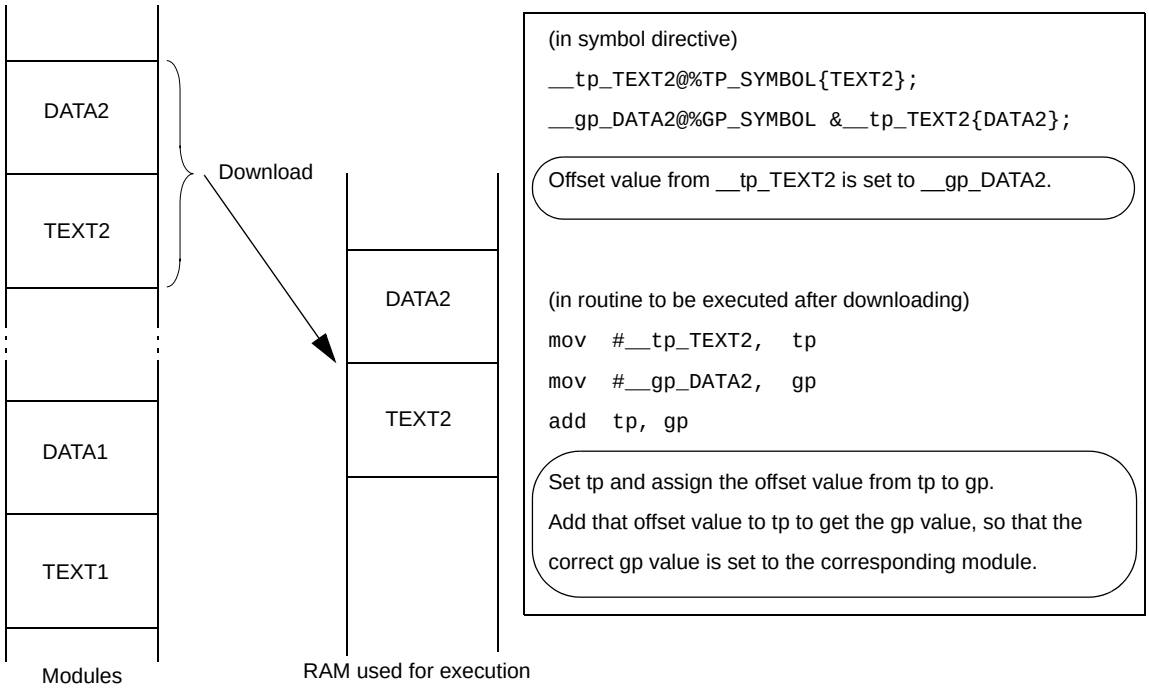
Other methods include directly specifying the gp symbol's address, and determining the base symbol and assigning a gp symbol value that is offset from the base symbol. The latter method is described below (for the former method, see "(2) Rules for determining gp symbol values").

A tp symbol is specified as the base symbol for a gp symbol.

When creating a gp symbol, if a tp symbol is specified as a base symbol, the value determined by the link directive as the gp's symbol value is the offset value from the tp symbol value.

In this way, the gp symbol value can be easily calculated based on the tp symbol value as "tp symbol value + offset value from tp symbol", which is useful for creating position-independent applications. For example, this method is helpful for copying an executable module to RAM (and then executing it) from an application that has multiple executable modules. In such cases, when determining the tp and gp values, once the tp value is known, the gp symbol value is simply added to that address (as the offset value from tp) to determine the gp value.

Figure 5-7. Example of gp Setting (When Specifying Offset from tp)



(2) Rules for determining gp symbol values

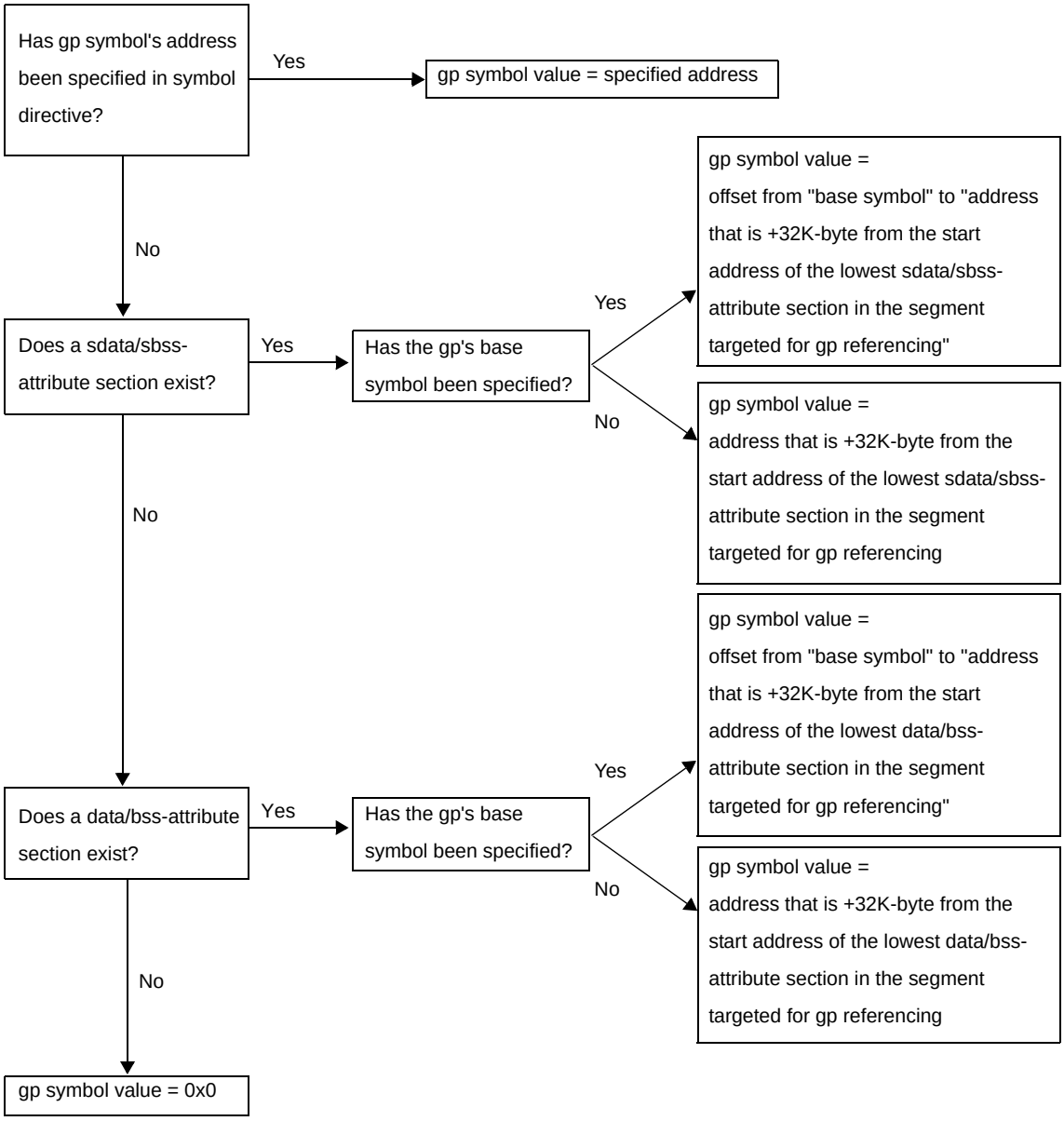
The following factors are involved in determining gp symbol value.

- Whether or not an address has been specified in the symbol directive.
- Whether or not sdata/sbss/data/bss-attribute sections exist.
- Whether or not a base symbol has been specified.

The linker checks for these factors in the link directive file and determines the gp symbol value.

The following figure illustrates the rules for determining gp symbol values.

Figure 5-8. Rules for Determining Global Pointer Values



5.3.3 Element pointer (ep)

The element pointer is a pointer that is provided to realize faster access (loading and storing) by allocating data (variables) that are globally declared within an application to RAM area in V850 core device.

Data (variables) that is globally declared and allocated to internal RAM area is referenced with ep-relative.

Although this reference uses the "ep and one instruction" combination, the attributes of sections are determined based on whether the one instruction is an sld/sst instruction or an ld/st instruction.

- The sections that can be referenced by "ep + sld/sst instruction" are:
  tidata.byte attribute, tibss.byte attribute, tidata.word attribute, or tibss.word attribute
- The sections that can be referenced by "ep + ld/st instruction" are:
  sidata attribute, sibss attribute, sedata attribute, or sebss attribute

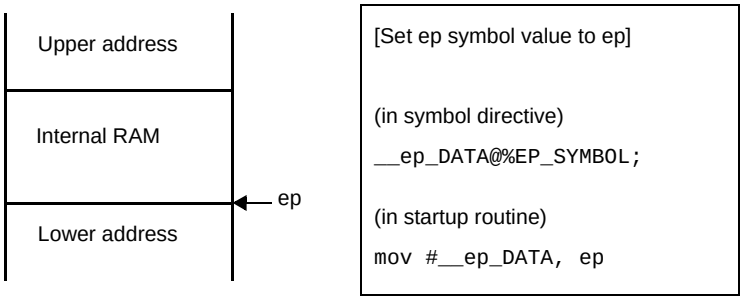
However, the sections with sedata/sebss attribute are not within internal RAM but within external RAM that is accessible via ep-relative referencing.

Generally, internal RAM capacity is too limited to store large amounts of data (variables), but storing certain data (variables) for which high-speed access is desired within the above area where "ep and one instruction" access is possible can be expected to improve the speed of the applications and reduce the code size. The sld/sst instruction is especially useful for reducing code size since its instruction length is two bytes compared to the ld/st instruction's four bytes.

If a creation of ep symbol has been specified in the link directive file's symbol directive, the linker automatically sets the ep symbol at the start of the internal RAM area according to the device file information that is provided for each device being used.

Note that only one ep symbol can be created within an application:it is not possible to create several per application.

Figure 5-9. Example of ep Setting



In the above example, the link directive is used to declare the creation of an ep symbol. Since the ep symbol name is "__ep_DATA", the linker sets the start address of internal RAM to "__ep_DATA".

To set this value to the ep, a startup routine (or other means) includes code (format: mov #__ep_DATA, ep) that assigns the value of "__ep_DATA" to the variable "ep". This correctly sets the element pointer value to the ep.

Remark The application's RAM usage can be set completely within internal RAM (not at all in external RAM), by creating only the ep symbol and not creating any gp symbols. However, if the runtime library will be used, gp symbols must be created since runtime functions reference data (variables) with gp-relative.

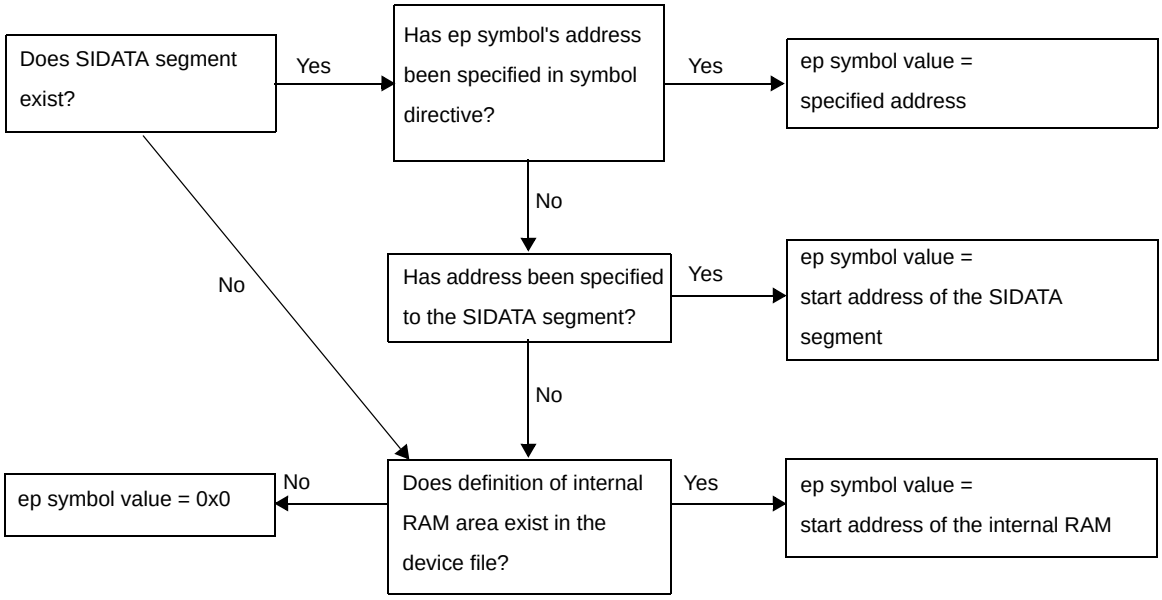
(1) Rules for determining ep symbol values

The following factors are involved in determining ep symbol value.

- Whether or not an address has been specified in the symbol directive.
- Whether or not SIDATA segment exist.
- Whether or not an internal RAM area has been difined in the device file.

The linker checks for these factors and determines the ep symbol value.
The following figure illustrates the rules for determining ep symbol values.

Figure 5-10. Rules for Determining Element Pointer Values



5.4 Coding Method

This section describes the format of the link directive file for each following item:

- Segment directive
- Mapping directive
- Symbol directive

The following is an outline of the link directive's format. An editor can be used to enter these directives in text format.

```
Segment directive1 {
        Mapping directive;
};
Segment directive2 {
        Mapping directive;
};
Segment directive3 {
        Mapping directive;
};
Segment directive4 {
        Mapping directive;
};
tp symbol directive;
gp symbol directive;
ep symbol directive;
```

Remark It is recommended to describe segment directive starting from the lowest address.

5.4.1 Characters used in link directive file

The following characters can be used in the link directive file.

- Numerals (0 to 9)
- Uppercase characters (A to Z)
- Lowercase characters (a to z)
- Underscore (_)
- Dot (.)
- Forward slash (/)
- Back slash (\)
- Colon (:) (can be used only for file name)
- Shift-JIS code (can be used only for file name; available only in the Japanese system)
- One-byte Japanese character (can be used only for file name; available only in the Japanese system)
- "#" (for comments)

"#" in the link directive file indicates the start of a comment. Text that starts with "#" and ends at end of the line is handled as a comment.

5.4.2 Link directive file name

Any file name can be assigned to a link directive file as long as the characters used are all valid characters for the link directive file. Note, however, that an extension is necessary. "dir" is recommended. When using the CubeSuite, please be sure to make it "dir" or "dr". Also note with caution that if an especially long file name is used, it may exceed the number of characters that can be handled during linkage (depending on the OS), which would preclude successful linkage.

If linkage is performed via command line entry, specify a link directive file with the -Xlink_directive option.

5.4.3 Segment directive

This section describes the format of the segment directive for each following item:

- Specification item
- Segment directive specification example

(1) Specification item

The items that are specified in the segment directive are listed below.

Table 5-5. Item Specified in Segment Directive

Item	Cording Format	Meanings	Omissible
Segment Name	Segmen-Name	Name of segment to be created	No
Segment type	!LOAD	Type (fixed) loaded to memory	No (part)[Note]
Segment attribute	?[R][W][X]	Specifies whether the segment to be created will have "read-enabled(R)" attribute, "write-enabled(W)" attribute, and/or "executable(X)" attribute (several can be specified)	No (part)[Note]
Address	V*address*	Start address of segment to be created	Yes
Maximum memory size	L*maximum-memory-size*	Upper limit of memory area occupied by segment to be created	Yes
Hole size	H*hole-size*	Size of hole to be created after segment (blank space between segment and next segment)	Yes

Item	Cording Format	Meanings	Omissible
Fill value	F*fill-value*	Value used to fill hole area	Yes
Alignment condition	A*alignment-condition*	Alignment condition for memory allocation	Yes

Note Some segment types and segment attributes cannot be omitted.
If only the reserved section is specified as the output section of mapping directives in that segment, then the type and attribute can be omitted.
If the output sections include an independently created section, then we recommend not omitting the type or attribute. If the type and attribute are omitted, then it will be interpreted as "!LOAD ?RW" having been specified.

A specific example of the segment directive's format is shown below.

```
Segment-Name: !segment-type ?segment-attribute Vaddress Lmaximum-memory-size Hhole-
size Ffill-value Aalignment-condition {
          :
        (Mapping directive)
          :
};
```

A blank space is used to separate these items from each other. A semicolon (;) must be added at the end of each segment directive.
The omissible specification items are "V*address*", "L*maximum memory size*", "H*hole size*", "F*fill value*", and "A*alignment condition*". Default values are used for these items when they are omitted. These default values are listed below.

Table 5-6. Default Values for Omitted Segment Directive Specification Items

Item	Default Value
Address	Address 0x0 for first segment, and the value continued from the end of the previous segment for other segments
Maximum memory size	0x100000 (bytes), it is a memory size to be allocated to segment, when device for which memory size allocated to segment exceeds 1M, is specified.
Hole size	0x0 (bytes)
Fill value	0x0000
Alignment Condition	0x8 (bytes)

Caution Describe segment directive starting from the lowest address.

(a) Segment name

Specify the name of the segment to be created.
When creating a segment, specification of the segment name cannot be omitted.
There is no restriction on the length of the character string that is to be specified as segment name. However, the name of segments which assign reserved sections listed in following table are fixed. Names other than those listed cannot be used for these segments.

Table 5-7. Reserved Section Names with Fixed Segment Names

Section Name	Segment Name
.sidata .sibss .tidata .tibss .tidata.byte .tibss.byte .tidata.word .tibss.word	SIDATA
.sedata .sebss	SEDATA
.sconst	SCONST

Remark The name of the segment for .sconst can be changed, but an error check is not performed to some of the data.

(b) Segment type

Specify the type of the segment to be created.
When creating a segment, some of the segment type specifications cannot be omitted.
If only the reserved section is specified as the output section of mapping directives in that segment, then the segment type can be omitted.
If the output sections include an independently created section, then we recommend not omitting the section type. If the section type are omitted, then it will be interpreted as "!LOAD" having been specified.
At present, only "LOAD" type (segment type that is loaded to memory) can be specified. The linker outputs an error message if another value is specified. The "LOAD" can be specified using either uppercase or lowercase letters.
Start the segment type specification with a "!", which must not be followed by blank space.

(c) Segment attributes

Specify the name of the segment to be created.
When creating a segment, some of the segment attribute specifications cannot be omitted.
If only the reserved section is specified as the output section of mapping directives in that segment, then the segment attribute can be omitted.
If the output sections include an independently created section, then we recommend not omitting the segment attribute. If the segment attribute are omitted, then it will be interpreted as "?RW" having been specified.
The specifiable segment attributes and their meanings are listed below.
A segment attribute depends on an attribute of mapping directive belonging to the segment. Therefore, the segment attribute specification must take into account the section attribute to be specified in the mapping directive.

Table 5-8. Segment Attributes and Their Meanings

Segment Attribute	Meanings
R	Read-enabled segment
W	Write-enabled segment
X	Executable segment

Several segment attributes can be specified at the same time, with R, W, and X specified in any order with no blank spaces between them. Start each section attribute specification with a "?", which must not be followed by a blank space.

Remark If multiple segment attribute specifications are performed in one segment directive, the linker outputs an error message and stops linking.

Example

```
SEG:    !LOAD   ?RX ?RW {};
```

(d) Address

Specify the start address of the section to be created.
When creating a segment, specification of the address can be omitted. When it is omitted, the address 0x0 is assigned as the start address if the segment is the first segment, otherwise the assigned value for the start address is the value continued from the end of the previous segment (based on the alignment).
Address specifications must be made with consideration given to the way memory is allocated in the target CPU.
For example, if the target CPU is a V850E core device, since different memory capacities are installed in the various V850 core devices, their internal ROM/RAM uses different start and end addresses. Consequently, the allocation address specification for each segment must take into account which CPU is being used. For description of a particular CPU's memory, see the CPU's User's Manual (Hardware Version) and/or the corresponding device file's User's Manual.
Specify even-numbered values as the address values. If an odd- numbered value is specified, the linker outputs a message and continues with linking on the assumption that the "specified address plus one" has been specified.
Start the address specification with a "V" (uppercase or lowercase), which must not be followed by a blank space. Address values can be specified using either decimal or hexadecimal numerals, but when using hexadecimal numerals be sure to add "0x" before the value. Expressions cannot be used in the address specification.

Remark By default, the "DATA_CMN" is aligned at the start of the PE1 area of the RAM area.
Use an address specification ("V") if you wish to specify a different location.

(e) Maximum memory size

Specify the maximum value for memory size of the segment to be created.
This specification is used not to exceed the segment's intended size. Therefore, if the segment's actual size is less than the specified "maximum memory size", the next segment will follow immediately afterward.
When creating a segment, specification of the maximum memory size can be omitted. The value 0x100000 (bytes) is used as the default value when it is omitted.
When created segment exceeds the value specified by maximum memory size, linker outputs an error message and stops linking.
Start the maximum memory size specification with a "L" (uppercase or lowercase), which must not be followed by a blank space. Expressions cannot be used in the maximum memory size specification.

(f) Hole size

Specify the hole size of the segment to be created.
The segment's hole is the space between one segment and the next segment. When a hole size has been specified, the specified hole is created at the end of the target segment.

When creating a segment, specification of the hole size can be omitted. The value 0x0 (bytes) is used as the default value (which specifies that no hole is created) when it is omitted.
Start the hole size specification with an "H" (uppercase or lowercase), which must not be followed by a blank space.
Expressions cannot be used in the hole size specification.

(g) Fill value

Specify a fill value as the value to be used for filling hole areas that are created either when segments are allocated or when explicitly specified via the "H" specification.
When specifying the fill value, specify the -Xtwo_pass_link option to perform linking in the 2-pass mode. If the linkage is performed with the fill value specification in the 1-pass mode (default), the linker outputs a message and continues ignoring this specification and linking.
When creating a segment, specification of the fill value can be omitted. The value 0x0000 is used as the default value (which fills hole areas with zeros) when it is omitted. However, if the -Xalign_fill option (linker fill value option) has been specified, the linker outputs a message and continues linking while ignoring the fill value specified by the link directive.
Start the fill value specification with an "F" (uppercase or lowercase), which must not be followed by a blank space. Specify a 2-byte four-digit hexadecimal value as the fill value. If the value dose not occupy all four digits, the remaining (higher) digits are assumed to be zeros. If the hole size is less than two bytes, the required digits are taken out of the lower value of the specified fill value. Expressions cannot be used in the fill value specification.

(h) Alignment condition

Specify the segment alignment condition (alignment value) to be used for memory allocation of the segment to be created.
When creating a segment, specification of the alignment condition can be omitted. The value 0x8 (bytes) is used as the default value (which sets 8-byte alignment) when it is omitted.
Start the alignment condition specification with an "A" (uppercase or lowercase), which must not be followed by a blank space. Specify even-numbered values as the alignment condition values. If an odd- numbered value is specified, the linker outputs a message and continues with linking on the assumption that the "specified address plus one" has been specified. Expressions cannot be used in the alignment condition specification.
If an address is specified, then the specified address is given precedence, and alignment-condition specifications will be ignored.

(2) Segment directive specification example

A segment specification example is shown below.

Table 5-9. Segment Example

Item	Value
Segment Name	PROG1
Segment type	Read-enabled, executable
Allocation address	address 0x1000
Maximum memory size	0x200000 (bytes)
Hole size	0x20 (bytes)
Fill value	0xFFFF
Alignment Condition	0x16 (bytes)

The segment directive code appears as shown below for above segment.

```
PROG1:  !LOAD   ?RX V0x1000 L0x200000   H0x20   F0xFFFF A0x16 {
          :
        (Mapping directive)
          :
};
```

Remark Basically, there is no problem if segment directives are described in the order of the allocation addresses.
The only exception applies to segments that have .sedata/.sebss section (by default, "SEDATA segment"), only when the allocation address is omitted.
In the CX, the SEDATA segment is defined as a segment used to reference the area below the internal RAM with 1 ep-relative instruction, and therefore, if the allocation address is omitted, the linker considers that the address obtained by subtracting 0x8000 from the internal RAM start address defined in the device file, has been specified.

The following is an example of this case.

```
SIDATA: !LOAD   ?RW V0xFFB000 {
        .tidata.byte    = $PROGBITS ?AW     .tidata.byte;
        .tibss.byte     = $NOBITS   ?AW     .tibss.byte;
        .tidata.word    = $PROGBITS ?AW     .tidata.word;
        .tibss.word     = $NOBITS   ?AW     .tibss.word;
        .sidata         = $PROGBITS ?AW     .sidata;
        .sibss          = $NOBITS   ?AW     .sibss;
};
SEDATA: !LOAD   ?RW {
        .sedata         = $PROGBITS ?AW     .sedata;
        .sebss          = $NOBITS   ?AW     .sebss;
};
DATA:   !LOAD   ?RW {
        .data           = $PROGBITS ?AW     .data;
        .sdata          = $PROGBITS ?AWG    .sdata;
        .sbss           = $NOBITS   ?AWG    .sbss;
        .bss            = $NOBITS   ?AW     .bss;
};
```

The SEDATA address is omitted and this start address is judged as 0xFF2000 (= 0xFFB00 - 0x8000) according to device file information. Since SIDATA is defined as being allocated to address 0xFFB000, the CX moves the SEDATA to the front of SIDATA and links them.
Moreover, since the address of the DATA segment defined after that is omitted, DATA is allocated immediately after the SEDATA.

5.4.4 Mapping directive

This section describes the format of the mapping directive for each following item:

- Specification item
- Mapping directive specification example

(1) Specification item

The items that are specified in the mapping directive are listed below.

Table 5-10. Item Specified in Segment Directive

Item	Cording Format	Meanings	Omissible
Output section name	Output-section-name	Name of section output to load module	No
Section type	$PROGBITS $NOBITS	Type of section to be created	No (part)Note
Section attribute	?[A][W][X][G]	Specifies whether the section to be created will have "memory-resident(A)" attribute, "write-enabled(W)" attribute, "executable(X)" attribute, and/or "accessible via gp with 16-bit displacement(G)" attribute (several can be specified).	No (part)Note
Input section name	Input-section-name	Name of input section allocated to output section	Yes
Address	V*address*	Start address of section to be created	Yes
Hole size	H*hole size*	Size of hole to be created after section (blank space between section and next section)	Yes
Alignment Condition	A*alignment condition*	Alignment condition for memory allocation	Yes
Object module file name	{*object-file-name object-file-name* ...}	Name of object module file that includes the sections to be extracted and used as the input sections (several can be specified; insert spaces between the specifications).	Yes

Note Some section types and section attributes cannot be omitted.
If only the reserved section is specified as the output section of mapping directives, then the type and attribute can be omitted.
If the output sections include an independently created section, then we recommend not omitting the type or attribute. If the type and attribute are omitted, then it will be interpreted as "$NOBITS ?AW" having been specified.

A specific example of the mapping directive's format is shown below.

```
Output-section-name =
        $Section-type
        ?Section-attribute
        Vaddress
        Hhole-size
        Aalignment-condition
        Input-section-name
        {object-file-name object-file-name};
```

A blank space is used to separate these items from each other. A semicolon (;) must be added at the end of each segment directive

The omissible specification items are "Vaddress", "Hhole size", "Aalignment condition", "input section name" and "object module file name". Default values or pre-set conventions are used for these items when they are omitted. These default values and pre-set conventions are listed below.

Table 5-11. Default Values/Conventions for Values That Can Be Omitted in Mapping Directive Specification Items

Item	Default Values/Conventions
Address	Sets according to address that was specified via the segment directive. If there are several sections and this is not the first one, the value is continued from the end of the previous section. If the section is the first section, the value is continued from the start of the segment.
Hole size	0x0 (bytes)
Alignment Condition	.tidata.byte /.tibss.byte section:0x1(bytes) Other sections: 0x4 (bytes)
Input section	Sections having the same attribute as the output section to be created are extracted from all objects. If an object module file name has been specified, they are extracted from the specified object.
Object module file name	Sections having the same attribute as the output section to be created are extracted from all objects. If an input section has been specified, they are extracted from all the objects that have the same attribute as the output section to be created.

These specification items are explained below.

(a) Output section name

Name of section output to load module When creating a section, specification of the output section type cannot be omitted.

There is no restriction on the length of the character string that is to be specified as output segment name. However, note the fixed correspondence of output section names and input section names listed in the following table and names other than those listed cannot be used for these sections.

Table 5-12. Reserved Section Names with Fixed Segment Names

Input Section Name	Output Section Name
.tidata section	.tidata
.tibss section	.tibss
.tidata.byte section	.tidata.byte
.tibss.byte section	.tibss.byte
.tidata.word section	.tidata.word
.tibss.word section	.tibss.word
.sidata section	.sidata
.sibss section	.sibss
.sedata section	.sedata

Input Section Name	Output Section Name
.sebss section	.sebss
.sconst section	.sconst
.pro_epi_runtime section	.pro_epi_runtime

Remark Although two or more mapping directives can be described in the same segment directive, two or more of the same output section names cannot be specified in different segment directive. If two or more of the same output section names are specified, the linker outputs an error message and stops linking.

(b) Section type

Specify the type of the output section.
When creating a section, some section types cannot be omitted.
If only the reserved section is specified as the output section, then the type can be omitted.
If the output sections include an independently created section, then we recommend not omitting the type. If the type are omitted, then it will be interpreted as "$NOBITS" having been specified.

The specifiable section types and their meanings are listed below.

Table 5-13. Section Types and Their Meanings

Section Type	Meanings
PROGBITS	Section that has actual values in an object module file --> Text or data (variable) with initial value
NOBITS	Section that does not have actual values in an object module file --> Data (variable) without initial value

Start the section type specification with a "$", which must not be followed by a blank space.
If only "$" is specified, the linker outputs an error message and stops linking.

(c) Section attributes

Specify the name of the section to be created.
When creating a section, some section attributes cannot be omitted.
If only the reserved section is specified as the output section, then the section attribute can be omitted.
If the output sections include an independently created section, then we recommend not omitting the section attribute. If the section attribute are omitted, then it will be interpreted as "?AW" having been specified.
The specifiable section attributes and their meanings are listed below.

Table 5-14. Section Attributes and Their Meanings

Section Attribute	Meanings
A	Section that occupies a memory area (corresponds to entire section)
W	Write-enable section (section allocated in RAM)
X	Executable section (mainly text section)
G	Section (.sdata,/.sbss section) that is allocated within a memory area that can be referred using a global pointer (gp) with 16-bit displacement

Several section attributes can be specified at the same time, with A, W, X, and G specified in any order with no blank spaces between them. Start each section attribute specification with a "?", which must not be followed by a blank space.
If a mapping directive is specified in a segment directive, then make sure that the specified section attribute matches the segment attribute specified in that segment directive. In other words, ignore section attribute G, and match section attributes A, W, and X with values corresponding to segment attributes R, W, and X.

Remark If a section attribute is specified more than once for the same mapping directive, then the linker will output an error, and linking will halt.

Example

```
sec     = $PROGBITS ?AX ?AW;
```

If a section with a writable attribute is allocated to internal ROM or internal instruction RAM, then a message is output, and linking continues.

(d) Input section name

Specify the input section information that is the basis for the output section to be created.
When creating a section, specifications of the input section name and object module file name can be omitted. If it is omitted, the information output to the output section varies according to the following combinations of specifications.

Table 5-15. Output Based on Combination of Input Section and Object Module File Specifications

Code Pattern		Output
(1)	Input section name + object module file name	The specified input section is extracted from the specified object and is then output.
(2)	Input section name only	The specified input section is extracted from all objects and are then output.
(3)	Object module file name only	Sections having the same attribute as the output section to be created are extracted from the specified object and are then output.
(4)	No specification	Sections having the same attribute as the output section to be created are extracted from all objects and are then output.

More specific examples are listed below.

Table 5-16. Specific Examples of Combined Input Section and Object Module File Specifications

Code Example	Output

`SEG1: !LOAD ?RX {`
`  sec1  = $PROGBITS ?AX usrsec1 {file1.obj};`
`}`
	"usrsec1" section is extracted form file1.obj and is output as "sec1" section.

`SEG1: !LOAD ?RX {`
`  sec1  = $PROGBITS ?AX usrsec1;`
`}`
	"usrsec1" section is extracted form all objects and is output as "sec1" section.

Code Example	Output

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX {file1.obj file2.obj};
}
	Sections having $PROGBITS type and A and X attributes are extracted from file1.obj and file2.obj and are output as "sec1" section.

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX;
}
	Sections having $PROGBITS type and A and X attributes are extracted from all objects and are output as "sec1" section.

If there is multiple information when allocating sections, sections are allocated using the numbers indicated in the [Code Pattern] column in "Table 5-15. Output Based on Combination of Input Section and Object Module File Specifications" as the priority order (in the case of two or more sections with the samepriority number, the one with the lowest address has higher priority).
Specify the section name that has been set by the application as the input section name. If the application has not set a section name, a default section name is already defined and should be used here.
As was explained in "(a) Output section name", there is a fixed correspondence between output section names and input section names. Other section names cannot be specified for section names that are included in this group.

(e) Address

Specify the start address of the section to be created.
When creating a section, specification of the address can be omitted. If it is omitted, the address is assigned based on the address specified via the segment directive. If there are several sections and this is not the first one, the value is continued from the end of the previous section.
Normally, section addresses are specified as a group for each segment, but separate address specifications can be made to assign certain addresses to certain sections.
Specify even-numbered values as the address values. If an odd- numbered value is specified, the linker outputs a message and continues with linking on the assumption that the "specified address plus one" has been specified.
Start the address specification with a "V" (uppercase or lowercase), which must not be followed by a blank space. Address values can be specified using either decimal or hexadecimal numerals, but when using hexadecimal numerals be sure to add "0x" before the value. Expressions cannot be used in the address specification.

(f) Hole size

Specify the hole size of the section to be created.
The section's hole is the space between one section and the next section. When a hole size has been specified, the specified hole is created at the end of the target section.
When creating a section, specification of the hole size can be omitted. The value 0x0 (bytes) is used as the default value (which specifies that no hole is created) when it is omitted.
Start the hole size specification with an "H" (uppercase or lowercase), which must not be followed by a blank space. Expressions cannot be used in the hole size specification.

(g) Alignment condition

Specify the section alignment condition (alignment value) to be used for memory allocation of the section to be created.

When creating a section, specification of the alignment condition can be omitted. If it is omitted, the default value is used, but that value differs among different types of section as shown below.

Table 5-17. Section Types and Default Values for Alignment Condition

Section Name	Alignment Condition
.tidata.byte/.tibss.byte section	0x1 (bytes)
TEXT attribute section except internal instruction RAM	0x2 (bytes)
Other sections	0x4 (bytes)

Start the alignment condition specification with an "A" (uppercase or lowercase), which must not be followed by a blank space.

Either even-numbered or odd-numbered values can be specified for .tidata.byte and .tibss.byte sections and only even-numbered values can be specified for all other sections. If an odd-numbered value is specified for any section other than a .tidata.byte or .tibss.byte section, the linker outputs a message and continues with linking on the assumption that the "specified value plus one" has been specified. Expressions cannot be used in the alignment condition specification.

Caution The alignment condition of sections allocated to internal instruction RAM is 4 (or a multiple thereof). If a value other than 4 (or a multiple thereof) is specified, then a message will be output, and linking will continue with the specified alignment condition.
4-byte access is needed when writing. In the case of internal instruction RAM that does not allow misaligned access, the alignment condition of the section to copy must be set to 4 when moving the ROMized section to internal instruction RAM.

(h) Object module file name

Enter the object module file name's specification at the end of the mapping directive and enclose each file name with "{ }". Insert a blank space between file names when specifying several file names (if the file name includes blank spaces, enclose the file name with quotation marks ("")).

When several object module files have been specified, they are allocated in the order they are specified, in ascending order from lower to higher addresses. However, if a different allocation order is specified for link directive by the "objects for linking" specification that occurs when the linker is started, the file name sequence specified be that specification's parameters takes priority.

```
Link directive
sec = $PROGBITS ?AX {filel.obj file2.obj file3.obj}
```

```
Linker activation
cx file3.obj filel.obj file2.obj
        --> file3.obj, file1.obj, and file2.obj are allocated in that order,
            starting from lower address
```

When an object module file name is specified in a mapping directive, specify all object module file names that include sections having the specified attribute.

For example, the four objects (file1.obj, file2.obj, file3.obj, and file4.obj) including text-attribute sections exist. In this case, if the link directive is entered as:

```
TEXT1:  !LOAD   ?RX {
        .text1  = $PROGBITS ?AX {filel.obj file2.obj};
};
TEXT2:  !LOAD   ?RX {
        .text2  = $PROGBITS ?AX {file3.obj};
};
```

and no specific allocation site for the text attribute in the file4.obj has been specified, the linker searches and allocates text-attribute sections from file4.obj as suitable text-attribute sections. Therefore, the mapping results may not be as expected (if the text-attribute section is not allocated to any section, the linker outputs a message).
Specify a file of the same name located in a different directory as follows by specifying a file name with the path displayed on the link map.

```
textsec1    = $PROGBITS ?AX {c:\work\dir1\file1.obj};
textsec2    = $PROGBITS ?AX {c:\work\dir2\file1.obj};
textsec3    = $PROGBITS ?AX {file1.obj};
```

In the above case, the file1.obj files that exist in the specified directories are allocated to textsec1 and textsec2 respectively, and the other file is allocated to textsec3. Since the path specification method during such allocation is only the format displayed to the link map, attention is required when making descriptions.
It is also possible to specify input object names for objects in library files. For example, the following is entered to specify output of object "lib1.obj" in the archive file "libusr.lib" to the "usrlib" section.

```
usrlib  = $PROGBITS ?AX {lib1.obj(a:\usrlib\libusr.lib)};
```

Moreover, describe as follows to allocate all the objects in the specified library.

```
usrlib  = $PROGBITS ?AX {libusr.lib};
```

In this case, the object in "libusr.lib" is allocated to "usrlib" section.

The specification of the object module file name can be omitted.

Example

```
sec     = $PROGBITS ?AX .text;
```

If the file name is omitted, the CX linker assumes that all object module files not otherwise specified have been specified.
The example below shows object module files "file1.obj", "file2.obj", "file3.obj", and "file4.obj" are specified and launched.

```
sec1     = $PROGBITS ?AX .text;
sec2     = $PROGBITS ?AX .text {file1.obj};
```

same as the following example:

```
sec1     = $PROGBITS ?AX .text {file2.obj file3.obj file4.obj};
sec2     = $PROGBITS ?AX .text {file1.obj};
```

(i) If specification duplicates

If the same section type, section attribute, input section name (can be omitted), or input file name (can be typed) is specified for multiple segments and there is a section corresponding to it, an object is assigned to a segment allocated at a lower address.

```
TEXT1:  !LOAD   ?RX V0x1000 {
        .text1  = $PROGBITS ?AX .text   {filel.obj file2.obj};
};
TEXT2:  !LOAD   ?RX V0x2000 {
        .text2  = $PROGBITS ?AX .text   {filel.obj file2.obj};
};
```

In the above case, the same section type, section attribute, input section name, and input file name are specified for TEXT1 and TEXT2, the object is assigned to TEXT1, which is allocated at the lower address.

(2) Mapping directive specification example

This example shows specifications for the following types of output sections. Two type of sections are created.

Table 5-18. Mapping Directive Specification Example

Item	Value-1	Value-2
Output section name	.text	textsec1
Section type	Text	Text
Section attribute	Read-enabled, executable	Read-enabled, executable
Hole size	0x10 (bytes)	0x20 (bytes)
Fill value	0xFFFF	0xFFFF
Alignment condition	0x10 (bytes)	0x10 (bytes)
Input section name	.text	usrsec1
Object module file name	main.obj	-

In the above case, the corresponding mapping directive specification is shown below.

```
.text       = $PROGBITS ?AX H0x10   F0xFFFF A0x10   .text   {main.obj};
textsec1    = $PROGBITS ?AX H0x20   F0xFFFF A0x10   usrsec1;
```

5.4.5 Symbol directive

This section describes the format of the symbol directive for each following item:

- Specification item
- Symbol directive specification example

(1) Specification item

The items that are specified in the symbol directive are listed below.

- tp symbol

Table 5-19. Specifiable Items When Creating tp Symbol

Item	Cording Format	Meanings	Omissible
Symbol name	Symbol-name	Name of tp symbol to be created	No
Symbol type	%TP_SYMBOL	Type of symbol to be created (fixed)	No
Address	V*address*	Address of tp symbol to be created	Yes
Alignment Condition	A*alignment-condition*	Alignment condition of symbol value	Yes
Segment Name	{*segment-name segment-name* ...}	Name of segment to be referred by tp symbol to be created (several can be specified; insert blank spaces between the specifications.)	Yes

A specific example of the symbol directive's format is shown below.

```
symbol-name@%TP_SYMBOL Vaddress Aalignment-condition {segment-name segment-name};
```

A blank space is used to separate these items from each other. A semicolon (;) must be added at the end of each segment directive.
The omissible specification items are "Vaddress", "Aalignment condition", and "segment name". Default values are used for these items when they are omitted. These default values are listed below.

Table 5-20. Default Values for tp Symbols

Item	Default Values
Address	If a segment name has been specified, this address is the start address of the text- attribute section that has been allocated to the lowest address in that segment. If a segment name has not been specified, this address is the start address of the text- attribute section that has been allocated to the lowest address in the text-attribute segment existing in the load module.
Alignment Condition	0x4 (bytes)
Segment Name	All text-attribute segments exist in objects are targeted.

- gp symbol

Table 5-21. Specifiable Items When Creating gp Symbol

Item	Cording Format	Meanings	Omissible
Symbol name	Symbol-name	Name of gp symbol to be created	No
Symbol type	%GP_SYMBOL	Type of symbol to be created (fixed)	No
Base symbol name	&*base-symbol-name*	tp symbol name which becomes the base symbol when specifying a gp symbol as offset value	Yes
Address	V*address*	Address of gp symbol to be created	Yes
Alignment Condition	A*alignment-condition*	Alignment condition of symbol value	Yes
Segment Name	{*segment-name segment-name* ...}	Name of segment to be referred by gp symbol to be created (several can be specified; insert blank spaces between the specifications.)	Yes

A specific example of the symbol directive's format is shown below.

```
symbol-name @%GP_SYMBOL  &base-symbol-name  Vaddress  Aalignment-condition {segment-name segment-name};
```

A blank space is used to separate these items from each other. A semicolon (;) must be added at the end of each segment directive.
The omissible specification items are "Vaddress", "Aalignment condition", and "segment name". Default values are used for these items when they are omitted. These default values are listed below.

Table 5-22. Default Values for gp Symbols

Item	Default Values
Base symbol name	Address to be determined as the gp symbol value, not for offset from tp symbol
Address	Linker can determine gp symbol value from items below. - Existing sections with sdata /sbss /data /bss attributes - Existing base symbol specifications
Alignment Condition	0x4 (bytes)
Segment Name	All sections with sdata/data/sbss/bss attributes existing in objects are targeted.

- ep symbol

Table 5-23. Specifiable Items When Creating ep Symbol

Item	Cording Format	Meanings	Omissible
Symbol name	Symbol-name	Name of ep symbol to be created	No
Symbol type	%EP_SYMBOL	Type of symbol to be created (fixed)	No
Address	V*address*	Address of ep symbol to be created	Yes
Alignment Condition	A*alignment-condition*	Alignment condition of symbol value	Yes

A specific example of the symbol directive's format is shown below.

```
symbol-name @%EP_SYMBOL Vaddress Aalignment-condition;
```

A blank space is used to separate these items from each other. A semicolon (;) must be added at the end of each specification.
The omissible specification items are "Vaddress" and "Aalignment condition". Default values are used for these items when they are omitted. These default values are listed below.

Table 5-24. Default Values for ep Symbols

Item	Default Values
Address	Linker can determine ep symbol value from items below. - Existing SIDATA segment - Definitions of existing internal RAM area in device file
Alignment Condition	0x4 (bytes)

These specification items are explained below.

(a) Symbol name [Specifiable symbols: tp, gp, ep]

Specify the name of the symbol to be created. When creating a symbol, specification of the symbol name cannot be omitted.
There is no restriction on the length of the character string that is to be specified as symbol name.

(b) Symbol type [Specifiable symbols: tp, gp, ep]

Specify whether the generated symbol will be a tp symbol, gp symbol, or ep symbol. When creating a symbol, specification of the symbol type cannot be omitted.
Specify "TP_SYMBOL", "GP_SYMBOL", or "EP_SYMBOL" corresponding to the desired type of symbol (tp symbol, gp symbol, or ep symbol). The linker outputs an error message if another value is specified.
Start the symbol type specification with a "%", which must not be followed by a blank space.

(c) Base symbol name [Specifiable symbol: gp]

Specify the tp symbol that will be used to determine the gp symbol value when creating gp symbols. When a base symbol name has been specified, the gp symbol value becomes the offset value from the tp symbol value.
When creating a gp symbol, specification of the base symbol name can be omitted.
Start the base symbol specification with a "&", which must not be followed by blank space. After the "&", enter the tp symbol name to be used as the base symbol.

(d) Address [Specifiable symbols: tp, gp, ep]

Specify the tp symbol value or gp symbol value (these values are addresses).
When creating a symbol, specification of the address can be omitted. If it is omitted, the address is determined as described below.

Table 5-25. Address Specification for tp Symbol, gp Symboland and ep Symbol

Symbol Value	Rule for Determination
tp symbol	- If a segment name has been specified, this address is the start address of the text- attribute section that has been allocated to the lowest address in that segment. - If a segment name has not been specified, this address is the start address of the text- attribute section that has been allocated to the lowest address in the text-attribute segment existing in the load module.
gp symbol	Linker can determine gp symbol value from items below. - Existing sections with sdata /sbss /data /bss attributes - Existing base symbol specifications
ep symbol	Linker can determine ep symbol value from items below. - Existing SIDATA segment - Definitions of existing internal RAM area in device file

Start the address specification with a "V" (uppercase or lowercase), which must not be followed by a blank space.

(e) Alignment condition [Specifiable symbols: tp, gp, ep]

Specify the alignment condition (alignment value) for setting values to the tp symbol, gp symbol, or ep symbol to be created.
When creating a symbol, specification of the alignment condition can be omitted. Default values are used for these items when they are omitted. This default value is 0x4 (bytes).
Start the alignment condition specification with an "A" (uppercase or lowercase), which must not be followed by a blank space. Specify even-numbered values as the alignment condition values. If an odd- numbered value is specified, the linker outputs a message and continues with linking on the assumption that the "specified address plus one" has been specified. Expressions cannot be used in the alignment condition specification.

(f) Segment name [Specifiable symbols: tp, gp]

Specify the name of the segment to be referred for the tp symbol value or gp symbol value to be created.
In other words, specify the segment that will be referenced by the tp symbol or gp symbol to be created.
Several segments can be specified as target segments for referencing.
When creating a symbol, specification of the segment name can be omitted. One of the following values is assumed as the default value when it is omitted.

Table 5-26. Segment Names Targeted for Reference by tp Symbol and gp Symbol

Symbol Value	Rule for Determination
tp symbol	All text-attribute segments exist in objects are targeted.
gp symbol	All sections with sdata/data/sbss/bss attributes existing in objects are targeted.

Specify a segment name that is assumed to be a target for gp-relative referencing as the target segment name for gp symbol referencing.
For example, do not specify a segment that includes .sedata section or .sebss section, which is assumed to be for ep-relative referencing.

Note, however, that when this is omitted, if multiple segments are allocated to an sdata/sbss attribute section, then the segment allocated to the lowest address is given precedence, and the gp symbol value could have an unintended value. If a non-gp relative section is created, be sure to specify an appropriate segment for the segment name corresponding to gp symbols.
Enter the segment name specification at the end of the symbol directive and enclose the segment name with "{}". If specifying several segment names, use blank spaces to separate them.

(2) Symbol directive specification example

This example shows specifications for the following types of symbols.

Table 5-27. Symbol Directive Specification Example

Symbol	Specification Item	Specified Value
tp symbol	Symbol name	__tp_TEXT
	Name of segment targeted for reference	TEXT1
gp symbol	Symbol name	__gp_DATA
	Offset specification symbol	__tp_TEXT
	Name of segment targeted for reference	DATA1, DATA2
ep symbol	Symbol name	__ep_DATA
	Address	0xFFFFD000

In the above case, the corresponding symbol directive specification is shown below.

```
__tp_TEXT@%TP_SYMBOL    {TEXT1};
__gp_DATA@%GP_SYMBOL    &__tp_TEXT  {DATA1 DATA2};
__ep_DATA@%EP_SYMBOL    V0xFFFFD000;
```

Note with caution that symbols will not be created unless a symbol directive specification has been made.

5.5 Reserved Words

The link directive file has reserved words. Reserved words cannot be used in the other specified usage.
The reserved words are as follows.

- Segment name (SIDATA, SEDATA, SCONST)
- Segment type (LOAD)
- Output section name (.tidata, .tibss etc)
- Section type (PROGBITS, NOBITS)
- Symbol type (TP_SYMBOL, GP_SYMBOL, EP_SYMBOL)

CHAPTER 6 FUNCTIONAL SPECIFICATIONS

This chapter describes the library functions provided in the CX.

6.1 Supplied Libraries

The CX provides the following libraries.

Table 6-1. Supplied Libraries

Supplied Libraries	Library Name	Outline
Standard library	libc.lib libc22.lib libc26.lib libc32.lib libccn.lib	Function with variable arguments Character string functions Memory management functions Character conversion functions Character classification functions Standard I/O functions Standard utility functions Non-local jump functions
Mathematical library		Mathematical functions
Initialization library		Initialization peripheral devices function
ROMization library		Copy functions
Multi-core library		Pseudo "main" functions for multi-core
Runtime library		Operation runtime functions Function pre/post processing runtime functions
Libraries used in V850E2V3-FPU	libf32.lib libf64.lib	Functions used in V850E2V3-FPU
Data position independent library	libp.lib	
Data position dependent library.	libnp.lib	

When the standard library or mathematical library is used in an application, include the related header files to use the library function.

Refer these libraries using the linker option (-l).

However, it is not necessary to refer the libraries if only "function with a variable arguments", "character conversion functions" and "character classification functions" are used.

When CubeSuite is used, these libraries are referred by default.

The operation runtime function is a routine that is automatically called by the CX when a floating-point operation or integer operation is performed. Function pre/post processing runtime function is a routine that is automatically called by the process of the CX prologue/epilogue functions.

Unlike the other library functions, the "operation runtime function" and "function pre/post processing runtime function" is not described in the C source or assembler source.

The ROMization library is referred by the linker. This library stores the functions (_rcopy, _rcopy1, _rcopy2, _rcopy4), which are used to copy packed data.

Description of each library is as follows.

The meaning of each element in the table is as follows.

Function/macro name	Name of function/macro.
Outline	Functional outline of function/macro.
#include	Header file that must be included in the C source when this function/macro is used. Include this file using the #include directive. "errno.h" must also be included if errno is used when an exception occurs.
ANSI	Indicates whether or not the function is differentiated by the ANSI standard. If it is stipulated, "YES" is shown in this column; if not, "--" is shown.
const	Differentiates whether or not this function/macro uses the memory area "const area". If the .const section is used, "YES" is shown in this column; if not, "--" is shown.
sdata	Differentiates whether or not this function/macro uses the memory area "sdata area". In other words, whether or not data for which the function has an initial value is allocated to RAM is differentiated. Because the section name must be ".sdata", generate the ".sdata section" even when this area is not used by the user application. If the .sdata section is used, "YES" is shown in this column; if not, "--" is shown. If "YES" is shown, data with an initial value is necessary, so the initial value must be copied to RAM before program execution. In other words, ROMization processing must be performed using the "Copy Functions".
sbss	Differentiates whether or not this function/macro uses the memory area "sbss area". In other words, whether or not the function uses RAM as a temporary area is differentiated. As the section name must be ".sbss", generate the ".sbss section" even when this area is not used by the user application. If the .sbss section is used, "YES" is shown in this column; if not, "--" is shown. When data without an initial value is allocated by .sbss section, it is not necessary to perform ROMization processing at the time of "Use of .sdata".
Re-ent	Indicates whether or not the function is re-entrant. If it is re-entrant, "YES" is shown; if not, "--" is shown. "Re-entrant" means that the function can "re-enter". A re-entrant function can be correctly executed even if an attempt is made in another process to execute that function while the function is being executed. For example, in an application using a real-time OS, this function is correctly executed even if dispatching to another task is triggered by an interrupt while a certain task is executing this function, and even if the function is executed in that task. A function that must use RAM as a temporary area may not necessarily be re-entrant.

6.1.1 Standard library

The functions contained in the standard library are listed below.

(1) Function with variable arguments

Table 6-2. Function with Variable Arguments

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
va_start	stdarg.h	YES	--	--	--	YES
va_end	stdarg.h	YES	--	--	--	YES
va_arg	stdarg.h	YES	--	--	--	YES

(2) Character string functions

Table 6-3. Character String Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
index	string.h	--	--	--	--	YES
strpbrk	string.h	YES	--	--	--	YES
rindex	string.h	--	--	--	--	YES
strrchr	string.h	YES	--	--	--	YES
strchr	string.h	YES	--	--	--	YES
strstr	string.h	YES	--	--	--	YES
strspn	string.h	YES	--	--	--	YES
strcspn	string.h	YES	--	--	--	YES
strcmp	string.h	YES	--	--	--	YES
strncmp	string.h	YES	--	--	--	YES
strcpy	string.h	YES	--	--	--	YES
strncpy	string.h	YES	--	--	--	YES
strcat	string.h	YES	--	--	--	YES
strncat	string.h	YES	--	--	--	YES
strtok	string.h	YES	--	--	YES	--
strlen	string.h	YES	--	--	--	YES
strerror	string.h	YES	YES	YES	--	--

(3) Memory management functions

Table 6-4. Memory Management Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
memchr	string.h	YES	--	--	--	YES
memcmp	string.h	YES	--	--	--	YES
bcmp	string.h	--	--	--	--	YES
memcpy	string.h	YES	--	--	--	YES
bcopy	string.h	--	--	--	--	YES
memmove	string.h	YES	--	--	--	YES
memset	string.h	YES	--	--	--	YES

(4) Character conversion functions

Table 6-5. Character Conversion Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
toupper	ctype.h	YES	YES	--	--	YES
_toupper	ctype.h	--	--	--	--	YES

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
tolower	ctype.h	YES	YES	--	--	YES
_tolower	ctype.h	--	--	--	--	YES
toascii	ctype.h	--	--	--	--	YES

(5) Character classification functions

Table 6-6. Character Classification Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
isalnum	ctype.h	YES	YES	--	--	YES
isalpha	ctype.h	YES	YES	--	--	YES
isascii	ctype.h	YES	YES	--	--	YES
isupper	ctype.h	YES	YES	--	--	YES
islower	ctype.h	YES	YES	--	--	YES
isdigit	ctype.h	YES	YES	--	--	YES
isxdigit	ctype.h	YES	YES	--	--	YES
iscntrl	ctype.h	YES	YES	--	--	YES
ispunct	ctype.h	YES	YES	--	--	YES
isspace	ctype.h	YES	YES	--	--	YES
isprint	ctype.h	YES	YES	--	--	YES
isgraph	ctype.h	YES	YES	--	--	YES

(6) Standard I/O functions

Table 6-7. Standard I/O Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
fread	stdio.h	YES	--	--	--	YES
getc	stdio.h	YES	--	--	--	YES
fgetc	stdio.h	YES	--	--	--	YES
fgets	stdio.h	YES	--	--	--	YES
fwrite	stdio.h	YES	--	--	--	YES
putc	stdio.h	YES	--	--	--	YES
fputc	stdio.h	YES	--	--	--	YES
fputs	stdio.h	YES	--	--	--	YES
getchar	stdio.h	YES	--	YES	--	--
gets	stdio.h	YES	--	YES	--	--
putchar	stdio.h	YES	--	YES	--	--
puts	stdio.h	YES	--	YES	--	--
sprintf	stdio.h	YES	YES	--	YES	--Note

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
fprintf	stdio.h	YES	YES	--	YES	--Note
vsprintf	stdio.h	YES	YES	--	YES	--
printf	stdio.h	YES	YES	--	YES	--
vfprintf	stdio.h	YES	YES	--	YES	--Note
vprintf	stdio.h	YES	YES	--	YES	--
sscanf	stdio.h	YES	YES	--	--	YES
fscanf	stdio.h	YES	YES	--	--	YES
scanf	stdio.h	YES	YES	YES	--	--
ungetc	stdio.h	YES	--	--	--	YES
rewind	stdio.h	YES	--	--	--	YES
perror	stdio.h	YES	YES	YES	YES	--

Note A function is not re-entrant if errno is updated and matherrf (matherr)/matherrdis called when an exception occurs.

Remark errno.h must be included if errno is used when an exception occurs.

(7) Standard utility functions

Table 6-8. Standard Utility Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
abs	stdlib.h	YES	--	--	--	YES
labs	stdlib.h	YES	--	--	--	YES
llabs	stdlib.h	--	--	--	--	YES
bsearch	stdlib.h	YES	--	--	--	YES
qsort	stdlib.h	YES	--	--	--	YES
div	stdlib.h	YES	--	--	--	YES
ldiv	stdlib.h	YES	--	--	--	YES
lldiv	stdlib.h	--	--	--	--	YES
itoa	stdlib.h	--	--	--	--	YES
ltoa	stdlib.h	--	--	--	--	YES
ultoa	stdlib.h	--	--	--	--	YES
lltoa	stdlib.h	--	--	--	--	YES
ulltoa	stdlib.h	--	--	--	--	YES
ecvt	stdlib.h	--	YES	--	YES	--
ecvtf	stdlib.h	--	YES	--	YES	--
fcvt	stdlib.h	--	YES	--	YES	--
fcvtf	stdlib.h	--	YES	--	YES	--
gcvt	stdlib.h	--	YES	--	YES	--Note1

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
gcvtf	stdlib.h	--	YES	--	YES	--Note1
atoi	stdlib.h	YES	YES	--	YES	--Note2
atol	stdlib.h	YES	YES	--	YES	--Note2
atoll	stdlib.h	--	YES	--	YES	--Note2
strtol	stdlib.h	YES	YES	--	YES	--Note2
strtoul	stdlib.h	YES	YES	--	YES	--Note2
strtoll	stdlib.h	--	YES	--	YES	--Note2
strtoull	stdlib.h	--	YES	--	YES	--Note2
atoff	stdlib.h	YES	YES	--	YES	--Note2
atof	stdlib.h	YES	YES	--	--	--
strtodf	stdlib.h	YES	YES	--	YES	--Note2
strtod	stdlib.h	YES	YES	--	YES	--Note2
calloc	stdlib.h	YES	--	YES	YES	--
malloc	stdlib.h	YES	--	YES	YES	--
realloc	stdlib.h	YES	--	YES	YES	--
free	stdlib.h	YES	--	YES	YES	--
rand	stdlib.h	YES	--	YES	--	--
srand	stdlib.h	YES	--	YES	--	--

Notes 1. A function is not re-entrant if errno is updated and matherrf (matherr)/matherrdis called when an exception occurs.

2. A function is not re-entrant if errno is updated when an exception occur.

Remark errno.h must be included if errno is used when an exception occurs.

(8) Non-local jump functions

Table 6-9. Non-Local Jump Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
longjmp	setjmp.h	YES	--	--	--	--
setjmp	setjmp.h	YES	--	--	--	YES

6.1.2 Mathematical library

The functions contained in the mathematical library are listed below.

(1) Mathematical functions

Table 6-10. Mathematical Functions

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
j0f	math.h	--	YES	--	YES	--Note

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
j1f	math.h	--	YES	--	YES	--Note
jnf	math.h	--	YES	--	YES	--Note
y0f	math.h	--	YES	--	YES	--Note
y1f	math.h	--	YES	--	YES	--Note
ynf	math.h	--	YES	--	YES	--Note
erff	math.h	--	YES	--	YES	--Note
erfcf	math.h	--	YES	--	YES	--Note
expf	math.h	YES	YES	--	YES	--Note
exp	math.h	YES	YES	--	YES	--Note
logf	math.h	YES	YES	--	YES	--Note
log	math.h	YES	YES	--	YES	--Note
log2f	math.h	--	YES	--	YES	--Note
log10f	math.h	YES	YES	--	YES	--Note
log10	math.h	YES	YES	--	YES	--Note
powf	math.h	YES	YES	--	YES	--Note
pow	math.h	YES	YES	--	YES	--Note
sqrtf	math.h	YES	YES	--	YES	--Note
sqrt	math.h	YES	YES	--	YES	--Note
cbrtf	math.h	--	YES	--	YES	--Note
cbrt	math.h	--	YES	--	YES	--Note
ceilf	math.h	YES	--	--	--	YES
ceil	math.h	YES	--	--	--	YES
fabsf	math.h	YES	--	--	--	YES
fabs	math.h	YES	--	--	--	YES
floorf	math.h	YES	--	--	--	YES
floor	math.h	YES	--	--	--	YES
fmodf	math.h	YES	YES	--	YES	--Note
fmod	math.h	YES	YES	--	YES	--Note
frexpf	math.h	YES	YES	--	YES	--Note
frexp	math.h	YES	YES	--	YES	--Note
ldexpf	math.h	YES	YES	--	YES	--Note
ldexp	math.h	YES	YES	--	YES	--Note
modff	math.h	YES	--	--	--	YES
modf	math.h	YES	--	--	--	YES
gammaf	math.h	--	YES	--	YES	--Note
hypotf	math.h	--	YES	--	YES	--Note
matherrf (matherr)	math.h	--	--	--	--	YES

Function/Macro Name	#include	ANSI	const	sdata	sbss	Re-ent
matherrd	math.h	--	--	--	--	YES
cosf	math.h	YES	YES	--	YES	--Note
cos	math.h	YES	YES	--	YES	--Note
sinf	math.h	YES	YES	--	YES	--Note
sin	math.h	YES	YES	--	YES	--Note
tanf	math.h	YES	YES	--	YES	--Note
tan	math.h	YES	YES	--	YES	--Note
acosf	math.h	YES	YES	--	YES	--Note
acos	math.h	YES	YES	--	YES	--Note
asinf	math.h	YES	YES	--	YES	--Note
asin	math.h	YES	YES	--	YES	--Note
atanf	math.h	YES	YES	--	YES	--Note
atan	math.h	YES	YES	--	YES	--Note
atan2f	math.h	YES	YES	--	YES	--Note
atan2	math.h	YES	YES	--	YES	--Note
coshf	math.h	YES	YES	--	YES	--Note
cosh	math.h	YES	YES	--	YES	--Note
sinhf	math.h	YES	YES	--	YES	--Note
sinh	math.h	YES	YES	--	YES	--Note
tanhf	math.h	YES	YES	--	YES	--Note
tanh	math.h	YES	YES	--	YES	--Note
acoshf	math.h	--	YES	--	YES	--Note
asinhf	math.h	--	YES	--	YES	--Note
atanhf	math.h	--	YES	--	YES	--Note

Note A function is not re-entrant if errno is updated and matherrf (matherr)/matherrd is called when an exception occurs.

Remark "errno.h" must also be included if errno is used when an exception occurs, "limits.h" if "limit values of general integer type" are used as a macro name, and "float.h" if limit values of floating-point type are used.

6.1.3 Initialization library

The functions contained in the initialization library are listed below.

(1) Initialization peripheral devices function

The initialization peripheral devices function performs initialization of peripheral devices immediately after the CPU reset.

This is called from inside the startup routine.

The function included in the library is a dummy routine that performs no actions; code a function in accordance with your system.

Table 6-11. Initialization Peripheral Devices Function

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
hdwinit	--	--	--	--	YES

6.1.4 ROMization library

The functions contained in the ROMization library are listed below.

(1) Copy functions

Table 6-12. Copy Functions

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
_rcopy	--	--	--	--	YES
_rcopy1	--	--	--	--	YES
_rcopy2	--	--	--	--	YES
_rcopy4	--	--	--	--	YES

6.1.5 Multi-core library

The functions contained in the multi-core libraryare listed below.

(1) Pseudo "main" functions for multi-core

Table 6-13. Pseudo "main" Functions for Multi-core

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
main_pe2	--	--	--	--	YES
main_pe3	--	--	--	--	YES
main_pe4	--	--	--	--	YES
main_pe5	--	--	--	--	YES
main_pe6	--	--	--	--	YES
main_pe7	--	--	--	--	YES
main_pe8	--	--	--	--	YES
main_pe9	--	--	--	--	YES
main_pe10	--	--	--	--	YES
main_pe11	--	--	--	--	YES
main_pe12	--	--	--	--	YES
main_pe13	--	--	--	--	YES
main_pe14	--	--	--	--	YES
main_pe15	--	--	--	--	YES
main_pe16	--	--	--	--	YES
main_pe17	--	--	--	--	YES
main_pe18	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
main_pe19	--	--	--	--	YES
main_pe20	--	--	--	--	YES
main_pe21	--	--	--	--	YES
main_pe22	--	--	--	--	YES
main_pe23	--	--	--	--	YES
main_pe24	--	--	--	--	YES
main_pe25	--	--	--	--	YES
main_pe26	--	--	--	--	YES
main_pe27	--	--	--	--	YES
main_pe28	--	--	--	--	YES
main_pe29	--	--	--	--	YES
main_pe30	--	--	--	--	YES
main_pe31	--	--	--	--	YES

6.1.6 Runtime library

The functions contained in the runtime library are listed below.

(1) Operation runtime functions

Table 6-14. Operation Runtime Functions

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___addf.s	--	YES	--	--	--Note
___subf.s	--	YES	--	--	--Note
___mulf.s	--	YES	--	--	--Note
___divf.s	--	YES	--	--	--Note
___cmpf.s	--	YES	--	--	--Note
___fcmp.s	--	YES	--	--	--Note
___negf.s	--	YES	--	--	--Note
___notf.s	--	YES	--	--	--Note
___addf.d	--	YES	--	--	--Note
___subf.d	--	YES	--	--	--Note
___mulf.d	--	YES	--	--	--Note
___divf.d	--	YES	--	--	--Note
___fcmp.d	--	YES	--	--	--Note
___negf.d	--	YES	--	--	--Note
___notf.d	--	YES	--	--	--Note
___add.l	--	--	--	--	YES
___sub.l	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___mul.l	--	--	--	--	YES
___div.l	--	--	--	--	YES
___div.ul	--	--	--	--	YES
___mod.l	--	--	--	--	YES
___mod.ul	--	--	--	--	YES
___shl.l	--	--	--	--	YES
___shr.l	--	--	--	--	YES
___sar.l	--	--	--	--	YES
___inc.l	--	--	--	--	YES
___dec.l	--	--	--	--	YES
___not.l	--	--	--	--	YES
___neg.l	--	--	--	--	YES
___cmp.l	--	--	--	--	YES
___cmp.ul	--	--	--	--	YES
___bext.l	--	--	--	--	YES
___bext.ul	--	--	--	--	YES
___bins.l	--	--	--	--	YES
___cvt.ws	--	--	--	--	YES
___cvt.wd	--	--	--	--	YES
___cvt.uws	--	--	--	--	YES
___cvt.uwd	--	--	--	--	YES
___cvt.ls	--	--	--	--	YES
___cvt.ld	--	--	--	--	YES
___cvt.uls	--	--	--	--	YES
___cvt.uld	--	--	--	--	YES
___trnc.sw	--	--	--	--	YES
___trnc.dw	--	--	--	--	YES
___trnc.suw	--	--	--	--	YES
___trnc.duw	--	--	--	--	YES
___trnc.sl	--	--	--	--	YES
___trnc.dl	--	--	--	--	YES
___trnc.sul	--	--	--	--	YES
___trnc.dul	--	--	--	--	YES
___cvt.sd	--	--	--	--	YES
___cvt.ds	--	--	--	--	YES
__mul	--	--	--	--	YES
__mulu	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
__div	--	--	--	--	YES
__divu	--	--	--	--	YES
__mod	--	--	--	--	YES
__modu	--	--	--	--	YES

Note A function is not re-entrant if matherrf (matherr)/matherrd is called.

(2) Function pre/post processing runtime functions

Table 6-15. Function pre/post Processing Runtime Functions

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___Epush250	--	--	--	--	YES
___Epush251	--	--	--	--	YES
___Epush252	--	--	--	--	YES
___Epush253	--	--	--	--	YES
___Epush254	--	--	--	--	YES
___Epush260	--	--	--	--	YES
___Epush261	--	--	--	--	YES
___Epush262	--	--	--	--	YES
___Epush263	--	--	--	--	YES
___Epush264	--	--	--	--	YES
___Epush270	--	--	--	--	YES
___Epush271	--	--	--	--	YES
___Epush272	--	--	--	--	YES
___Epush273	--	--	--	--	YES
___Epush274	--	--	--	--	YES
___Epush280	--	--	--	--	YES
___Epush281	--	--	--	--	YES
___Epush282	--	--	--	--	YES
___Epush283	--	--	--	--	YES
___Epush284	--	--	--	--	YES
___Epush290	--	--	--	--	YES
___Epush291	--	--	--	--	YES
___Epush292	--	--	--	--	YES
___Epush293	--	--	--	--	YES
___Epush294	--	--	--	--	YES
___Epushlp0	--	--	--	--	YES
___Epushlp1	--	--	--	--	YES
___Epushlp2	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___Epushlp3	--	--	--	--	YES
___Epushlp4	--	--	--	--	YES
___push2000	--	--	--	--	YES
___push2001	--	--	--	--	YES
___push2002	--	--	--	--	YES
___push2003	--	--	--	--	YES
___push2004	--	--	--	--	YES
___push2040	--	--	--	--	YES
___push2100	--	--	--	--	YES
___push2101	--	--	--	--	YES
___push2102	--	--	--	--	YES
___push2103	--	--	--	--	YES
___push2104	--	--	--	--	YES
___push2140	--	--	--	--	YES
___push2200	--	--	--	--	YES
___push2201	--	--	--	--	YES
___push2202	--	--	--	--	YES
___push2203	--	--	--	--	YES
___push2204	--	--	--	--	YES
___push2240	--	--	--	--	YES
___push2300	--	--	--	--	YES
___push2301	--	--	--	--	YES
___push2302	--	--	--	--	YES
___push2303	--	--	--	--	YES
___push2304	--	--	--	--	YES
___push2340	--	--	--	--	YES
___push2400	--	--	--	--	YES
___push2401	--	--	--	--	YES
___push2402	--	--	--	--	YES
___push2403	--	--	--	--	YES
___push2404	--	--	--	--	YES
___push2440	--	--	--	--	YES
___push2500	--	--	--	--	YES
___push2501	--	--	--	--	YES
___push2502	--	--	--	--	YES
___push2503	--	--	--	--	YES
___push2504	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___push2540	--	--	--	--	YES
___push2600	--	--	--	--	YES
___push2601	--	--	--	--	YES
___push2602	--	--	--	--	YES
___push2603	--	--	--	--	YES
___push2604	--	--	--	--	YES
___push2640	--	--	--	--	YES
___push2700	--	--	--	--	YES
___push2701	--	--	--	--	YES
___push2702	--	--	--	--	YES
___push2703	--	--	--	--	YES
___push2704	--	--	--	--	YES
___push2740	--	--	--	--	YES
___push2800	--	--	--	--	YES
___push2801	--	--	--	--	YES
___push2802	--	--	--	--	YES
___push2803	--	--	--	--	YES
___push2804	--	--	--	--	YES
___push2840	--	--	--	--	YES
___push2900	--	--	--	--	YES
___push2901	--	--	--	--	YES
___push2902	--	--	--	--	YES
___push2903	--	--	--	--	YES
___push2904	--	--	--	--	YES
___push2940	--	--	--	--	YES
___pushlp00	--	--	--	--	YES
___pushlp01	--	--	--	--	YES
___pushlp02	--	--	--	--	YES
___pushlp03	--	--	--	--	YES
___pushlp04	--	--	--	--	YES
___pushlp40	--	--	--	--	YES
___pop2000	--	--	--	--	YES
___pop2001	--	--	--	--	YES
___pop2002	--	--	--	--	YES
___pop2003	--	--	--	--	YES
___pop2004	--	--	--	--	YES
___pop2040	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___pop2100	--	--	--	--	YES
___pop2101	--	--	--	--	YES
___pop2102	--	--	--	--	YES
___pop2103	--	--	--	--	YES
___pop2104	--	--	--	--	YES
___pop2140	--	--	--	--	YES
___pop2200	--	--	--	--	YES
___pop2201	--	--	--	--	YES
___pop2202	--	--	--	--	YES
___pop2203	--	--	--	--	YES
___pop2204	--	--	--	--	YES
___pop2240	--	--	--	--	YES
___pop2300	--	--	--	--	YES
___pop2301	--	--	--	--	YES
___pop2302	--	--	--	--	YES
___pop2303	--	--	--	--	YES
___pop2304	--	--	--	--	YES
___pop2340	--	--	--	--	YES
___pop2400	--	--	--	--	YES
___pop2401	--	--	--	--	YES
___pop2402	--	--	--	--	YES
___pop2404	--	--	--	--	YES
___pop2440	--	--	--	--	YES
___pop2500	--	--	--	--	YES
___pop2501	--	--	--	--	YES
___pop2502	--	--	--	--	YES
___pop2503	--	--	--	--	YES
___pop2504	--	--	--	--	YES
___pop2540	--	--	--	--	YES
___pop2600	--	--	--	--	YES
___pop2601	--	--	--	--	YES
___pop2602	--	--	--	--	YES
___pop2603	--	--	--	--	YES
___pop2604	--	--	--	--	YES
___pop2640	--	--	--	--	YES
___pop2700	--	--	--	--	YES
___pop2701	--	--	--	--	YES

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
___pop2702	--	--	--	--	YES
___pop2703	--	--	--	--	YES
___pop2704	--	--	--	--	YES
___pop2740	--	--	--	--	YES
___pop2800	--	--	--	--	YES
___pop2801	--	--	--	--	YES
___pop2802	--	--	--	--	YES
___pop2803	--	--	--	--	YES
___pop2804	--	--	--	--	YES
___pop2840	--	--	--	--	YES
___pop2900	--	--	--	--	YES
___pop2901	--	--	--	--	YES
___pop2902	--	--	--	--	YES
___pop2903	--	--	--	--	YES
___pop2904	--	--	--	--	YES
___pop2940	--	--	--	--	YES
___poplp00	--	--	--	--	YES
___poplp01	--	--	--	--	YES
___poplp02	--	--	--	--	YES
___poplp03	--	--	--	--	YES
___poplp04	--	--	--	--	YES
___poplp40	--	--	--	--	YES

6.1.7 Libraries used in V850E2V3-FPU

The functions contained in the libraries used in V850E2V3-FPUare listed below.

(1) Functions used in V850E2V3-FPU

Table 6-16. Functions Used in V850E2V3-FPU

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
expf	YES	YES	--	YES	--Note
exp	YES	YES	--	YES	--Note
logf	YES	YES	--	YES	--Note
log	YES	YES	--	YES	--Note
log10f	YES	YES	--	YES	--Note
log10	YES	YES	--	YES	--Note
powf	YES	YES	--	YES	--Note
pow	YES	YES	--	YES	--Note

Function/Macro Name	ANSI	const	sdata	sbss	Re-ent
sqrtf	YES	YES	--	YES	--Note
sqrt	YES	YES	--	YES	--Note
ceilf	YES	--	--	--	YES
ceil	YES	--	--	--	YES
floorf	YES	--	--	--	YES
floor	YES	--	--	--	YES
fmodf	YES	YES	--	YES	--Note
fmod	YES	YES	--	YES	--Note
frexpf	YES	YES	--	YES	--Note
frexp	YES	YES	--	YES	--Note
ldexpf	YES	YES	--	YES	--Note
ldexp	YES	YES	--	YES	--Note
modff	YES	--	--	--	YES
modf	YES	--	--	--	YES
cosf	YES	YES	--	YES	--Note
cos	YES	YES	--	YES	--Note
sinf	YES	YES	--	YES	--Note
sin	YES	YES	--	YES	--Note
tanf	YES	YES	--	YES	--Note
tan	YES	YES	--	YES	--Note
acosf	YES	YES	--	YES	--Note
acos	YES	YES	--	YES	--Note
asinf	YES	YES	--	YES	--Note
asin	YES	YES	--	YES	--Note
atanf	YES	YES	--	YES	--Note
atan	YES	YES	--	YES	--Note
atan2f	YES	YES	--	YES	--Note
atan2	YES	YES	--	YES	--Note
coshf	YES	YES	--	YES	--Note
cosh	YES	YES	--	YES	--Note
sinhf	YES	YES	--	YES	--Note
sinh	YES	YES	--	YES	--Note
tanhf	YES	YES	--	YES	--Note
tanh	YES	YES	--	YES	--Note

Note A function is not re-entrant if errno is updated and matherrf (matherr)/matherrdis called when an exception occurs.

6.2 Header Files

The list of header files required for using the libraries of the CX are listed below.
The macro definitions and function declarations are described in each file.

Table 6-17. Header Files

File Name	Outline
ctype.h	Header file for character conversion and classification
errno.h	Header file for reporting error condition
float.h	Header file for floating-point representation and floating-point operation
limits.h	Header file for quantitative limiting of integers
math.h	Header file for mathematical calculation
setjmp.h	Header file for non-local jump
stdarg.h	Header file for supporting functions having variable arguments
stddef.h	Header file for common definitions
stdio.h	Header file for standard I/O
stdlib.h	Header file for standard utilities
string.h	Header file for memory manipulation and character string manipulation

6.3 Re-entrant

"Re-entrant" means that the function can "re-enter". A re-entrant function can be correctly executed even if an attempt is made in another process to execute that function while the function is being executed. For example, in an application using a real-time OS, this function is correctly executed even if dispatching to another task is triggered by an interrupt while a certain task is executing this function, and even if the function is executed in that task. A function that must use RAM as a temporary area may not necessarily be re-entrant.

For re-entrant of each function, see tables from "Table 6-2. Function with Variable Arguments" to "Table 6-15. Function pre/post Processing Runtime Functions".

6.4 Library Function

This section explains Library Function.

6.4.1 Functions with variable arguments

Functions with a variable arguments are as follows

Table 6-18. Functions with Variable Arguments

Function/Macro Name	Outline
va_start	Initialization of variable for scanning argument list
va_end	End of scanning argument list
va_arg	Moving variable for scanning argument list

va_start

Initialization of variable for scanning argument list

[Classification]

Standard library

[Syntax]

#include <stdarg.h>
void va_start(va_list *ap*, *last-named-argument*);

[Description]

This function initializes variable *ap* so that it indicates the beginning (argument next to last-named-argument) of the list of the variable arguments.

To define function *func* having a variable arguments in a portable form, the following format is used.

```
#include    <stdarg.h>
void func(arg-declarations, ...) {
        va_list ap;
        type    argN;
        va_start(ap, last-named-argument);
        argN = va_arg(ap, type);
        va_end(ap);
}
```

Remark *arg-declarations* is an argument list with the *last-named-argument* declared at the end. ", ..." that follows indicates a list of the variable arguments. va_listis the type of the variable (*ap* in the above example) used to scan the argument list.

[Example]

```
#include    <stdarg.h>
void abc(int first, int second, ...) {
        va_list ap;
        int     i;
        char    c, *fmt;
        va_start(ap, second);
        i = va_arg(ap, int);
        c = va_arg(ap, int);        /*char type is converted into int type.*/
        fmt = va_arg(ap, char *);
        va_end(ap);
}
```

va_end

End of scanning argument list

[Classification]

Standard library

[Syntax]

#include <stdarg.h>
void va_end(va_list *ap*);

[Description]

This function indicates the end of scanning the list. By enclosing va_arg between va_start and va_end, scanning the list can be repeated.

va_arg

Moving variable for scanning argument list

[Classification]

Standard library

[Syntax]

#include <stdarg.h>
type va_arg(va_list *ap*, *type*);

[Description]

This function returns the argument indicated by variable *ap*, and advances variable *ap* to indicate the next argument. For the *type* of va_arg, specify the type converted when the argument is passed to the function. With the C compiler specify the int type for an argument of char and short types, and specify the unsigned int type for an argument of unsigned char and unsigned short types. Although a different type can be specified for each argument, stipulate "which type of argument is passed" according to the conventions between the called function and calling function.

Also stipulate "how many functions are actually passed" according to the conventions between the called function and calling function.

6.4.2 Character string functions

Character string functions are as follows.

Table 6-19. Character String Functions

Function/Macro Name	Outline
index	Character string search (start position)
strpbrk	Character string search (start position)
rindex	Character string search (end position)
strrchr	Character string search (end position)
strchr	Character string search (start position of specified character)
strstr	Character string search (start position of specified character string)
strspn	Character string search (maximum length including specified character)
strcspn	Character string search (maximum length not including specified character)
strcmp	Character string comparison
strncmp	Character string comparison (with number of characters specified)
strcpy	Character string copy
strncpy	Character string copy (with number of characters specified)
strcat	Character string concatenation
strncat	Character string concatenation (with number of characters specified)
strtok	Token division
strlen	Length of character string
strerror	Character string conversion of error number

index

Character string search (start position)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *index(const char *s, int c);
```

[Return value]

Returns a pointer indicating the character that has been found. If *c* does not appear in this character string, the null pointer is returned.

[Description]

This function obtains the position at which a character the same as *c* converted into char type appears in the character string indicated by *s*. The null character (\0) indicating termination is regarded as part of this character string.

strpbrk

Character string search (start position)

[Classification]

Standard library

[Syntax]

#include <string.h>
char *strpbrk(const char **s1*, const char **s2*);

[Return value]

Returns the pointer indicating this character. If any of the characters from *s2* does not appear in *s1*, the null pointer is returned.

[Description]

This function obtains the position in the character string indicated by *s1* at which any of the characters in the character string indicated by *s2* (except the null character (\0)) appears first.

rindex

Character string search (end position)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *rindex(const char *s, int c);
```

[Return value]

Returns a pointer indicating *c* that has been found. If *c* does not appear in this character string, the null pointer is returned.

[Description]

This function obtains the position at which *c* converted into char type appears last in the character string indicated by *s*. The null character (\0) indicating termination is regarded as part of this character string.

strrchr

Character string search (end position)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strrchr(const char *s, int c);
```

[Return value]

Returns a pointer indicating *c* that has been found. If *c* does not appear in this character string, the null pointer is returned.

[Description]

This function obtains the position at which *c* converted into char type appears last in the character string indicated by *s*. The null character (\0) indicating termination is regarded as part of this character string.

strchr

Character string search (start position of specified character)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strchr(const char *s, int c);
```

[Return value]

Returns a pointer indicating the character that has been found. If *c* does not appear in this character string, the null pointer is returned.

[Description]

This function obtains the position at which a character the same as *c* converted into char type appears in the character string indicated by *s*. The null character (\0) indicating termination is regarded as part of this character string.

strstr

Character string search (start position of specified character)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strstr(const char *s1, const char *s2);
```

[Return value]

Returns the pointer indicating the character string that has been found. If character string *s2* is not found, the null pointer is returned. If *s2* indicates a character string with a length of 0, *s1* is returned.

[Description]

This function obtains the position of the portion (except the null character (\0)) that first coincides with the character string indicated by *s2*, in the character string indicated by *s1*.

strspn

Character string search (maximum length including specified character)

[Classification]

Standard library

[Syntax]

#include <string.h>
size_t strspn(const char *s1, const char *s2);

[Return value]

Returns the length of the portion that has been found.

[Description]

This function obtains the maximum and first length of the portion consisting of only the characters (except the null character (\0)) in the character string indicated by *s2*, in the character string indicated by *s1*.

strcspn

Character string search (maximum length not including specified character)

[Classification]

Standard library

[Syntax]

#include <string.h>
size_t strcspn(const char *s1, const char *s2);

[Return value]

Returns the length of the portion that has been found.

[Description]

This function obtains the length of the maximum and first portion consisting of characters missing from the character string indicated by *s2* (except the null character (\0) at the end) in the character string indicated by *s1*.

strcmp

Character string comparison

[Classification]

Standard library

[Syntax]

#include <string.h>
int strcmp(const char *s1, const char *s2);

[Return value]

Returns an integer greater than, equal to, or less than 0, depending on whether the character string indicated by *s1* is greater than, equal to, or less than the character string indicated by *s2*.

[Description]

This function compares the character string indicated by *s1* with the character string indicated by *s2*.

strncmp

Character string comparison (with number of characters specified)

[Classification]

Standard library

[Syntax]

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t *length*);

[Return value]

Returns an integer greater than, equal to, or less than 0, depending on whether the character string indicated by *s1* is greater than, equal to, or less than the character string indicated by *s2*.

[Description]

This function compares up to *length* characters of the array indicated by *s1* with characters of the array indicated by *s2*.

strcpy

Character string copy

[Classification]

Standard library

[Syntax]

#include <string.h>
char *strcpy(char **dst*, const char **src*);

[Return value]

Returns the value of *dst*.

[Description]

This function copies the character string indicated by *src* to the array indicated by *dst*.

[Example]

```
#include    <string.h>
void func(char *str, const char *src) {
        strcpy(str, src);   /*Copies character string indicated by src to array indicated by
                              str.*/
          :
}
```

strncpy

Character string copy (with number of characters specified)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strncpy(char *dst, const char *src, size_t length);
```

[Return value]

Returns the value of *dst*.

[Description]

This function copies up to *length* characters (including the null character (\0)) from the array indicated by *src* to the array indicated by *dst*. If the array indicate by *src* is shorter than *length* characters, null characters (\0) are appended to the duplication in the array indicated by *dst*, until all *length* characters are written.

strcat

Character string concatenation

[Classification]

Standard library

[Syntax]

#include <string.h>
char *strcat(char **dst*, const char **src*);

[Return value]

Returns the value of *dst*.

[Description]

This function concatenates the duplication of the character string indicated by *src* to the end of the character string indicated by *dst*, including the null character (\0). The first character of *src* overwrites the null character (\0) at the end of *dst*.

strncat

Character string concatenation (with number of characters specified)

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strncat(char *dst, const char *src, size_t length);
```

[Return value]

Returns the value of *dst*.

[Description]

This function concatenates up to *length* characters (including the null character (\0) of *src*) to the end of the character string indicated by *dst*, starting from the beginning of the character string indicated by *src*. The null character (\0) at the end of *dst* is written over the first character of *src*. The null character indicating termination (\0) is always added to this result.

[Caution]

Because the null character (\0) is always appended when strncat is used, if copying is limited by the number of *length* arguments, the number of characters appended to *dst* is *length* + 1.

strtok

Token division

[Classification]

Standard library

[Syntax]

```
#include <string.h>
char    *strtok(char *s, const char *delimiters);
```

[Return value]

Returns a pointer to a token. If a token does not exist, the null pointer is returned.

[Description]

This function divides the character string indicated by *s* into strings of tokens by delimiting the character string with a character in the character string indicated by *delimiters*. If this function is called first, *s* is used as the first argument. Then, calling with the null pointer as the first argument continues. The delimiting character string indicated by *delimiters* can differ on each call. On the first call, the character string indicated by s is searched for the first character not included in the delimiting character string indicated by *delimiters*. If such a character is not found, a token does not exist in the character string indicated by *s*, and strtok returns the null pointer. If a character is found, that character is the beginning of the first token. After that, strtok searches from the position of that character for a character included in the delimiting character string at that time.

If such a character is not found, the token is expanded to the end of the character string indicated by *s*, and the subsequent search returns the null pointer. If a character is found, the subsequent character is overwritten by the null character (\0) indicating the termination of the token. strtok saves the pointer indicating the subsequent character. If the null pointer is used as the value of the first argument, a code that is not re-entrant is returned. This can be avoided by preserving the address of the last delimiting character in the application program, and passing *s* as an argument that is not vacant, by using this address.

strlen

Length of character string

[Classification]

Standard library

[Syntax]

```
#include <string.h>
size_t  strlen(const char *s);
```

[Return value]

Returns the number of characters existing before the null character (\0) indicating termination.

[Description]

This function obtains the length of the character string indicated by *s*.

strerror

Character string conversion of error number

[Classification]

Standard library

[Syntax]

#include <string.h>
char *strerror(int *errnum*);

[Return value]

Returns a pointer to the converted character string.

[Description]

This function converts error number *errnum* into a character string according to the correspondence relationship of the processing system definition. The value of *errnum* is usually the duplication of global variable errno. Do not change the specified array of the application program.

6.4.3 Memory management functions

Memory management functions are as follows.

Table 6-20. Memory Management Functions

Function/Macro Name	Outline
memchr	Memory search
memcmp	Memory comparison
bcmp	Memory comparison (char argument version of memcmp)
memcpy	Memory copy
bcopy	Memory copy (char argument version of memcpy)
memmove	Memory move
memset	Memory set

memchr

Memory search

[Classification]

Standard library

[Syntax]

#include <string.h>
void *memchr(const void *s, int c, size_t *length*);

[Return value]

If *c* is found, a pointer indicating this character is returned. If *c* is not found, the null pointer is returned.

[Description]

This function obtains the position at which character *c* (converted into char type) appears first in the first *length* number of characters in an area indicated by *s*.

memcmp

Memory comparison

[Classification]

Standard library

[Syntax]

#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);

[Return value]

An integer greater than, equal to, or less than *0* is returned, depending on whether the object indicated by *s1* is greater than, equal to, or less than the object indicated by *s2*.

[Description]

This function compares the first n characters of an object indicated by *s1* with the object indicated by *s2*.

[Example]

```
#include    <string.h>
int func(const void *s1, const void *s2) {
        int i;
        i = memcmp(s1, s2, 5);  /*Compares the first five characters of the character
                                  string indicated by s1 with the first five characters of
                                  the character string indicated by s2*/
        return(i);
}
```

bcmp

Memory comparison (char argument version of memcmp)

[Classification]

Standard library

[Syntax]

#include <string.h>

int bcmp(const char *s1, const char *s2, size_t n);

[Return value]

An integer greater than, equal to, or less than *0* is returned, depending on whether the object indicated by *s1* is greater than, equal to, or less than the object indicated by *s2*.

[Description]

This function compares the first *n* characters of an object indicated by *s1* with the object indicated by *s2*.

memcpy

Memory copy

[Classification]

Standard library

[Syntax]

```
#include <string.h>
void    *memcpy(void *out, const void *in, size_t n);
```

[Return value]

Returns the value of *out*. The operation is undefined if the copy source and copy destination areas overlap.

[Description]

This function copies *n* bytes from an object indicated by *in* to an object indicated by *out*.

bcopy

Memory copy (char argument version of memcpy)

[Classification]

Standard library

[Syntax]

#include <string.h>

char* bcopy(const char **in*, char **out*, size_t *n*);

[Return value]

Returns the value of *out*. The operation is undefined if the copy source and copy destination areas overlap.

[Description]

This function copies *n* bytes from an object indicated by *in* to an object indicated by *out*.

memmove

Memory move

[Classification]

Standard library

[Syntax]

```
#include <string.h>
void    *memmove(void *dst, void *src, size_t length);
```

[Return value]

Returns the value of *dst* at the copy destination.

[Description]

This function moves the *length* number of characters from a memory area indicated by *src* to a memory area indicated by *dst*. Even if the copy source and copy destination areas overlap, the characters are correctly copied to the memory area indicated by *dst*.

memset

Memory set

[Classification]

Standard library

[Syntax]

```
#include <string.h>
void    *memset(const void *s, int c, size_t length);
```

[Return value]

Returns the value of *s*.

[Description]

This function copies the value of *c* (converted into unsigned char type) to the first *length* character of an object indicated by *s*.

6.4.4 Character conversion functions

Character conversion functions are as follows.

Table 6-21. Character Conversion Functions

Function/Macro Name	Outline
toupper	Conversion from lower-case to upper-case (not converted if argument is not in lower-case)
_toupper	Conversion from lower-case to upper-case (correctly converted only if argument is in lower-case)
tolower	Conversion from upper-case to lower-case (not converted if argument is not in upper-case)
_tolower	Conversion from upper-case to lower-case (correctly converted only if argument is in upper-case)
toascii	Conversion from integer to ASCII character

toupper

Conversion from lower-case to upper-case (not converted if argument is not in lower-case)

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int toupper(int *c*);

[Return value]

If islower is true with respect to *c*, returns a character that makes isupper true in response; otherwise, returns *c*.

[Description]

This function is a macro that converts lowercase characters into the corresponding uppercase characters and leaves the other characters unchanged.

This macro is defined only when *c* is an integer in the range of EOF to 255. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef toupper".

[Example]

```
#include    <ctype.h>
int chc = 'a';
int ret = func(chc);
int func(int c) {
        int i;
        i = toupper(c);         /*Converts lowercase character 'a' of c into uppercase
                                  character 'A'.*/
        return(i);
}
```

_toupper

Conversion from lower-case to upper-case (correctly converted only if argument is in lower-case)

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int _toupper(int *c*);

[Return value]

If islower is true with respect to *c*, returns a character that makes isupper true in response; otherwise, returns *c*.
Also with _toupper, operation can be inconsistent when specifying illegal values for *c*.

[Description]

This function is a macro that performs the same operation as toupper if the argument is of lowercase characters.
Because the argument is not checked, the correct conversion is performed only if the argument is of lowercase characters. If otherwise, the operation will be undefined. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef _toupper".

tolower

Conversion from upper-case to lower-case (not converted if argument is not in upper-case)

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int tolower(int *c*);

[Return value]

If isupper is true with respect to c, returns a character that makes islower true in response; otherwise, returns *c*.

[Description]

This function is a macro that converts uppercase characters into the corresponding lowercase characters and leaves the other characters unchanged.

This macro is defined only when *c* is an integer in the range of EOF to 255. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef tolower".

_tolower

Conversion from upper-case to lower-case (correctly converted only if argument is in upper-case)

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int _tolower(int *c*);

[Return value]

If isupper is true with respect to c, returns a character that makes islower true in response; otherwise, returns *c*.
Also with _tolower, operation can be inconsistent when specifying illegal values for *c*.

[Description]

This function is a macro that performs the same operation as tolower if the argument is of uppercase characters.
Because the argument is not checked, the correct conversion is performed only if the argument is of uppercase characters. If otherwise, the operation will be undefined. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef _tolower".

toascii

Conversion from integer to ASCII character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int toascii(int *c*);

[Return value]

Returns an integer in the range of 0 to 127.

[Description]

This function is a macro that forcibly converts an integer into an ASCII character (0 to 127) by clearing bit 8 and higher of the argument to 0.

A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef toascii".

6.4.5 Character classification functions

Character classification functions are as follows.

Table 6-22. Character Classification Functions

Function/Macro Name	Outline
isalnum	Identification of ASCII letter or numeral
isalpha	Identification of ASCII letter
isascii	Identification of ASCII code
isupper	Identification of upper-case character
islower	Identification of lower-case character
isdigit	Identification of decimal number
isxdigit	Identification of hexadecimal number
iscntrl	Identification of control character
ispunct	Identification of delimiter character
isspace	Identification of space/tab/carriage return/line feed/vertical tab/page feed
isprint	Identification of display character
isgraph	Identification of display character other than space

isalnum

Identification of ASCII letter or numeral

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isalnum(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is an ASCII alphabetic character or numeral. This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isalnum".

isalpha

Identification of ASCII letter

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isalpha(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is an ASCII alphabetic character. This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isalpha".

isascii

Identification of ASCII code

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isascii(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is an ASCII code (0x00 to 0x7F). This macro is defined for all integer values. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isascii".

isupper

Identification of upper-case character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isupper(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is an uppercase character (A to Z). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isupper".

islower

Identification of lower-case character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int islower(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a lowercase character (a to z). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef islower".

isdigit

Identification of decimal number

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isdigit(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a decimal number. This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isdigit".

isxdigit

Identification of hexadecimal number

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isxdigit(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a hexadecimal number (0 to 9, a to f, or A to F). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isxdigit".

iscntrl

Identification of control character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int iscntrl(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a control character (0x00 to 0x1F or 0x7F). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef iscntrl".

ispunct

Identification of delimiter character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int ispunct(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a printable delimiter (isgraph(c) && !isalnum(c)). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef ispunct".

isspace

Identification of space/tab/carriage return/line feed/vertical tab/page feed

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isspace(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a space, tap, line feed, carriage return, vertical tab, or form feed (0x09 to 0x0D, or 0x20). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isspace".

isprint

Identification of display character

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isprint(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a display character (0x20 to 0x7E). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isprint".

[Example]

```
#include    <ctype.h>
void func(void) {
        int     i, j = 0;
        char    s[50];
                for (i =50; i <= 99; i++) {
                        if (isprint(i)) {   /*Store the printable characters in the code
                                              range 50 to 99, in the array s.*/
                        s[j] = i;
                        j++;
                        }
                }
        :
}
```

isgraph

Identification of display character other than space

[Classification]

Standard library

[Syntax]

#include <ctype.h>
int isgraph(int *c*);

[Return value]

These macros return a value other than 0 if the value of argument *c* matches the respective description (i.e., if the result is true). If the result is false, 0 is returned.

[Description]

This function is a macro that checks whether a given character is a display character[Note] (0x20 to 0x7E) other than space (0x20). This macro is defined only when *c* is made true by isascii or when *c* is EOF. A compiled subroutine can be used instead of the macro definition, which is invalidated by using "#undef isgraph".

Note printing character

6.4.6 Standard I/O functions

Standard I/O functions are as follows.

Table 6-23. Standard I/O Functions

Function/Macro Name	Outline
fread	Read from stream
getc	Read character from stream (same as fgetc)
fgetc	Read character from stream (same as getc)
fgets	Read one line from stream
fwrite	Write to stream
putc	Write character to stream
fputc	Write character to stream
fputs	Output character string to stream
getchar	Read one character from standard input
gets	Read character string from standard input
putchar	Write character to standard output stream
puts	Output character string to standard output stream
sprintf	Output with format
fprintf	Output text in specified format to stream
vsprintf	Write text in specified format to character string
printf	Output text in specified format to standard output stream
vfprintf	Write text in specified format to stream
vprintf	Write text in specified format to standard output stream
sscanf	Input with format
fscanf	Read and interpret data from stream
scanf	Read and interpret text from standard input stream
ungetc	Push character back to input stream
rewind	Reset file position indicator
perror	Error processing

fread

Read from stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
size_t fread(void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

[Return value]

The number of elements that were input (_nmemb_) is returned.
Error return does not occur.

[Description]

This function inputs _nmemb_ elements of _size_ from the input stream pointed to by _stream_ and stores them in _ptr_. Only the standard input/output stdin can be specified for _stream_.

[Example]

```
#include    <stdio.h>
void func(void) {
        struct {
                int     c;
                double  d;
        } buf[10];
        fread(buf, sizeof(buf[0]), sizeof(buf) / sizeof(buf [0]), stdin);
}
```

getc

Read character from stream (same as fgetc)

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int getc(FILE *stream);
```

[Return value]

The input character is returned.
Error return does not occur.

[Description]

This function inputs one character from the input stream pointed to by *stream*. Only the standard input/output stdin can be specified for *stream*.

fgetc

Read character from stream (same as getc)

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int fgetc(FILE **stream*);

[Return value]

The input character is returned.
Error return does not occur.

[Description]

This function inputs one character from the input stream pointed to by *stream*. Only the standard input/output stdin can be specified for *stream*.

[Example]

```
#include    <stdio.h>

int func(void) {
        int c;
        c = fgetc(stdin);
        return(c);
}
```

fgets

Read one line from stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
char    *fgets(char *s, int n, FILE *stream);
```

[Return value]

s is returned.
Error return does not occur.

[Description]

This function inputs at most *n-1* characters from the input stream pointed to by *stream* and stores them in *s*. Character input is also ended by the detection of a new-line character. In this case, the new-line character is also stored in *s*. The end-of-string null character is stored at the end in *s*. Only the standard input/output stdin can be specified for *stream*.

fwrite

Write to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
size_t  fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

[Return value]

The number of elements that were output (*nmemb*) is returned.
Error return does not occur.

[Description]

This function outputs *nmemb* elements of *size* from the array pointed to by *ptr* to the output stream pointed to by *stream*. Only the standard input/output stdout or stderr can be specified for *stream*.

putc

Write character to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int putc(int *c*, FILE **stream*);

[Return value]

The character *c* is returned.
Error return does not occur.

[Description]

This function outputs the character *c* to the output stream pointed to by *stream.* Only the standard input/output stdout or stderr can be specified for *stream*.

fputc

Write character to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int fputc(int *c*, FILE **stream*);

[Return value]

The character *c* is returned.
Error return does not occur.

[Description]

This functionoutputs the character *c* to the output stream pointed to by *stream*. Only the standard input/output stdout or stderr can be specified for *stream*.

[Example]

```
#include    <stdio.h>
void func(void) {
        fputc('a', stdout);
}
```

fputs

Output character string to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

[Return value]

0 is returned.
Error return does not occur.

[Description]

This function outputs the string *s* to the output stream pointed to by *stream*. The end-of-string null character is not output. Only the standard input/output stdout or stderr can be specified for *stream*.

getchar

Read one character from standard input

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int getchar(void);
```

[Return value]

The input character is returned.
Error return does not occur.

[Description]

This function inputs one character from the standard input/output stdin.

gets

Read character string from standard input

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
char    *gets(char *s);
```

[Return value]

s is returned.
Error return does not occur.

[Description]

This function inputs characters from the standard input/output stdin until a new-line character is detected and stores them in *s*. The new-line character that was input is discarded, and an end-of-string null character is stored at the end in *s*.

putchar

Write character to standard output stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int putchar(int *c*);

[Return value]

The character *c* is returned.
Error return does not occur.

[Description]

This function outputs the character *c* to the standard input/output stdout.

puts

Output character string to standard output stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int puts(const char *s);
```

[Return value]

0 is returned.
Error return does not occur.

[Description]

This function outputs the string *s* to the standard input/output stdout. The end-of-string null character is not output, but a new-line character is output in its place.

sprintf

Output with format

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int sprintf(char *s, const char **format*[, *arg*, ...]);

[Return value]

The number of characters that were output (excluding the null character (\0)) is returned.
Error return does not occur.

[Description]

This function applies the format specified by the string pointed to by *format* to the respective *arg* arguments, and writes out the formatted data that was output as a result to the array pointed to by *s*.

If there are not sufficient arguments for the format, the operation is undefined. If the end of the formatted string is reached, control returns. If there are more arguments that those required by the format, the excess arguments are ignored. If the area of *s* overlaps one of the arguments, the operation is undefined.

The argument *format* specifies "the output to which the subsequent argument is to be converted". The null character (\0) is appended at the end of written characters (the null character (\0) is not counted in a return value).

The *format* consists of the following two types of directives:

Ordinary characters	Characters that are copied directly without conversion (other than "%").
Conversion specifications	Specifications that fetch zero or more arguments and assign a specification.

Each conversion specification begins with character "%" (to insert "%" in the output, specify "%%" in the format string). The following appear after the "%":

%[flag][field-width][precision][size][type-specification-character]

The meaning of each conversion specification is explained below.

(1) flag

Zero or more flags, which qualify the meaning of the conversion specification, are placed in any order.
The flag characters and their meanings are as follows:

-	The result of the conversion will be left-justified in the field, with the right side filled with blanks (if this flag is not specified, the result of the conversion is right-justified).
+	The result of a signed conversion will start with a + or - sign (if this flag is not specified, the result of the conversion starts with a sign only when a negative value has been converted).
Space	If the first character of a signed conversion is not a sign and a signed conversion is not generated a character, a space (" ") will be appended to the beginning of result of the conversion. If both the space flag and + flag appear, the space flag is ignored.

#	The result is to be converted to an alternate format. For o conversion, the precision is increased so that the first digit of the conversion result is 0. For x or X conversion, 0x or 0X is appended to the beginning of a non-zero conversion result. For e, f, g, E, or G conversion, a decimal point "." is added to the conversion result even if no digits follow the decimal point[Note]. For g or G conversion, trailing zeros will not be removed from the conversion result. The operation is undefined for conversions other than the above.
0	For d, e, f, g, i, o, u, x, E, G, or X conversion, zeros are added following the specification of the sign or base to fill the field width. If both the 0 flag and - flag are specified, the 0 flag is ignored. For d, i, o, u, x, or X conversion, when the precision is specified, the zero (0) flag is ignored. Note that 0 is interpreted as a flag and not as the beginning of the field width. The operation is undefined for conversion other than the above.

Note Normally, a decimal point appears only when a digit follows it.

(2) field width

This is an optional minimum field width. If the converted value is smaller than this field width, the left side is filled with spaces (if the left justification flag explained above is assigned, the right side will be filled with spaces). This field width takes the form of "*" or a decimal integer. If "*" is specified, an int type argument is used as the field width. A negative field width is not supported. If an attempt is made to specify a negative field width, it is interpreted as a minus (-) flag appended to the beginning of a positive field width.

(3) precision

For d, i, o, u, x, or X conversion, the value assigned for the precision is the minimum number of digits to appear. For e, f, or E conversion, it is the number of digits to appear after the decimal point. For g or G conversion, it is the maximum number of significant digits. The precision takes the form of "*" or "." followed by a decimal integer. If "*" is specified, an int type argument is used as the precision. If a negative precision is specified, it is treated as if the precision were omitted. If only "." is specified, the precision is assumed to be 0. If the precision appears together with a conversion specification other than the above, the operation is undefined.

(4) size

This is an arbitrary optional size character h, l, ll, or L, which changes the default method for interpreting the data type of the corresponding argument.

When h is specified, a following d, i, o, u, x, or X type specification is forcibly applied to a short or unsigned short argument.

When l is specified, a following d, i, o, u, x, or X type specification is forcibly applied to a long or unsigned long argument. l is also causes a following n type specification to be forcibly applied to a pointer to long argument. If another type specification character is used together with h or l, the operation is undefined.

When ll is specified, a following d, i, o, u, x, or X type specification is forcibly applied to a long long and unsigned long long argument. Furthermore, for ll, a following n type specification is forcibly applied to a long long pointer. If another type specification character is used together with ll, the operation is undefined.

When L is specified, a following e, E, f, g, or G type specification is forcibly applied to a long double argument. If another type specification character is used together with L, the operation is undefined.

(5) type specification character

These are characters that specify the type of conversion that is to be applied.

The characters that specify conversion types and their meanings are as follows.

%	Output the character "%". No argument is converted. The conversion specification is "%%".
c	Convert an int type argument to unsigned char type and output the characters of the conversion result.

d	Convert an int type argument to a signed decimal number.
e, E	Convert a double type argument to [-]d.dddde±dd format, which has one digit before the decimal point (not 0 if the argument is not 0) and the number of digits after the decimal point is equal to the precision. The E conversion specification generates a number in which the exponent part starts with "E" instead of "e".
f	Convert a double type argument to decimal notation of the form [-]dddd.dddd.
g, G	Convert a double type argument to e (E for a G conversion specification) or f format, with the number of digits in the mantissa specified for the precision. Trailing zeros of the conversion result are excluded from the fractional part. The decimal point appears only when it is followed by a digit.
i	Perform the same conversion as d.
n	Store the number of characters that were output in the same object. A pointer to int type is used as the argument.
p	Output a pointer in an implementation-defined format. The CX handles a pointer as unsigned long (this is the same as the lu specification).
o, u, x, X	Convert an unsigned int type argument to octal notation (o), unsigned decimal notation (u), or unsigned hexadecimal notation (x or X) with dddd format. For x conversion, the letters abcdef are used. For X conversion, the letters ABCDEF are used.
s	The argument must be a pointer pointing to a character type array. Characters from this array are output up until the null character (\0) indicating termination (the null character (\0) itself is not included). If the precision is specified, no more than the specified number of characters will be output. If the precision is not specified or if the precision is greater than the size of this array, make sure that this array includes the null character (\0).

[Example]

```
#include    <stdio.h>
void func(int val) {
        char    s[20];
        sprintf(s, "%-10.51x\n", val);  /*Specifies left-justification, field width 10,
                                          precision 5, size long, and hexadecimal notation
                                          for the value of val, and outputs the result
                                          with an appended new-line character to the array
                                          pointed to by s.*/
}
```

fprintf

Output text in specified format to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int fprintf(FILE **stream*, const char **format*[, *arg*, ...]);

[Return value]

The number of characters that were output is returned.

[Description]

This function applies the format specified by the string pointed to by *format* to the respective *arg* arguments, and outputs the formatted data that was output as a result to *stream*. Only the standard input/output stdout or stderr can be specified for *stream*. The method of specifying *format* is the same as described for the sprintf function. However, fprintf differs from sprintf in that no null character (\0) is output at the end.

[Caution]

Stdin (standard input) and stdout (standard error) are specified for the argument *stream*. 1 memory addresses such as an I/O address is allocated for the I/O destination of stream. To use these streams in combination with a debugger, the initial values of the stream structure defined in stdio.h must be set. Be sure to set the initial values prior to calling the function.

[Definition of stream structure in stdio.h]

```
typedef struct {
        int         mode;   /*with error descriptions*/
        unsigend    handle;
        int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

The first structure member, mode, indicates the I/O status and is internally defined as ACCSD_OUT/ADDSD_IN. The third member, unget_c, indicates the pushed-back character (stdin only) setting and is internally defined as -1.

When the definition is -1, it indicates that there is no pushed-back character. The second member, handle, indicates the I/O address. Set the value according to the debugger to be used.

Example I/O address setting

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

[Example]

```
#include    <stdio.h>
void func(int val) {
        fprintf(stdout, "%-10.5x\n", val);
}
/*Example using vfprintf in a general error reporting routine*/
void error(char *function_name, char *format, ...) {
        va_list arg;
        va_start(arg, format);
        fprintf(stderr, "ERROR in %s:", function_name); /*Output function name for which
                                                          error occurred*/
        vfprintf(stderr, format, arg);                 /*Output remaining messages*/
        va_end(arg);
}
```

vsprintf

Write text in specified format to character string

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

[Return value]

The number of characters that were output (excluding the null character (\0)) is returned.
Error return does not occur.

[Description]

This function applies the format specified by the string pointed to by *format* to the argument string pointed to by *arg*, and outputs the formatted data that was output as a result to the array pointed to be *s*. The vsprintf function is equivalent to sprintf with the list of a variable number of real arguments replaced by *arg*. *arg* must be initialized by the va_start macro before the vsprintf function is called.

printf

Output text in specified format to standard output stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int printf(const char **format*[, *arg*, ...]);

[Return value]

The number of characters that were output is returned.

[Description]

This function applies the format specified by the string pointed to by *format* to the respective *arg* arguments, and outputs the formatted data that was output as a result to the standard input/output stdout. The method of specifying *format* is the same as described for the sprintf function. However, printf differs from sprintf in that no null character (\0) is output at the end.

[Caution]

Stdin (standard input) and stdout (standard error) are specified for the argument *stream*. 1 memory addresses such as an I/O address is allocated for the I/O destination of stream. To use these streams in combination with a debugger, the initial values of the stream structure defined in stdio.h must be set. Be sure to set the initial values prior to calling the function.

[Definition of stream structure in stdio.h]

```
typedef struct {
        int         mode;   /*with error descriptions*/
        unsigend    handle;
        int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

The first structure member, mode, indicates the I/O status and is internally defined as ACCSD_OUT/ADDSD_IN. The third member, unget_c, indicates the pushed-back character (stdin only) setting and is internally defined as -1.

When the definition is -1, it indicates that there is no pushed-back character. The second member, handle, indicates the I/O address. Set the value according to the debugger to be used.

Example I/O address setting

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

vfprintf

Write text in specified format to stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int vfprintf(FILE *_stream_, const char *_format_, va_list _arg_);

[Return value]

The number of characters that were output is returned.

[Description]

This function applies the format specified by the string pointed to by *format* to argument string pointed to by *arg*, and outputs the formatted data that was output as a result to *stream*. Only the standard input/output stdout or stderr can be specified for *stream*. The method of specifying *format* is the same as described for the sprintf function. The vfprintf function is equivalent to fprintf with the list of a variable number of real arguments replaced by *arg*. *arg* must be initialized by the va_start macro before the vfprintf function is called.

[Caution]

Stdin (standard input) and stdout (standard error) are specified for the argument *stream*. 1 memory addresses such as an I/O address is allocated for the I/O destination of stream. To use these streams in combination with a debugger, the initial values of the stream structure defined in stdio.h must be set. Be sure to set the initial values prior to calling the function.

[Definition of stream structure in stdio.h]

```
typedef struct {
        int         mode;   /*with error descriptions*/
        unsigend    handle;
        int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

The first structure member, mode, indicates the I/O status and is internally defined as ACCSD_OUT/ADDSD_IN. The third member, unget_c, indicates the pushed-back character (stdin only) setting and is internally defined as -1.

When the definition is -1, it indicates that there is no pushed-back character. The second member, handle, indicates the I/O address. Set the value according to the debugger to be used.

Example I/O address setting

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

[Example]

```
#include    <stdio.h>
void func(int val) {
        fprintf(stdout, "%-10.5x\n", val);
}
/*Example using vfprintf in a general error reporting routine*/
void error(char *function_name, char *format, ...) {
        va_list arg;
        va_start(arg, format);
        fprintf(stderr, "ERROR in %s:", function_name); /*Output function name for which
                                                          error occurred*/
        vfprintf(stderr, format, arg);                 /*Output remaining messages*/
        va_end(arg);
}
```

vprintf

Write text in specified format to standard output stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int vprintf(const char *_format_, va_list _arg_);

[Return value]

The number of characters that were output is returned.

[Description]

This function applies the format specified by the string pointed to by *format* to the argument string pointed to by *arg*, and outputs the formatted data that was output as a result to the standard input/output stdout. The method of specifying *format* is the same as described for the sprintf function. The vprintf function is equivalent to printf with the list of a variable number of real arguments replaced by *arg*. *arg* must be initialized by the va_start macro before the vprintf function is called.

[Caution]

Stdin (standard input) and stdout (standard error) are specified for the argument *stream*. 1 memory addresses such as an I/O address is allocated for the I/O destination of stream. To use these streams in combination with a debugger, the initial values of the stream structure defined in stdio.h must be set. Be sure to set the initial values prior to calling the function.

[Definition of stream structure in stdio.h]

```
typedef struct {
        int         mode;   /*with error descriptions*/
        unsigend    handle;
        int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

The first structure member, mode, indicates the I/O status and is internally defined as ACCSD_OUT/ADDSD_IN. The third member, unget_c, indicates the pushed-back character (stdin only) setting and is internally defined as -1.

When the definition is -1, it indicates that there is no pushed-back character. The second member, handle, indicates the I/O address. Set the value according to the debugger to be used.

Example I/O address setting

```
__struct_stdout.handle = 0xFFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

sscanf

nput with format

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int sscanf(const char *s, const char *format[, arg, ...]);

[Return value]

The number of input fields for which scanning, conversion, and storage were executed normally is returned. The return value does not include scanned fields that were not stored. If an attempt is made to read to the end of the file, the return value is EOF. If no field was stored, the return value is 0.

[Description]

This function reads the input to be converted according to the *format* specified by the character string pointed to by format from the array pointed to by *s* and treats the *arg* arguments that follow format as pointers that point to objects for storing the converted input.

An input string that can be recognized and "the conversion that is to be performed for assignment" are specified for *format*. If sufficient arguments do not exist for *format*, the operation is undefined. If *format* is used up even when arguments remain, the remaining arguments are ignored.

The *format* consists of the following three types of directives:

One or more Space characters	Space ( ), tab (\t), or new-line (\n). If a space character is found in the string when sscanf is executed, all consecutive space characters are read until the next non-space character appears (the space characters are not stored).
Ordinary characters	All ASCII characters other than "%". If an ordinary character is found in the string when sscanf is executed, that character is read but not stored. sscanf reads a string from the input field, converts it into a value of a specific type, and stores it at the position specified by the argument, according to the conversion specification. If an explicit match does not occur according to the conversion specification, no subsequent space character is read.
Conversion specification	Fetches 0 or more arguments and directs the conversion.

Each conversion specification starts with "%". The following appear after the "%":

%[assignment-suppression-character][field-width][size][type-specification-character]

Each conversion specification is explained below.

(1) Assignment suppression character

The assignment suppression character "*" suppresses the interpretation and assignment of the input field.

(2) field width

This is a non-zero decimal integer that defines the maximum field width.

It specifies the maximum number of characters that are read before the input field is converted. If the input field is smaller than this field width, sscanf reads all the characters in the field and then proceeds to the next field and its conversion specification.

If a space character or a character that cannot be converted is found before the number of characters equivalent to the field width is read, the characters up to the white space or the character that cannot be converted are read and stored. Then, sscanf proceeds to the next conversion specification.

(3) size

This is an arbitrary optional size character h, l, ll, or L, which changes the default method for interpreting the data type of the corresponding argument.

When h is specified, a following d, i, n, o, u, or x type specification is forcibly converted to short int type and stored as short type. Nothing is done for c, e, f, n, p, s, D, I, O, U, or X.

When l is specified, a following d, i, n, o, u, or x type specification is forcibly converted to long int type and stored as long type. An e, f, or g type specification is forcibly converted to double type and stored as double type. Nothing is done for c, n, p, s, D, I, O, U, and X.

When ll is specified, a following d, i, o, u, x, or X type specification is forcibly converted to long long type and stored as long long type. Nothing is done for other type specifications.

When L is specified, a following e, f, or g type specification is forcibly converted to long double type and stored as long double type. Nothing is done for other type specifications.

In cases other than the above, the operation is undefined.

(4) type specification character

These are characters that specify the type of conversion that is to be applied.

The characters that specify conversion types and their meanings are as follows.

%	Match the character "%". No conversion or assignment is performed. The conversion specification is "%%".
c	Scan one character. The corresponding argument should be "char **arg*".
d	Read a decimal integer into the corresponding argument. The corresponding argument should be "int **arg*".
e, f, g	Read a floating-point number into the corresponding argument. The corresponding argument should be "float **arg*".
i	Read a decimal, octal, or hexadecimal integer into the corresponding argument. The corresponding argument should be "int **arg*".
n	Store the number of characters that were read in the corresponding argument. The corresponding argument should be "int **arg*".
o	Read an octal integer into the corresponding argument. The corresponding argument must be "int **arg*".
p	Store the pointer that was scanned. This is an implementation definition. The ca processes %p and %U in exactly the same manner. The corresponding argument should be "void ***arg*".
s	Read a string into a given array. The corresponding argument should be "char *arg*[ ]".
u	Read an unsigned decimal integer into the corresponding argument. The corresponding argument should be "unsigned int **arg*".
x, X	Read a hexadecimal integer into the corresponding argument. The corresponding argument should be "int **arg*".

D	Read a decimal integer into the corresponding argument. The corresponding argument should be "long *arg".
E, F, G	Read a floating-point number into the corresponding argument. The corresponding argument should be "double *arg".
I	Read a decimal, octal, or hexadecimal integer into the corresponding argument. The corresponding argument should be "long *arg".
O	Read an octal integer into the corresponding argument. The corresponding argument should be "long *arg".
U	Read an unsigned decimal integer into the corresponding argument. The corresponding argument should be "unsigned long *arg".
[ ]	Read a non-empty string into the memory area starting with argument *arg*. This area must be large enough to accommodate the string and the null character (\0) that is automatically appended to indicate the end of the string. The corresponding argument should be "char *arg". The character pattern enclosed by [ ] can be used in place of the type specification character s. The character pattern is a character set that defines the search set of the characters constituting the input field of sscanf. If the first character within [ ] is "^", the search set is complemented, and all ASCII characters other than the characters within [ ] are included. In addition, a range specification feature that can be used as a shortcut is also available. For example, %[0-9] matches all decimal numbers. In this set, "-" cannot be specified as the first or last character. The character preceding "-" must be less in lexical sequence than the succeeding character. - %[abcd] Matches character strings that include only a, b, c, and d. - %[^abcd] Matches character strings that include any characters other than a, b, c, and d. - %[A-DW-Z] Matches character strings that include A, B, C, D, W, X, Y, and Z. - %[z-a] Matches z, -, and a (this is not considered a range specification).

Make sure that a floating-point number (type specification characters e, f, g, E, F, and G) corresponds to thefollowing general format.

[+ | -] ddddd [.] ddd [E | e [+ | -] ddd]

However, the portions enclosed by [ ] in the above format are arbitrarily selected, and ddd indicates a decimal digit.

[Caution]

- sscanf may stop scanning a specific field before the normal end-of-field character is reached or may stop completely.
- sscanf stops scanning and storing a field and moves to the next field under the following conditions.
  - The substitution suppression character (*) appears after "%" in the format specification, and the input field at that point has been scanned but not stored.
  - A field width (positive decimal integer) specification character was read.
  - The character to be read next cannot be converted according to the conversion specification (for example, if Z is read when the specification is a decimal number).
  - The next character in the input field does not appear in the search set (or appears in the complement search set).

  If sscanf stops scanning the input field at that point because of any of the above reasons, it is assumed that the next character has not yet been read, and this character is used as the first character of the next field or the first character for the read operation to be executed after the input.
- sscanf ends under the following conditions:
  - The next character in the input field does not match the corresponding ordinary character in the string to be converted.
  - The next character in the input field is EOF.
  - The string to be converted ends.
- If a list of characters that is not part of the conversion specification is included in the string to be converted, make sure that the same list of characters does not appear in the input. sscanf scans matching characters but does not store them. If there was a mismatch, the first character that does not match remains in the input as if it were not read.

[Example]

```
#include    <stdio.h>
void func(void) {
        int         i, n;
        float       x;
        const char  *s;
        char        name[10];
        s = "23 11.1e-1 NAME";
        n = sscanf(s,"%d%f%s", &i, &x, name);   /*Stores 23 in i, 1.110000 in x, and "NAME"
                                                    in name.  The return value n is 3.*/
}
```

fscanf

Read and interpret data from stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format[, arg, ...]);
```

[Return value]

The number of input fields for which scanning, conversion, and storage were executed normally is returned. The return value does not include scanned fields that were not stored. If an attempt is made to read to the end of the file, the return value is EOF. If no field was stored, the return value is 0.

[Description]

Reads the input to be converted according to the format specified by the character string pointed to by *format* from *stream* and treats the *arg* arguments that follow format as objects for storing the converted input. Only the standard input/output stdin can be specified for *stream*. The method of specifying *format* is the same as described for the sscanf function.

scanf

Read and interpret text from standard output stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

#include <stdio.h>
int scanf(const char **format*[, *arg*, ...]);

[Return value]

The number of input fields for which scanning, conversion, and storage were executed normally is returned. The return value does not include scanned fields that were not stored. If an attempt is made to read to the end of the file, the return value is EOF. If no field was stored, the return value is 0.

[Description]

Reads the input to be converted according to the format specified by the character string pointed to by *format* from the standard input/output stdin and treats the *arg* arguments that follow format as objects for storing the converted input. The method of specifying *format* is the same as described for the sscanf function.

[Example]

```
#include    <stdio.h>
void func(void) {
        int     i, n;
        double  x;
        char    name[10];
        n = scanf("%d%lf%s", &i, &x, name); /*Perform formatted input of input from stdin
                                              using the format "23 11.1e-1 NAME"*/
}
```

ungetc

Push character back to input stream

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

[Return value]

The character *c* is returned.
Error return does not occur.

[Description]

This function pushes the character *c* back into the input stream pointed to by *stream*. However, if *c* is EOF, no pushback is performed. The character *c* that was pushed back will be input as the first character during the next character input. Only one character can be pushed back by ungetc. If ungetc is executed continuously, only the last ungetc will have an effect. Only the standard input/output stdin can be specified for *stream*.

rewind

Reset file position indicator

Remark These functions are not supported by the debugging functions which CubeSuite provides.

[Classification]

Standard library

[Syntax]

```
#include <stdio.h>
void    rewind(FILE *stream);
```

[Description]

This function clears the error indicator of the input stream pointed to by *stream*, and positions the file position indicator at the beginning of the file.

However, only the standard input/output stdin can be specified for *stream*. Therefore, rewind only has the effect of discarding the character that was pushed back by ungetc.

perror

Error processing

[Classification]

Standard library

[Syntax]

#include <stdio.h>
void perror(const char *s);

[Description]

This function outputs to stderr the error message that corresponds to global variable errno.
The message that is output is as follows.

When *s* is not NULL	`fprintf(stderr, "%s:%s\n", s, s_fix);`
When *s* is NULL	`fprintf(stderr, "%s\n", s_fix);`

s_fix is as follows.

When errno is EDOM	"EDOM error"
When errno is ERANGE	"ERANGE error"
When errno is 0	"no error"
Otherwise	"error *xxx*" (*xxx* は abs (errno) % 1000)

[Example]

```
#include    <stdio.h>
void func(double x) {
        double  d;
        errno = 0;
        d = exp(x);
        if(errno)
                perror("func1");    /*If a calculation exception is generated by exp perror
                                      is called*/
}
```

6.4.7 Standard utility functions

Standard Utility functions are as follows.

Table 6-24. Standard Utility Functions

Function/Macro Name	Outline
abs	Output absolute value (int type)
labs	Output absolute value (long type)
llabs	Output absolute value (long long type)
bsearch	Binary search
qsort	Sort
div	Division (int type)
ldiv	Division (long type)
lldiv	Division (long long type)
itoa	Conversion of integer (int type) to character string
ltoa	Conversion of integer (long type) to character string
ultoa	Conversion of integer (unsigned long type) to character string
lltoa	Conversion of integer (long long type) to character string
ulltoa	Conversion of integer (unsigned long long type) to character string
ecvt	Conversion of floating-point value to numeric character string (with total number of characters specified)
ecvtf	Conversion of floating-point value to numeric character string (with total number of characters specified)
fcvt	Conversion of floating-point value to numeric character string (with total number of characters specified)
fcvtf	Conversion of floating-point value to numeric character string (with number of digits below decimal point specified)
gcvt	Conversion of floating-point value to numeric character string (in specified format)
gcvtf	Conversion of floating-point value to numeric character string (in specified format)
atoi	Conversion of character string to integer (int type)
atol	Conversion of character string to integer (long type)
atoll	Conversion of character string to integer (long long type)
strtol	Conversion of character string to integer (long type) and storing pointer in last character string
strtoul	Conversion of character string to integer (unsigned long type) and storing pointer in last character string
strtoll	Conversion of character string to integer (long long type) and storing pointer in last character string
strtoull	Conversion of character string to integer (unsigned long long type) and storing pointer in last character string
atoff	Conversion of character string to floating-point number (float type)
atof	Conversion of character string to floating-point number (double type)
strtodf	Conversion of character string to floating-point number (float type) (storing pointer in last character string)

Function/Macro Name	Outline
strtod	Conversion of character string to floating-point number (double type) (storing pointer in last character string
calloc	Memory allocation (initialized to zero)
malloc	Memory allocation(not initialized to zero)
realloc	Memory re-allocation
free	Memory release
rand	Pseudorandom number sequence generation
srand	Setting of type of pseudorandom number sequence

abs

Output absolute value (int type)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
int abs(int *j*);

[Return value]

Returns the absolute value of *j* (size of *j*), | *j* |.

[Description]

This function obtains the absolute value of *j* (size of *j*), | *j* |. If *j* is a negative number, the result is the reversal of j. If *j* is not negative, the result is *j*.

[Example]

```
#include    <stdlib.h>
void func(int l) {
        int val;
        val = -15;
        l = abs(val);   /*Returns absolute value of val, 15, to 1.*/
}
```

labs

Output absolute value (long type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
long    labs(long j);
```

[Return value]

Returns the absolute value of *j* (size of *j*), | *j* |.

[Description]

This function obtains the absolute value of *j* (size of *j*), | *j* |. If *j* is a negative number, the result is the reversal of j. If *j* is not negative, the result is *j*. This function is the same as abs, but uses long type instead of int type, and the return value is also of long type.

llabs

Output absolute value (long long type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
long long      llabs(long long j);
```

[Return value]

Returns the absolute value of *j* (size of *j*), | *j* |.

[Description]

This function obtains the absolute value of *j* (size of *j*), | *j* |. If *j* is a negative number, the result is the reversal of j. If *j* is not negative, the result is *j*. This function is the same as abs, but uses long long type instead of int type, and the return value is also of long long type.

bsearch

Binary search

[Classification]

Standard library

[Syntax]

#include <stdlib.h>

void* bsearch(const void *_key_, const void *_base_, size_t _nmemb_, size_t _size_, int (*_compar_)(const void *, const void*));

[Return value]

A pointer to the element in the array that coincides with *key* is returned. If there are two or more elements that coincide with *key*, the one that has been found first is indicated. If there are not elements that coincide with *key*, a null pointer is returned.

[Description]

This function searches an element that coincides with *key* from an array starting with *base* by means of binary search. *nmemb* is the number of elements of the array. *size* is the size of each element. The array must be arranged in the ascending order in respect to the compare function indicated by *compar* (last argument). Define the compare function indicated by *compar* to have two arguments. If the first argument is less than the second, a negative integer must be returned as the result. If the two arguments coincide, zero must be returned. If the first is greater than the second, a positive integer must be returned.

[Example]

```
#include    <stdlib.h>
#include    <string.h>
int compar(char **x, char **y);

void func(void) {
        static  char    *base[] = {"a", "b", "c", "d", "e", "f"};
        char            *key = "c";     /*Search key is "c".*/
        char            **ret;
                                        /*Pointer to "c" is stored in ret.*/
        ret = (char **) bsearch((char *) &key, (char *) base, 6, sizeof(char *), compar);
}
int compar(char **x, char **y) {
        return(strcmp(*x, *y));          /*Returns positive, zero, or negative integer as
                                           result of comparing arguments.*/
}
```

qsort

Sort

[Classification]

Standard library

[Syntax]

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void*, const void *));

[Description]

This function sorts the array pointed to by *base* into ascending order in relation to the comparison function pointed to by *compar*. *nmemb* is the number of array elements, and *size* is the size of each element. The comparison function pointed to by *compar* is the same as the one described for bsearch.

div

Division (int type)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
div_t div(int *n*, int *d*);

[Return value]

The structure storing the result of the division is returned.

[Description]

This function is used to divide a value of int type

This function calculates the quotient and remainder resulting from dividing numerator *n* by denominator *d*, and stores these two integers as the members of the following structure div_t.

```
typedef struct {
        int quot;
        int rem;
} div_t;
```

quot the quotient, and rem is the remainder. If *d* is not zero, and if "*r* = div(*n*, *d*);", *n* is a value equal to "*r*.rem + *d* * *r*.quot".

If *d* is zero, the resultant quot member has a sign the same as *n* and has the maximum size that can be expressed. The rem member is 0.

[Example]

```
#include    <stdlib.h>
void func(void) {
        div_t   r;
        r = div(110, 3);    /*36 is stored in r.quot, and 2 is stored in r.rem.*/
}
```

ldiv

Division (long type)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
ldiv_t ldiv(long *n*, long *d*);

[Return value]

The structure storing the result of the division is returned.

[Description]

This function is used to divide a value of long type.

This function calculates the quotient and remainder resulting from dividing numerator *n* by denominator *d*, and stores these two integers as the members of the following structure div_t.

```
typedef struct {
        long    quot;
        long    rem;
} ldiv_t;
```

quot the quotient, and rem is the remainder. If *d* is not zero, and if "*r* = div(*n*, *d*);", *n* is a value equal to "*r*.rem + *d* * *r*.quot".

If *d* is zero, the resultant quot member has a sign the same as *n* and has the maximum size that can be expressed. The rem member is 0.

lldiv

Division (long long type)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
lldiv_t lldiv(long long *n*, long long *d*);

[Return value]

The structure storing the result of the division is returned.

[Description]

This function is used to divide a value of long long type.

This function calculates the quotient and remainder resulting from dividing numerator *n* by denominator *d*, and stores these two integers as the members of the following structure div_t.

```
typedef struct {
        long long   quot;
        long long   rem;
} lldiv_t;
```

quot the quotient, and rem is the remainder. If *d* is not zero, and if "*r* = div(*n*, *d*);", *n* is a value equal to "*r*.rem + *d* * *r*.quot".

If *d* is zero, the resultant quot member has a sign the same as *n* and has the maximum size that can be expressed. The rem member is 0.

itoa

Conversion of integer (int type) to character string

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
char *itoa(int *value*, char **string*, int *radix*);

[Return value]

string is returned.

[Description]

This function converts an int type numeric *value* to a character string for a *radix*-based number and stores it in the array indicated by *string*. The terminating null character (\0) always is added at the end of the character string. Numeric values from 2 to 36 can be specified for *radix*. If *radix* is 10, *value* is handled as a signed numeric value, and when *value* < 0, the "-" character is appended at the beginning of the character string. Otherwise, *value* is handled as an unsigned numeric value. If *radix* > 10, the lowercase letters a to z are assigned for 10 to 35.

[Example]

```
#include    <stdlib.h>
void func(void) {
        char    buf[128];
        itoa(12345, buf, 16);   /*Converts 12345 to a hexadecimal character string*/
}
```

ltoa

Conversion of integer (long type) to character string

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *ltoa(long int value, char *string, int radix);
```

[Return value]

string is returned.

[Description]

This function converts a long int type numeric *value* to a character string for a *radix*-based number and stores it in the array indicated by *string*. Except for the type of *value*, this is the same as itoa.

ultoa

Conversion of integer (unsigned long type) to character string

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *ultoa(unsigned long int value, char *string, int radix);
```

[Return value]

string is returned.

[Description]

This function converts an unsigned long int type numeric *value* to a character string for a *radix*-based number and stores it in the array indicated by *string*. Except for the type of *value*, this is the same as itoa.

lltoa

Conversion of integer (long long type) to character string

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *lltoa(long long int value, char *string, int radix);
```

[Return value]

string is returned.

[Description]

This function converts a long long int type numeric *value* to a character string for a *radix*-based number and stores it in the array indicated by *string*. Except for the type of *value*, this is the same as itoa.

ulltoa

Conversion of integer (unsigned long long type) to character string

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *ulltoa(unsigned long long int value, char *string, int radix);
```

[Return value]

string is returned.

[Description]

This function converts a unsigned long long int type numeric *value* to a character string for a *radix*-based number and stores it in the array indicated by *string*. Except for the type of *value*, this is the same as itoa.

ecvt

Conversion of floating-point value to numeric character string (with total number of characters specified)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *ecvt(double val, int chars, int *decpt, int *sgn);
```

[Return value]

Returns a pointer indicating a new character string including the character string representation of *val*.

[Description]

This function generates a character string indicating a numeric value *val* of double type in number (terminated with the null character (\0)). The second argument *chars* specifies the total number of characters to be written (because only numbers are written, this argument specifies the valid number of numerals in the converted character string). The digits of the integer of *val* are always included.

ecvtf

Conversion of floating-point value to numeric character string (with total number of characters specified)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
char *ecvtf(float *val*, int *chars*, int **decpt*, int **sgn*);

[Return value]

Returns a pointer indicating a new character string including the character string representation of *val*.

[Description]

This function generates a character string indicating a numeric value *val* of float type in number (terminated with the null character (\0)). The second argument *chars* specifies the total number of characters to be written (because only numbers are written, this argument specifies the valid number of numerals in the converted character string). The digits of the integer of *val* are always included.

[Example]

```
#include    <stdlib.h>
void func(void) {
        float   val;
        int     dec, sgn;
        val = 111.11;
       ecvtf(val, 12, &dec, &sgn); /*Converts value 111.11 of val to character string of 12
                                    characters.  dec records number of digits, 3, at left
                                    of decimal point,  and sgn records sign(0 because
                                    numeric value is positive).*/
}
```

fcvt

Conversion of floating-point value to numeric character string (with total number of characters specified)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *fcvt(double val, int decimals, int *decpt, int *sgn);
```

[Return value]

Returns a pointer indicating a new character string including the character string representation of *val*.

[Description]

This function is the same as ecvt, except the interpretation of the second argument. The second argument *decimals* specify the number of characters to be written after the decimal point. ecvt and fcvtf only write a number to an output character string. Therefore, record the position of the decimal point to **decpt* and the sign of the numeric value to **sgn*. After the number has been formatted, the number of digits at the left of the decimal point is stored in **decpt*. If the numeric value is positive, 0 is stored in **sgn*; if it is negative, 1 is stored.

fcvtf

Conversion of floating-point value to numeric character string (with number of digits below decimal point specified)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *fcvtf(float val, int decimals, int *decpt, int *sgn);
```

[Return value]

Returns a pointer indicating a new character string including the character string representation of *val*.

[Description]

This function is the same as ecvtf, except the interpretation of the second argument. The second argument *decimals* specify the number of characters to be written after the decimal point. ecvtf and fcvtf only write a number to an output character string. Therefore, record the position of the decimal point to **decpt* and the sign of the numeric value to **sgn*. After the number has been formatted, the number of digits at the left of the decimal point is stored in **decpt*. If the numeric value is positive, 0 is stored in **sgn*; if it is negative, 1 is stored.

gcvt

Conversion of floating-point value to numeric character string (in specified format)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
char *gcvtf(double *val*, int *prec*, char **buf*);

[Return value]

Returns a pointer (same as argument *buf*) to the formatted character string representation of *val*.

[Description]

This function converts a numeric value into a character string, and stores it to buffer *buf*. gcvtf uses the same rule as the format "%.*prec*" (sign is appended to the negative number only) of sprintf, and selects an exponent format or normal decimal point format according to the valid number of digits (specified by *prec*).

gcvtf

Conversion of floating-point value to numeric character string (in specified format)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
char    *gcvtf(float val, int prec, char *buf);
```

[Return value]

Returns a pointer (same as argument *buf*) to the formatted character string representation of *val*.

[Description]

This function converts a numeric value into a character string, and stores it to buffer *buf*. gcvtf uses the same rule as the format "%.*prec*" (sign is appended to the negative number only) of sprintf, and selects an exponent format or normal decimal point format according to the valid number of digits (specified by *prec*).

atoi

Conversion of character string to integer (int type)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
int atoi(const char *_str_);

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.

[Description]

This function converts the first part of the character string indicated by *str* into an int type representation. atoi is the same as "(int) strtol (*str*, NULL, 10)".

atol

Conversion of character string to integer (long type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
long    atol(const char *str);
```

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.

[Description]

This function converts the first part of the character string indicated by *str* into a long int type representation. atol is the same as "strtol (*str*, NULL, 10)".

atoll

Conversion of character string to integer (long long type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
long long   atoll(const char *str);
```

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.

[Description]

This function converts the first part of the character string indicated by *str* into a long long int type representation. atol is the same as "strtol (*str*, NULL, 10)".

strtol

Conversion of character string to integer (long type) and storing pointer in last character string

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
long strtol(const char **str*, char ***ptr*, int *base*);

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.
If an overflow occurs (because the converted value is too great), LONG_MAX or LONG_MIN is returned, and macro ERANGE is set to global variable errno.

[Description]

This function converts the first part of the character string indicated by *str* into a long type representation. strol first divides the input characters into the following three parts: the "first blank", "a string represented by the *base* number determined by the value of base and is subject to conversion into an integer", and "the last one or more character string that is not recognized (including the null character (\0))". Then strtol converts the string into an integer, and returns the result.

(1) Specify 0 or 2 to 36 as argument *base*.

(a) If *base* is 0
The expected format of the character string subject to conversion is of integer format having an optional + or - sign and "0x", indicating a hexadecimal number, prefixed.

(b) If the value of *base* is 2 to 36
The expected format of the character string is of character string or numeric string type having an optional + or - sign prefixed and expressing an integer whose base is specified by base. Characters "a" (or "A") through "z" (or "Z") are assumed to have a value of 10 to 35. Only characters whose value is less than that of base can be used.

(c) If the value of *base* is 16
"0x" is prefixed (suffixed to the sign if a sign exists) to the string of characters and numerals (this can be omitted).

(2) The string subject to conversion is defined as the longest partial string at the beginning of the input character string that starts with the first character other than blank and has an expected format.

(a) If the input character string is vacant, if it consists of blank only, or if the first character that is not blank is not a sign or a character or numeral that is permitted, the subject string is vacant.

(b) If the string subject to conversion has an expected format and if the value of *base* is 0, the base number is judged from the input character string. The character string led by 0x is regarded as a hexadecimal value, and the character string to which 0 is prefixed but x is not is regarded as an octal number. All the other character strings are regarded as decimal numbers.

(c) If the value of *base* is 2 to 36, it is used as the base number for conversion as mentioned above.

(d) If the string subject to conversion starts with a - sign, the sign of the value resulting from conversion is reversed.

(3) The pointer that indicates the first character string

(a) This is stored in the object indicated by *ptr*, if *ptr* is not a null pointer.

(b) If the string subject conversion is vacant, or if it does not have an expected format, conversion is not executed. The value of *str* is stored in the object indicated by *ptr* if ptr is not a null pointer.

Remark This function is not re-entrant

[Example]

```
#include    <stdlib.h>
void func(long ret) {
        char    *p;
        ret = strtol("10", &p, 0);      /*10 is returned to ret.*/
        ret = strtol("0x10", &p, 0);    /*16 is returned to ret.*/
        ret = strtol("10x", &p, 2);     /*2 is returned to ret, and pointer to "x" is
                                          returned to area of p.*/
        ret = strtol("2ax3", &p, 16);   /*42 is returned to ret, and pointer to "x" is
                                          returned to area of p.*/
          :
}
```

strtoul

Conversion of character string to integer (unsigned long type) and storing pointer in last character string

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
unsigned long strtoul(const char **str*, char ***ptr*, int *base*);

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.
If an overflow occurs, ULONG_MAX is returned, and macro ERANGE is set to global variable errno.

[Description]

This function is the same as strtol except that the type of the return value is of unsigned long type.

strtoll

Conversion of character string to integer (long long type) and storing pointer in last character string

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
long long strtoll(const char **str*, char ***ptr*, int *base*);

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.
If an overflow occurs (the converted value is too larger), LLONG_MAX or LLONG_MIN is returned, and macro ERANGE is set to global variable errno.

[Description]

This function is the same as strtol except that the type of the return value is of long long type.

strtoull

Conversion of character string to integer (unsigned long long type) and storing pointer in last character string

[Classification]

Standard library

[Syntax]

#include <stdlib.h>

unsigned long long strtoull(const char **str*, char ***ptr*, int *base*);

[Return value]

Returns the converted value if the partial character string could be converted. If it could not, 0 is returned.

If an overflow occurs, ULLONG_MAX is returned, and macro ERANGE is set to global variable errno.

[Description]

This function is the same as strtol except that the type of the return value is of unsigned long long type.

atoff

Conversion of character string to floating-point number (float type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
float   atoff(const char *str);
```

[Return value]

If the partial character string has been converted, the resultant value is returned. If the character string could not be converted, 0 is returned.

If an overflow occurs (the value is not in the range in which it can be expressed), HUGE_VAL or -HUGE_VAL is returned, and ERANGE is set to global variable errno. If an underflow occurs, 0 is returned, and macro ERANGE is set to global variable errno.

[Description]

This function converts the first portion of the character string indicated by *str* into a float type representation. atoff is the same as "strtodf (*str*, NULL)".

atof

Conversion of character string to floating-point number (double type)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
double  atof(const char *str);
```

[Return value]

If the partial character string has been converted, the resultant value is returned. If the character string could not be converted, 0 is returned.

If an overflow occurs (the value is not in the range in which it can be expressed), HUGE_VAL or -HUGE_VAL is returned, and ERANGE is set to global variable errno. If an underflow occurs, 0 is returned, and macro ERANGE is set to global variable errno.

[Description]

This function converts the first portion of the character string indicated by *str* into a float type representation. atoff is the same as "strtod (*str*, NULL)".

strtodf

Conversion of character string to floating-point number (float type) (storing pointer in last character string)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
float strtodf(const char **str*, char ***ptr*);

[Return value]

If the partial character string has been converted, the resultant value is returned. If the character string could not be converted, 0 is returned. If an overflow occurs (the value is not in the range in which it can be expressed), HUGE_VAL or -HUGE_VAL is returned, and ERANGE is set to global variable errno. If an underflow occurs, 0 is returned, and macro ERANGE is set to global variable errno.

[Description]

This function converts the first part of the character string indicated by *str* into a float type representation. The part of the character string to be converted is in the following format and is at the beginning of *str* with the maximum length, starting with a normal character that is not a space.

[+ | -] digits [.] [digits] [(e | E) [+ | -] digits]

If *str* is vacant or consists of space characters only, if the first normal character is other than "+", "-", ".", or a numeral, the partial character string does not include a character. If the partial character string is vacant, conversion is not executed, and the value of str is stored in the area indicated by *ptr*. If the partial character string is not vacant, it is converted, and a pointer to the last character string (including the null character (\0) indicating at least the end of str) is stored in the area indicated by *ptr*.

Remark This function is not re-entrant.

[Example]

```
#include    <stdlib.h>
#include    <stdio.h>
void func(float ret) {
        char    *p, *str, s[30];
        str = "+5.32a4e";
        ret = strtodf(str, &p);         /*5.320000 is returned to ret, and pointer to "a"
                                          is stored in area of p.*/
        sprintf(s, "%lf\t%c", ret, *p); /*"5.320000 a" is stored in array indicated by s.*/
}
```

strtod

Conversion of character string to floating-point number (double type) (storing pointer in last character string)

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
double  strtod(const char *str, char **ptr);
```

[Return value]

If the partial character string has been converted, the resultant value is returned. If the character string could not be converted, 0 is returned.

If an overflow occurs (the value is not in the range in which it can be expressed), HUGE_VAL or -HUGE_VAL is returned, and ERANGE is set to global variable errno. If an underflow occurs, 0 is returned, and macro ERANGE is set to global variable errno.

[Description]

This function converts the first part of the character string indicated by *str* into a float type representation. The part of the character string to be converted is in the following format and is at the beginning of *str* with the maximum length, starting with a normal character that is not a space.

[+ | -] digits [.] [digits] [(e | E) [+ | -] digits]

If *str* is vacant or consists of space characters only, if the first normal character is other than "+", "-", ".", or a numeral, the partial character string does not include a character. If the partial character string is vacant, conversion is not executed, and the value of str is stored in the area indicated by *ptr*. If the partial character string is not vacant, it is converted, and a pointer to the last character string (including the null character (\0) indicating at least the end of str) is stored in the area indicated by *ptr*.

Remark This function is not re-entrant.

calloc

Memory allocation (initialized to zero)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
void *calloc(size_t *nmemb*, size_t *size*);

[Return value]

When area allocation succeeds, a pointer to that area is returned. When the area could not be allocated, a null pointer is returned.

[Description]

This function allocates an area for an array of *nmemb* elements. The allocated area is initialized to zeros.

[Caution]

The memory area management functions automatically allocate memory area as necessary from the heap memory area.

Also, the size of the default is 0x1000 bytes, so when it's changed, the heap memory area must be allocated. The area allocation should be performed first by an application.

[Heap memory setup example]

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

Remarks 1. The symbol "___sysheap" (three underscores "_") of the variable "_sysheap" (two underscores "_") points to the starting address of heap memory. This value should be a word integer value.

2. The required heap memory size (bytes) should be set for the variable "__sizeof_sysheap" (two leading underscores). If assembly language is used for coding, this value should be set for the symbol "___sizeof_sysheap" (three leading underscores).

[Example]

```
#include    <stdlib.h>
typedef struct {
        double  d[3];
        int     i[2];
} s_data;
int func(void) {
        sdata   *buf;
        if((buf = calloc(40, sizeof(s_data))) == NULL)  /*Allocate an area for 40 s_data*/
                return(1);
           :
        free(buf);                                      /*Release the area*/
        return(0);
}
```

malloc

Memory allocation(not initialized to zero)

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
void *malloc(size_t *size*);

[Return value]

When area allocation succeeds, a pointer to that area is returned. When the area could not be allocated, a null pointer is returned.

[Description]

This function allocates an area having a size indicated by *size*. The area is not initialized.

[Caution]

The memory area management functions automatically allocate memory area as necessary from the heap memory area.

Also, the size of the default is 0x1000 bytes, so when it's changed, the heap memory area must be allocated. The area allocation should be performed first by an application.

[Heap memory setup example]

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

Remarks 1. The symbol "___sysheap" (three underscores "_") of the variable "_sysheap" (two underscores "_") points to the starting address of heap memory. This value should be a word integer value.

2. The required heap memory size (bytes) should be set for the variable "__sizeof_sysheap" (two leading underscores). If assembly language is used for coding, this value should be set for the symbol "___sizeof_sysheap" (three leading underscores).

realloc

Memory re-allocation

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
void *realloc(void **ptr*, size_t *size*);

[Return value]

When area allocation succeeds, a pointer to that area is returned. When the area could not be allocated, a null pointer is returned.

[Description]

This function changes the size of the area pointed to by *ptr* to the size indicated by *size*. The contents of the area are unchanged up to the smaller of the previous size and the specified *size*. If the area is expanded, the contents of the area greater than the previous size are not initialized. When *ptr* is a null pointer, the operation is the same as that of malloc (*size*). Otherwise, the area that was acquired by calloc, malloc, or realloc must be specified for *ptr*.

[Caution]

The memory area management functions automatically allocate memory area as necessary from the heap memory area.

Also, the size of the default is 0x1000 bytes, so when it's changed, the heap memory area must be allocated. The area allocation should be performed first by an application.

[Heap memory setup example]

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

Remarks 1. The symbol "___sysheap" (three underscores "_") of the variable "_sysheap" (two underscores "_") points to the starting address of heap memory. This value should be a word integer value.

2. The required heap memory size (bytes) should be set for the variable "__sizeof_sysheap" (two leading underscores). If assembly language is used for coding, this value should be set for the symbol "___sizeof_sysheap" (three leading underscores).

free

Memory release

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
void free(void *ptr);

[Description]

This function releases the area pointed to by *ptr* so that this area is subsequently available for allocation. The area that was acquired by calloc, malloc, or realloc must be specified for *ptr*.

[Example]

```
#include    <stdlib.h>
typedef struct {
        double  d[3];
        int     i[2];
} s_data;
int func(void) {
        sdata   *buf;
        if((buf = calloc(40, sizeof(s_data))) == NULL)  /*Allocate an area for 40 s_data*/
                return(1);
           :
        free(buf);                                      /*Release the area*/
        return(0);
}
```

rand

Pseudorandom number sequence generation

[Classification]

Standard library

[Syntax]

#include <stdlib.h>
int rand(void);

[Return value]

Random numbers are returned.

[Description]

This function returns a random number that is greater than or equal to zero and less than or equal to RAND_MAX.

[Example]

```
#include    <stdlib.h>
void func(void) {
        if(rand() & 0xF) < 4)
                func1();    /*Execute func1 with a probability of 25%*/
}
```

srand

Setting of type of pseudorandom number sequence

[Classification]

Standard library

[Syntax]

```
#include <stdlib.h>
void    srand(unsigned int seed);
```

[Description]

This function assigns seed as the new pseudo random number sequence *seed* to be used by the rand call that follows. If srand is called using the same *seed* value, the same numbers in the same order will appear for the random numbers that are obtained by rand. If rand is executed without executing srand, the results will be the same as when srand(1) was first executed.

6.4.8 Non-local jump functions

Non-local jump functions are as follows.

Table 6-25. Non-Local Jump Functions

Function/Macro Name	Outline
longjmp	Non-local jump
setjmp	Set destination of non-local jump

longjmp

Non-local jump

[Classification]

Standard library

[Syntax]

#include <setjmp.h>
void longjmp(jmp_buf *env*, int *val*);

[Return value]

The second argument *val* is returned. However, 1 is returned if *val* is 0.

[Description]

This function performs a non-local jump to the place immediately after setjmp using *env* saved by setjmp. *val* as a return value for setjmp.

[Caution]

When -Xreg_mode=common has been specified, setjmp and longjmp perform the same operation as -Xreg_mode=32 specified. Therefore even if the value of r20 to r24 is changed after a setjmp calling, return to the value before the setjmp calling after the longjmp calling.

[Example]

```
#include    <setjmp.h>
#define ERR_XXX1    1
jmp_buf jmp_env;

void func(void) {
        for(;;) {
                switch(setjmp(jmp_env)) {
                        case    ERR_XXX1:   /*Termination of error XXX1*/
                                break;
                        case    0:          /*No non-local jumps*/
                        default:
                                break;
                }
        }
}

void func1(void) {
        longjmp(jmp_env, ERR_XXX1); /*Non-local jumps are performed upon generation of
                                      error XXX1*/
}
```

setjmp

Set destination of non-local jump

[Classification]

Standard library

[Syntax]

#include <setjmp.h>
int setjmp(jmp_buf *env*);

[Return value]

0 is returned.

[Description]

This function sets *env* as the destination for a non-local jump. In addition, the environment in which setjmp was run is saved to *env*.

[Caution]

When -Xreg_mode=common has been specified, setjmp and longjmp perform the same operation as -Xreg_mode=32 specified. Therefore even if the value of r20 to r24 is changed after a setjmp calling, return to the value before the setjmp calling after the longjmp calling.

6.4.9 Mathematical functions

Mathematical functions are as follows.

Table 6-26. Mathematical Functions

Function/Macro Name	Outline
j0f	Bessel function of first kind (0 order)
j1f	Bessel function of first kind (1 order)
jnf	Bessel function of first kind (*n* order)
y0f	Bessel function of second kind (0 order)
y1f	Bessel function of second kind (1 order)
ynf	Bessel function of second kind (*n* order)
erff	Error function (approximate value)
erfcf	Error function (complementary probability)
expf	Exponent function
exp	Exponent function
logf	Logarithmic function (natural logarithm)
log	Logarithmic function (natural logarithm)
log2f	Logarithmic function (base = 2)
log10f	Logarithmic function (base = 10)
log10	Logarithmic function (base = 10)
powf	Power function
pow	Power function
sqrtf	Square root function
sqrt	Square root function
cbrtf	Cubic root function
cbrt	Cubic root function
ceilf	ceiling function
ceil	ceiling function
fabsf	Absolute value function
fabs	Absolute value function
floorf	floor function
floor	floor function
fmodf	Remainder function
fmod	Remainder function
frexpf	Divide floating-point number into mantissa and power
frexp	Divide floating-point number into mantissa and power
ldexpf	Convert floating-point number to power
ldexp	Convert floating-point number to power
modff	Divide floating-point number into integer and decimal

Function/Macro Name	Outline
modf	Divide floating-point number into integer and decimal
gammaf	Logarithmic gamma function
hypotf	Euclidean distance function
matherrf (matherr)	Error processing function
matherrd	Error processing function
cosf	Cosine
cos	Cosine
sinf	Sine
sin	Sine
tanf	Tangent
tan	Tangent
acosf	Arc cosine
acos	Arc cosine
asinf	Arc sine
asin	Arc sine
atanf	Arc tangent
atan	Arc tangent
atan2f	Arc tangent (y / x)
atan2	Arc tangent (y / x)
coshf	Hyperbolic cosine
cosh	Hyperbolic cosine
sinhf	Hyperbolic sine
sinh	Hyperbolic sine
tanhf	Hyperbolic tangent
tanh	Hyperbolic tangent
acoshf	Arc hyperbolic cosine
asinhf	Arc hyperbolic sine
atanhf	Arc hyperbolic tangent

j0f

Bessel function of first kind (0 order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float j0f(float *x*);

[Return value]

Returns the Bessel function of the first kind of the 0 degree.

[Description]

This function calculates the Bessel functions of the first kind of the 0 degrees.

j1f

Bessel function of first kind (1 order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float j1f(float *x*);

[Return value]

Returns the Bessel function of the first kind of the first degree.

[Description]

This function calculates the Bessel functions of the first kind of the first degrees.

Remark If the solution is a denormal number, j1f sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(void) {
        float   ret, x;
        ret = j1f(x);   /*Calculates Bessel function of first kind and first decree in
                          response to value of x, and returns function to ret.*/
          :
        return(ret);
}
```

jnf

Bessel function of first kind (*n* order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float jnf(int *n*, float *x*);

[Return value]

Returns the Bessel function of the first kind of the *n* degree.

[Description]

This function calculates the Bessel function of the first kind of the *n* degree.

Remark If the absolute value of *n* is bigger than 3000, jnf returns a Not a Nuber(NaN) and sets macro ERANGE to global variable errno.
If the solution is a denormal number, jnf sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

y0f

Bessel function of second kind (0 order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float y0f(float *x*);

[Return value]

Returns the Bessel function of the second kind of the 0 degree.

[Description]

This function calculates the Bessel functions of the second kind of the 0 degrees.

Remark If inputting zero, y0f returns -∞ and sets macro ERANGE to global variable errno.
If inputting the negative number, y0f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

y1f

Bessel function of second kind (1 order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float y1f(float *x*);

[Return value]

Returns the Bessel function of the second kind of the first degree.

[Description]

This function calculates the Bessel functions of the second kind of the first degrees.

Remark If inputting zero, y1f returns +∞ and sets macro ERANGE to global variable errno.
If inputting the negative number, y1f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

ynf

Bessel function of second kind (*n* order)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float ynf(int *n*, float *x*);

[Return value]

Returns the Bessel function of the second kind of the *n* degree.

[Description]

This function calculates the Bessel function of the second kind of the *n* degree.

Remark If *x* is zero, ynf returns -∞ and sets macro ERANGE to global variable errno.
If *x* is the negative number, ynf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If the absolute value of *n* is bigger than 3000, ynf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If overflow occurred, ynf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

erff

Error function (approximate value)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float erff(float *x*);

[Return value]

Returns the approximate value (numeric value between 0 and 1) of the "error function".

[Description]

This function calculates the approximate value (numeric value between 0 and 1) of the "error function" that estimates the probability for which the observed value is in a range of standard deviation *x*.

Remark If the solution is a denormal number, erff sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(void) {
        float   ret, x;
        ret = erff(x);  /*Calculates approximate value of error function in response to
                          value of x and returns it to ret.*/
          :
        return(ret);
}
```

erfcf

Error function (complementary probability)

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   erfcf(float x);
```

[Return value]

Returns the complementary probability.

[Description]

This function calculates complementary probability through "1.0-erff(*x*)". This function is provided to prevent the accuracy from dropping if erff(*x)* is called by *x* with a large value and the result is subtracted from 1.0.

Remark If the solution is a denormal number, erfcf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

expf

Exponent function

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   expf(float x);
```

[Return value]

Returns the *x*th power of e.

expf returns an denormal number if an underflow occurs (if *x* is a negative number that cannot express the result), and sets macro ERANGE to global variable errno. If an overflow occurs (if *x* is too great a number), HUGE_VAL (maximum double type numerics that can be expressed) is returned, and macro ERANGE is set to global variable errno.

[Description]

This function calculates the *x*th power of e (e is the base of a natural logarithm and is about 2.71828).

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

exp

Exponent function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double exp(double *x*);

[Return value]

Returns the *x*th power of e.

expf returns an denormal number if an underflow occurs (if *x* is a negative number that cannot express the result), and sets macro ERANGE to global variable errno. If an overflow occurs (if *x* is too great a number), HUGE_VAL (maximum double type numerics that can be expressed) is returned, and macro ERANGE is set to global variable errno.

[Description]

This function calculates the *x*th power of e (e is the base of a natural logarithm and is about 2.71828).

Remark The error processing of this function can be changed by using the matherrd function.

logf

Logarithmic function (natural logarithm)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float logf(float *x*);

[Return value]

Returns the natural logarithm of *x*.
logf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is negative. If *x* is zero, it returns -∞ and sets macro ERANGE to global variable errno.

[Description]

This function calculates the natural logarithm of *x*, i.e., logarithm with base e.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

log

Logarithmic function (natural logarithm)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double log(double *x*);

[Return value]

Returns the natural logarithm of *x*.
logf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is negative. If *x* is zero, it returns -∞ and sets macro ERANGE to global variable errno.

[Description]

This function calculates the natural logarithm of *x*, i.e., logarithm with base e.

Remark The error processing of this function can be changed by using the matherrd function.

log2f

Logarithmic function (base = 2)

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   log2f(float x);
```

[Return value]

Returns the logarithm of *x* with base 2.

log2f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is negative. If *x* is zero, it returns -∞ and sets macro ERANGE to global variable errno.

[Description]

This function calculates the logarithm of *x* with base 2. This is realized by "log (*x)* / log (2)".

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

log10f

Logarithmic function (base = 10)

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   log10f(float x);
```

[Return value]

Returns the logarithm of *x* with base 10.

log10f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is negative. If *x* is zero, it returns -∞ and sets macro ERANGE to global variable errno.

[Description]

This function calculates the logarithm of *x* with base 10. This is realized by "log (*x*) / log (10)".

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

log10

Logarithmic function (base = 10)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double log10(double *x*);

[Return value]

Returns the logarithm of *x* with base 10.
log10f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is negative. If *x* is zero, it returns -∞ and sets macro ERANGE to global variable errno.

[Description]

This function calculates the logarithm of *x* with base 10. This is realized by "log (*x*) / log (10)".

Remark The error processing of this function can be changed by using the matherrd function.

powf

Power function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float powf(float *x*, float *y*);

[Return value]

Returns the *y*th power of *x*.

powf returns a negative solution only if *x* < 0 and *y* is an odd integer. If *x* < 0 and *y* is a non-integer or if *x* = *y* = 0, powf returns a Not a Nuber(NaN) and sets the macro EDOM for the global variable errno. If *x* = 0 and *y* < 0 or if an overflow occurs, powf returns ±HUGE_VAL and sets the macro ERANGE for errno. If the solution vanished approaching zero, powf returns 0 and sets the macro ERANGE for errno. If the solution is a denormal number, powf sets the macro ERANGE for errno.

[Description]

This function calculates the *y*th power of *x*.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(void) {
        float   ret, x, y;
        ret = powf(x, y);   /*Returns yth power of x to ret.*/
          :
        return(ret);
}
```

pow

Power function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double pow(double *x*, double *y*);

[Return value]

Returns the *y*th power of *x*.

powf returns a negative solution only if $x < 0$ and *y* is an odd integer. If $x < 0$ and *y* is a non-integer or if $x = y = 0$, powf returns a Not a Nuber(NaN) and sets the macro EDOM for the global variable errno. If $x = 0$ and $y < 0$ or if an overflow occurs, powf returns ±HUGE_VAL and sets the macro ERANGE for errno. If the solution vanished approaching zero, powf returns 0 and sets the macro ERANGE for errno. If the solution is a denormal number, powf sets the macro ERANGE for errno.

[Description]

This function calculates the *y*th power of *x*.

Remark The error processing of this function can be changed by using the matherrd function.

sqrtf

Square root function

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   sqrtf(float x);
```

[Return value]

Returns the positive square root of *x*.
sqrtf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is a negative real number.

[Description]

This function calculates the square root of *x*.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

[Caution]

If the device has an V850E2V3 FPU, then enabling optimization generates a sqrtf.s instruction instead of calling a library function. This will not change the setting of the global variable "errno", or the error processing of the matherrf (matherr) function.
Specify the "-Xcall_lib" option to call the library function.

sqrt

Square root function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double sqrt(double *x*);

[Return value]

Returns the positive square root of *x*.
sqrtf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if *x* is a negative real number.

[Description]

This function calculates the square root of *x*.

Remark The error processing of this function can be changed by using the matherrd function.

cbrtf

Cubic root function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float cbrtf(float *x*);

[Return value]

Returns the cubic root of *x*.

[Description]

This function calculates the cubic root of *x*.

cbrt

Cubic root function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double cbrt(double *x*);

[Return value]

Returns the cubic root of *x*.

[Description]

This function calculates the cubic root of *x*.

ceilf

ceiling function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float ceilf(float *x*);

[Return value]

Returns the minimum integer greater than *x* and *x*.

[Description]

This function calculates the minimum integer value greater than *x* and *x*.

ceil

ceiling function

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
double  ceil(double x);
```

[Return value]

Returns the minimum integer greater than *x* and *x*.

[Description]

This function calculates the minimum integer value greater than *x* and *x*.

fabsf

Absolute value function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float fabsf(float *x*);

[Return value]

Returns the absolute value (size) of *x*.

[Description]

This function calculates the absolute value (size) of *x* by directly manipulating the bit representation of *x*.

fabs

Absolute value function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double fabs(double *x*);

[Return value]

Returns the absolute value (size) of *x*.

[Description]

This function calculates the absolute value (size) of *x* by directly manipulating the bit representation of *x*.

floorf

floor function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float floorf(float *x*);

[Return value]

Returns the maximum integer value less than *x* and *x*.

[Description]

This function calculates the maximum integer value less than *x* and *x*.

floor

floor function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float floorf(float *x*);

[Syntax]

#include <math.h>
double floor(double *x*);

[Return value]

Returns the maximum integer value less than *x* and *x.*

[Description]

This function calculates the maximum integer value less than *x* and *x*.

fmodf

Remainder function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float fmodf(float *x*, float *y*);

[Return value]

Returns a floating-point value that is the remainder resulting from dividing *x* by *y*.
fmodf (*x*, 0) returns *x*.

[Description]

This function calculates a floating-point value that is the remainder resulting from dividing *x* by *y*. In other words, it calculates the value "*x* - i * *y*" for the maximum integer i that has a sign the same as *x* and is less than *y*, if *y* is not zero.

Remark If *x* is ±∞ or *y* is zero, fmodf returns a Not a Nuber(NaN) and sets macro ERANGE to global variable errno. The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
void func(void) {
        float   ret, x, y;
        ret = fmodf(x, y);  /*Returns remainder resulting from dividing x by y to ret.*/
          :
}
```

fmod

Remainder function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double fmod(double *x*, double *y*);

[Return value]

Returns a floating-point value that is the remainder resulting from dividing *x* by *y*.
fmod (*x*, 0) returns *x*.

[Description]

This function calculates a floating-point value that is the remainder resulting from dividing *x* by *y*. In other words, it calculates the value "*x* - i * *y*" for the maximum integer i that has a sign the same as *x* and is less than *y*, if *y* is not zero.

Remark If *x* is ±∞ or *y* is zero, fmod returns a Not a Nuber(NaN) and sets macro ERANGE to global variable errno. The error processing of this function can be changed by using the matherrd function.

frexpf

Divide floating-point number into mantissa and power

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float frexpf(float *val*, int **exp*);

[Return value]

Returns mantissa m.
frexpf sets 0 to **exp* and returns 0 if *val* is 0.

[Description]

This function expresses *val* of float type as mantissa m and the pth power of 2. The resulting mantissa m is 0.5 <= | x | < 1.0, unless *val* is zero. p is stored in **exp*. m and p are calculated so that *val* = m * 2^{p}.

Remark If *val* is ±∞, frexpf returns zero and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
void func(void) {
        float   ret, x;
        int     exp;
        x = 5.28;
        ret = frexpf(x, &exp);  /*Resultant mantissa 0.66 is returned to ret, and 3 is
                                  stored in exp*/
          :
}
```

frexp

Divide floating-point number into mantissa and power

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double frexp(double *val*, int **exp*);

[Return value]

Returns mantissa m.
frexpf sets 0 to **exp* and returns 0 if *val* is 0.

[Description]

This function expresses *val* of double type as mantissa m and the pth power of 2. The resulting mantissa m is 0.5 <= | x | < 1.0, unless *val* is zero. p is stored in **exp*. m and p are calculated so that *val* = m * 2^{p}.

Remark If *val* is ±∞, frexpf returns zero and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

ldexpf

Convert floating-point number to power

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float ldexpf(float *val*, int *exp*);

[Return value]

Returns the value calculated by*val* x 2 ^{exp}.
If an underflow or overflow occurs as a result of executing ldexpf, macro ERANGE is set to global variable errno. If an underflow occurs, ldexpf returns an denormal number. If an overflow occurs, it returns HUGE_VAL.

[Description]

This function calculates *val* x 2 ^{exp}.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

ldexp

Convert floating-point number to power

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double ldexp(double *val*, int *exp*);

[Return value]

Returns the value calculated by$val \times 2^{exp}$.
If an underflow or overflow occurs as a result of executing ldexpf, macro ERANGE is set to global variable errno. If an underflow occurs, ldexpf returns an denormal number. If an overflow occurs, it returns HUGE_VAL.

[Description]

This function calculates $val \times 2^{exp}$.

Remark The error processing of this function can be changed by using the matherrd function.

modff

Divide floating-point number into integer and decimal

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float modff(float *val*, float **ipart*);

[Return value]

Returns a decimal part. The sign of the result is the same as the sign of *val*.

[Description]

This function divides *val* of float type into integer and decimal parts, and stores the integer part in **ipart*. Rounding is not performed. It is guaranteed that the sum of the integer part and decimal part accurately coincides with *val*. For example, where *realpart* = modff (*val*, *&intpart*), "*realpart* + *intpart*t" coincides with *val*.

modf

Divide floating-point number into integer and decimal

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double modf(double *val*, double **ipart*);

[Return value]

Returns a decimal part. The sign of the result is the same as the sign of *val*.

[Description]

This function divides *val* of double type into integer and decimal parts, and stores the integer part in **ipart*. Rounding is not performed. It is guaranteed that the sum of the integer part and decimal part accurately coincides with *val*. For example, where *realpart* = modff (*val*, *&intpart*), "*realpart* + *intpart*t" coincides with *val*.

gammaf

Logarithmic gamma function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float gammaf(float *x*);

[Return value]

The natural logarithm of the gamma function of *x* is returned.
If *x* is 0 or an overflow occurs, HUGE_VAL is returned, and macro ERANGE is set to global variable errno.

[Description]

This function calculates ln (Γ (x)), i.e., the natural logarithm of the gamma function of *x*. The gamma function (expf (gammaf (*x*)) is a generalized factorial, and has a relational expression of Γ (N) = N x Γ (N - 1). Therefore, the result of the gamma function itself increases very rapidly. Consequently, gammaf is defined as "ln (Γ (*x*))", instead of simply "Γ (*x*)", to expand the valid range of the result that can be expressed.

Remark If inputting the negative number, gammaf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(float x) {
        float   ret;
        ret = gammaf(x);    /*Returns natural logarithm of gamma function of x to ret.*/
          :
        return(ret);
}
```

hypotf

Euclidean distance function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float hypotf(float *x*, float *y*);

[Return value]

Returns a Euclidean distance "sqrt ($x^2 + y^2$)" between the origin (0, 0) and a point indicated by Cartesian coordinates (*x*, *y*).

If an overflow occurs, HUGE_VAL is returned, and macro ERANGE is set to global variable errno.

[Description]

This function calculates a Euclidean distance "sqrt ($x^2 + y^2$)" between the origin (0, 0) and a point indicated by Cartesian coordinates (*x*, *y*).

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
void func(float x) {
        float   ret, y;
        ret = hypotf(x, y); /*Returns Euclidean distance between origin (0, 0) and
                              coordinates (x, y) to ret.*/
}
```

matherrf (matherr)

Error processing function

Remark "matherr" can be used as "matherrf".

[Classification]

Mathematical library

[Syntax]

#include <math.h>
int matherrf(struct exceptionf *e);

[Return value]

By changing the value of e ->retval, the result of the function called from the customized matherrf can be changed. This also applies to the function on the calling side. The matherrf returns a value other than 0 if the error has been resolved, and 0 if the error could not be resolved. If matherrf returns 0, set an appropriate value to global variable errono on the calling side.

[Description]

This is a function that is called if an error occurs in a mathematical library function.

By preparing a function named matherrf via a user subroutine, therefore, error processing can be customized. Customized matherrf must return 0 if resolution of an error has failed, and a value other than 0 if the error has been resolved. If matherrf returns a value other than 0, the value of global variable errno is not changed.

Error processing can be customized by using the information passed by pointer *e to structure exceptionf. Structure exceptionf is defined as follows in "math.h".

```
#define __exceptionf exceptionf
struct  __exceptionf {
        int             type;
        const char      *name;
        float           arg1, arg2, retval;
};
```

The meaning of each member is as follows:

type	Type of mathematical function error that has occurred. The type of the macro encoding error is also defined in "math.h".
name	Pointer indicating a character string that holds the name of the mathematical library function in which an error has occurred, and ends with a space character.
arg1, arg2	Arguments responsible for the error.
retval	Error return value that is returned by the calling function.

The types of mathematical library function errors that may occur are as follows.

DOMAIN	The argument is not in the range of the definition area of the function Example: logf (-1);
OVERFLOW	Overflow Example: expf (1000);
INEXACT	Annihilation of solution toward 0 Example: exp (-1000);
UNDERFLOW	Underflow, solutions to denormal number. Solution < 1.1755e-38 and non 0 and precision is lower than the normal value.
Z_DIVISION	Zero division.

Remark Calling matherr when an operation exception occurs and updating global variable errno with a standard function are not re-entrant.

[Caution]

When -Xreg_mode=common has been specified, runtime functions perform the same operation as -Xreg_mode=32 specified. Therefore even if the value of r15 to r19 is changed in matherrf when an exception occurs, it isn't changed by the program to which the runtime function was called.

[Example]

```
#include    <math.h>
#include    <stdio.h>
void func(void) {
        float   ret;
        ret = logf(-0.1);               /*3 is returned to ret.*/
}
int matherrf(struct exceptionf *e) {
        char    s[30];
        switch(e->type) {
                case    DOMAIN:
                        sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
                        e->retval = 3;          /*Changes error return value to 3.*/
                        break;
                default:
                        sprintf(s, "%s other error %e\n", e->name, e->arg1);
        }
        return(1);
}
```

matherrd

Error processing function

[Classification]

Mathematical library

[Syntax]

#include <math.h>
int matherrd(struct exceptiond *e);

[Return value]

By changing the value of e ->retval, the result of the function called from the customized matherrd can be changed. This also applies to the function on the calling side. The matherrd returns a value other than 0 if the error has been resolved, and 0 if the error could not be resolved. If matherrd returns 0, set an appropriate value to global variable errono on the calling side.

[Description]

This is a function that is called if an error occurs in a mathematical library function.

By preparing a function named matherrd via a user subroutine, therefore, error processing can be customized. Customized matherrd must return 0 if resolution of an error has failed, and a value other than 0 if the error has been resolved. If matherrd returns a value other than 0, the value of global variable errno is not changed.

Error processing can be customized by using the information passed by pointer *e to structure exceptiond. Structure exceptiond is defined as follows in "math.h".

```
#define __exceptiond exceptiond
#endif
struct  __exceptiond {
        int             type;
        const char      *name;
        double          arg1, arg2, retval;
};
```

The meaning of each member is as follows:

type	Type of mathematical function error that has occurred. The type of the macro encoding error is also defined in "math.h".
name	Pointer indicating a character string that holds the name of the mathematical library function in which an error has occurred, and ends with a space character.
arg1, arg2	Arguments responsible for the error.
retval	Error return value that is returned by the calling function.

The types of mathematical library function errors that may occur are as follows.

DOMAIN	The argument is not in the range of the definition area of the function Example: logf (-1);
OVERFLOW	Overflow Example: expf (1000);
INEXACT	Annihilation of solution toward 0 Example: exp (-1000);
UNDERFLOW	Underflow, solutions to denormal number. Solution < 1.1755e-38 and non 0 and precision is lower than the normal value.
Z_DIVISION	Zero division.

Remark Calling matherr when an operation exception occurs and updating global variable errno with a standard function are not re-entrant.

[Caution]

When -Xreg_mode=common has been specified, runtime functions perform the same operation as -Xreg_mode=32 specified. Therefore even if the value of r15 to r19 is changed in matherrd when an exception occurs, it isn't changed by the program to which the runtime function was called.

[Example]

```
#include    <math.h>
#include    <stdio.h>
void func(void) {
        float   ret;
        ret = logf(-0.1);               /*3 is returned to ret.*/
}
int matherrd(struct exceptiond *e) {
        char    s[30];
        switch(e->type) {
                case    DOMAIN:
                        sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
                        e->retval = 3;          /*Changes error return value to 3.*/
                        break;
                default:
                        sprintf(s, "%s other error %e\n", e->name, e->arg1);
        }
        return(1);
}
```

cosf

Cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float cosf(float *x*);

[Return value]

Returns the cosine of *x*.

[Description]

This function calculates the cosine of *x*. Specify the angle in radian.

Remark If inputting ±∞, cosf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

cos

Cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double cos(double *x*);

[Return value]

Returns the cosine of *x*.

[Description]

This function calculates the cosine of *x*. Specify the angle in radian.

Remark If inputting ±∞, cos returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
The error processing of this function can be changed by using the matherrd function.

sinf

Sine

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   sinf(float x);
```

[Return value]

Returns the sine of *x*.

[Description]

This function calculates the sine of *x*. Specify the angle in radian.

Remark If inputting ±∞, sinf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If the solution is a denormal number, sinf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

sin

Sine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double sin(double *x*);

[Return value]

Returns the sine of *x*.

[Description]

This function calculates the sine of *x*. Specify the angle in radian.

Remark If inputting ±∞, sin returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If the solution is a denormal number, sin sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrd function.

tanf

Tangent

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   tanf(float x);
```

[Return value]

Returns the tangent of *x*.

[Description]

This function calculates the cosine of *x*. Specify the angle in radian.

Remark If inputting ±∞, tanf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If the solution is a denormal number, tanf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

tan

Tangent

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double tan(double *x*);

[Return value]

Returns the tangent of *x*.

[Description]

This function calculates the cosine of *x*. Specify the angle in radian.

Remark If inputting ±∞, tan returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno.
If the solution is a denormal number, tan sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrd function.

acosf

Arc cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float acosf(float *x*);

[Return value]

Returns the arc cosine of *x*. The returned value is in radian and in a range of 0 to π.
If *x* is not between -1 and 1, a Not a Nuber(NaN) is returned, and macro EDOM is set to global variable errno.

[Description]

This function calculates the arc cosine of *x*. Specify *x* as, -1<= *x* <= 1.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

acos

Arc cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double acos(double *x*);

[Return value]

Returns the arc cosine of *x*. The returned value is in radian and in a range of 0 to π.
If *x* is not between -1 and 1, a Not a Nuber(NaN) is returned, and macro EDOM is set to global variable errno.

[Description]

This function calculates the arc cosine of *x*. Specify *x* as, -1<= *x* <= 1.

Remark The error processing of this function can be changed by using the matherrd function.

asinf

Arc sine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float asinf(float *x*);

[Return value]

Returns the arc sine (arcsine) of *x*. The returned value is in radian and in a range of -π / 2 to π / 2.
If *x* is not between -1 and 1, a Not a Nuber(NaN) is returned, and macro EDOM is set to global variable errno.

[Description]

This function calculates the arc sine (arcsine) of *x*. Specify *x* as, -1 <= *x* <= 1.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

asin

Arc sine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double asin(double *x*);

[Return value]

Returns the arc sine (arcsine) of *x*. The returned value is in radian and in a range of -π / 2 to π / 2.
If *x* is not between -1 and 1, a Not a Nuber(NaN) is returned, and macro EDOM is set to global variable errno.

[Description]

This function calculates the arc sine (arcsine) of *x*. Specify *x* as, -1 <= *x* <= 1.

Remark The error processing of this function can be changed by using the matherrd function.

atanf

Arc tangent

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float atanf(float *x*);

[Return value]

Returns the arc tangent (arctangent) of *x*. The returned value is in radian and in a range of -π / 2 to π / 2.

[Description]

This function calculates the arc tangent (arctangent) of *x*. Specify *x* as, -1 <= *x* <= 1.

Remark If the solution is a denormal number, atanf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(float x) {
        float   ret;
        ret = atanf(x); /*Returns value of arctangent of x to ret.*/
          :
        return(ret);
}
```

atan

Arc tangent

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double atan(double *x*);

[Return value]

Returns the arc tangent (arctangent) of *x*. The returned value is in radian and in a range of -π / 2 to π / 2.

[Description]

This function calculates the arc tangent (arctangent) of *x*. Specify *x* as, -1 <= *x* <= 1.

Remark If the solution is a denormal number, atan sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrd function.

atan2f

Arc tangent (y / x)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float atan2f(float *y*, float *x*);

[Return value]

Returns the arc tangent (arctangent) of *y* / *x*. The returned value is in radian and in a range of -π to π.
atan2f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if both *x* and *y* are 0.0. If the solution vanished approaching zero, atan2f returns ±0 and sets macro ERANGE to global variable errno. If the solution is a denormal number, atan2f sets macro ERANGE to global variable errno.

[Description]

This function calculates the arc tangent of *y* / *x*. atan2f calculates the correct result even if the angle is in the vicinity of π / 2 or - π / 2 (if *x* is close to 0).

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

atan2

Arc tangent (y / x)

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double atan2(double *y*, double *x*);

[Return value]

Returns the arc tangent (arctangent) of *y* / *x*. The returned value is in radian and in a range of -π to π.
atan2f returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if both *x* and *y* are 0.0. If the solution vanished approaching zero, atan2f returns ±0 and sets macro ERANGE to global variable errno. If the solution is a denormal number, atan2f sets macro ERANGE to global variable errno.

[Description]

This function calculates the arc tangent of *y* / *x*. atan2f calculates the correct result even if the angle is in the vicinity of π / 2 or - π / 2 (if *x* is close to 0).

Remark The error processing of this function can be changed by using the matherrd function.

coshf

Hyperbolic cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float coshf(float *x*);

[Return value]

Returns the hyperbolic cosine of *x*.
coshf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.

[Description]

This function calculates the hyperbolic cosine of *x*. Specify the angle in radian. The definition expression is as follows.

$$(e^{x} + e^{-x}) / 2$$

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

cosh

Hyperbolic cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double cosh(double *x*);

[Return value]

Returns the hyperbolic cosine of *x*.
coshf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.

[Description]

This function calculates the hyperbolic cosine of *x*. Specify the angle in radian. The definition expression is as follows.

$$(e^{x} + e^{-x}) / 2$$

Remark The error processing of this function can be changed by using the matherrd function.

sinhf

Hyperbolic sine

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   sinhf(float x);
```

[Return value]

Returns the hyperbolic sine of *x*.
sinhf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.

[Description]

This function calculates the hyperbolic sine of *x*. Specify the angle in radian. The definition expression is as follows.

$$(e^{x} - e^{-x}) / 2$$

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

sinh

Hyperbolic sine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
double sinh(double *x*);

[Return value]

Returns the hyperbolic sine of *x*.
sinhf returns HUGE_VAL and sets macro ERANGE to global variable errno if an overflow occurs.

[Description]

This function calculates the hyperbolic sine of *x*. Specify the angle in radian. The definition expression is as follows.

$$(e^{x} - e^{-x}) / 2$$

Remark The error processing of this function can be changed by using the matherrd function.

tanhf

Hyperbolic tangent

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   tanhf(float x);
```

[Return value]

Returns the hyperbolic tangent of *x*.

[Description]

This function calculates the hyperbolic tangent of *x*. Specify the angle in radian. The definition expression is as follows.

$$\sinh(x) / \cosh(x)$$

Remark If the solution is a denormal number, tanhf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

tanh

Hyperbolic tangent

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
double  tanh(double x);
```

[Return value]

Returns the hyperbolic tangent of *x*.

[Description]

This function calculates the hyperbolic tangent of *x*. Specify the angle in radian. The definition expression is as follows.

$$\sinh(x) / \cosh(x)$$

Remark If the solution is a denormal number, tanh sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrd function.

acoshf

Arc hyperbolic cosine

[Classification]

Mathematical library

[Syntax]

#include <math.h>
float acoshf(float *x*);

[Return value]

Returns the arc hyperbolic cosine of *x* (*x* is a numeric number of 1 or greater).
acoshf returns a Not a Nuber(NaN) if *x* is less than 1. Macro EDOM is set to global variable errno.

[Description]

This function calculates the arc hyperbolic cosine of *x* (where *x* is a numeric value of 1 or greater). The definition expression is as follows.

$$\ln (x + \sqrt{(x^2 - 1)})$$

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

[Example]

```
#include    <math.h>
float func(float x) {
        float   ret;
        ret = acoshf(x);    /*Returns value of arc hyperbolic cosine of x to ret.*/
          :
        return(ret);
}
```

asinhf

Arc hyperbolic sine

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   asinhf(float x);
```

[Return value]

Returns the arc hyperbolic sine of *x*.

[Description]

This function calculates the arc hyperbolic sine of *x*. The definition expression is as follows.

$$\mathrm{sign}(x) * \ln(|x| + \sqrt{1 + x^2})$$

Remark If the solution is a denormal number, asinhf sets macro ERANGE to global variable errno.
The error processing of this function can be changed by using the matherrf (matherr) function.

atanhf

Arc hyperbolic tangent

[Classification]

Mathematical library

[Syntax]

```
#include <math.h>
float   atanhf(float x);
```

[Return value]

Returns the arc hyperbolic tangent of *x*.

atanhf returns a Not a Nuber(NaN) and sets macro EDOM to global variable errno if the absolute value of *x* is greater than 1.

[Description]

This function calculates the arc hyperbolic tangent of *x*.

Remark The error processing of this function can be changed by using the matherrf (matherr) function.

6.4.10 Initialization peripheral devices function

Initialization peripheral devices function are as follows.

Table 6-27. Initialization Peripheral Devices Function

Function/Macro Name	Outline
hdwinit	Initialization of peripheral devices immediately after the CPU reset

hdwinit

Initialization of peripheral devices immediately after the CPU reset.

[Classification]

Initialization library

[Syntax]

```
void    hdwinit(void);
```

[Description]

The initialization peripheral devices function performs initialization of peripheral devices immediately after the CPU reset.

This is called from inside the startup routine.

The function included in the library is a dummy routine that performs no actions; code a function in accordance with your system.

6.4.11 Copy functions

These functions are the routines that copies data and program codes with initial values to RAM.

- A ROMization function itself does not use the sdata area and sbss area. Writes the data to sdata area.
- A ROMization function is usually called only once before the main program is executed. So it does not considers re-entrant.
- When a load module is downloaded to the in-circuit emulator (ICE), the data with initial values and placed in the data area or sdata area is set as soon as the load module has been downloaded.
  Therefore, debugging can be performed without calling the copy function. If a ROMization load module is created and executed on the actual machine, however, the initial values are not set and the operation is not performed as expected unless data with an initial value is copied using the copy function. The reason for the trouble is that an initial value is not set by this copy function. If a routine that clears RAM to zero is executed during initialization, call the copy function before that routine. Otherwise the initial values will also be cleared to zero.

Copy functions are as follows.

Table 6-28. Copy Functions

Function/Macro Name	Outline
_rcopy	Copies packed data to RAM, 1-byte at a time (Same as _rcopy1)
_rcopy1	Copies packed data to RAM, 1-byte at a time (Same as _rcopy)
_rcopy2	Copies packed data to RAM, 2-bytes at a time
_rcopy4	Copies packed data to RAM, 4-bytes at a time

Remarks 1. _rcopy and _rcopy1 perform the same operation.
When a program code is copied to the internal instruction RAM of a V850 device that has an internal instruction RAM (such as the V850E/ME2), it must be copied in 4-byte units because of the hardware specifications. In this case, the program code is copied using the "_rcopy4" function. Any function could be used if no hardware restrictions. When a program code is copied in 2-byte or 4-byte units, the area that must be copied may be exceeded. If the size of a packed data area is not a multiple of 4, therefore, an area other than the packed data area is also copied at the same time. Take this into consideration.

2. See "8.4 Copy Functions" for details of this processing.

6.4.12 Pseudo "main" functions for multi-core

Pseudo "main" functions for multi-core are as follows.

Table 6-29. Pseudo "main" Functions for Multi-core

Function/Macro Name	Outline
main_pen	Does not return control to the caller.

main_pe*n*

It is an infinite loop, and does not return control to the caller.

[Classification]

Multi-core library

[Syntax]

int main_pe*n* (void);

Remark This is a convenience declaration. It allows the user to control the parameters/return value of the startup routine.

[Description]

This is a do-nothing function.

When using a multi-core device, if the user does not provide a main() function for other than PE1, then this will be linked from the multi-core startup routine.

It is an infinite loop internally, and does not return control to the caller.

main_pe2 to main_pe31 are provided.

[Example]

```
        ld.hu   PEID, r10
        cmp     1, r10
        be      .L1
        # Non-PE1 processing
        jarl    main_pe2, lp ;        /* Does not return, because it is an infinite loop */
.L1:
        # PE1 processing continues
```

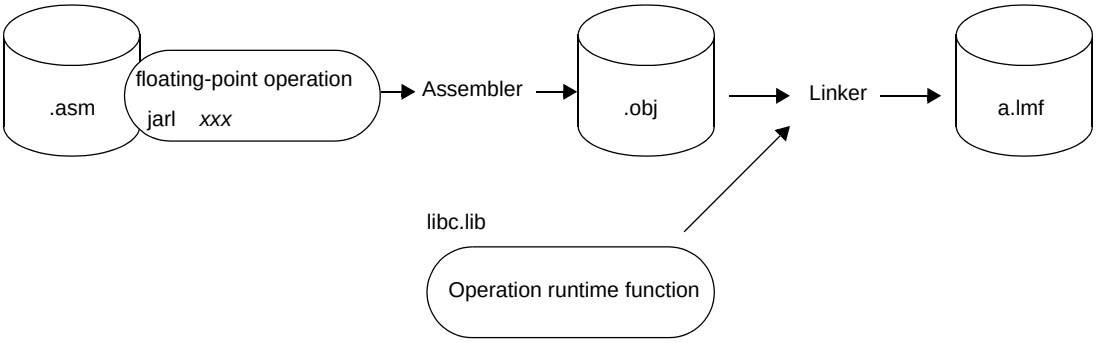
6.4.13 Operation runtime functions

This section explains the operation runtime functions.

The operation runtime function is a routine the CX uses automatically to do calculation on 64-bit data and floating-point operations. This library is included in the libc.lib file along with the standard library. The header file does not need to be included. Similar to "function pre/post processing runtime function", "operation runtime function" is not described in the C source or assembler source.

When using the operation runtime function for an application program, libc.lib must be referred by linker when an executable object module file is created.

Figure 6-1. Image of Using Operation Runtime Function



Operation runtime functions are as follows.

Table 6-30. Operation Runtime Functions

Classification	Function Name	Outline
float type operation function	___addf.s	Addition of single-precision floating-point
	___subf.s	Subtraction of single-precision floating-point
	___mulf.s	Multiplication of single-precision floating-point
	___divf.s	Division of single-precision floating-point
	___cmpf.s	Comparison of single-precision floating-point
	___fcmp.s	Comparison of single-precision floating-point
	___negf.s	Negate of single-precision floating-point
	___notf.s	Logical negation of single-precision floating-point
double type operation function	___addf.d	Addition of double-precision floating-point
	___subf.d	Subtraction of double-precision floating-point
	___mulf.d	Multiplication of double-precision floating-point
	___divf.d	Division of double-precision floating-point
	___fcmp.d	Comparison of double-precision floating-point
	___negf.d	Negate of double-precision floating-point
	___notf.d	Logical negation of double-precision floating-point

Classification	Function Name	Outline
long long type operation function	___add.l	Addition of 64-bit integer
	___sub.l	Subtraction of 64-bit integer
	___mul.l	Multiplication of 64-bit integer
	___div.l	Division of signed 64-bit integer
	___div.ul	Division of unsigned 64-bit integer
	___mod.l	Remainder of signed 64-bit integer
	___mod.ul	Remainder of unsigned 64-bit integer
	___shl.l	Logical left shift of 64-bit integer
	___shr.l	Logical right shift of 64-bit integer
	___sar.l	Arithmetic right shift 64-bit integer
	___inc.l	Increment of 64-bit integer
	___dec.l	Decrement of 64-bit integer
	___not.l	Logical negation of 64-bit integer
	___neg.l	Negate of 64-bit integer
	___cmp.l	Comparison of signed 64-bit integer
	___cmp.ul	Comparison of unsigned 64-bit integer
	___bext.l	Bit field extraction of signed 64-bit integer
	___bext.ul	Bit field extraction of unsigned 64-bit integer
	___bins.l	Bit field insertion of 64-bit integer

Classification	Function Name	Outline
Type conversion function	___cvt.ws	Conversion from 32-bit integer to single-precision floating-point number
	___cvt.wd	Conversion from 32-bit integer to double-precision floating-point number
	___cvt.uws	Conversion from unsigned 32-bit integer to single-precision floating-point number
	___cvt.uwd	Conversion from unsigned 32-bit integer to double-precision floating-point number
	___cvt.ls	Conversion from 64-bit integer to single-precision floating-point number
	___cvt.ld	Conversion from 64-bit integer to double-precision floating-point number
	___cvt.uls	Conversion from unsigned 64-bit integer to single-precision floating-point number
	___cvt.uld	Conversion from unsigned 64-bit integer to double-precision floating-point number
	___trnc.sw	Conversion from single-precision floating-point number to 32-bit integer
	___trnc.dw	Conversion from double-precision floating-point number to 32-bit integer
	___trnc.suw	Conversion from single-precision floating-point number to unsigned 32-bit integer
	___trnc.duw	Conversion from double-precision floating-point number to unsigned 32-bit integer
	___trnc.sl	Conversion from single-precision floating-point number to 64-bit integer
	___trnc.dl	Conversion from double-precision floating-point number to 64-bit integer
	___trnc.sul	Conversion from single-precision floating-point number to unsigned 64-bit integer
	___trnc.dul	Conversion from double-precision floating-point number to unsigned 64-bit integer
	___cvt.sd	Conversion from single-precision floating-point number to double-precision floating-point number
	___cvt.ds	Conversion from double-precision floating-point number to single-precision floating-point number
int type operation function	__mul	Multiplication of signed integer
	__mulu	Multiplication of unsigned integer
	__div	Division of signed integer
	__divu	Division of unsigned integer
	__mod	Remainder of signed integer
	__modu	Remainder of unsigned integer

Remarks 1. The operation runtime function is originally used by code generation part and is not assumed to be used alone. Therefore, preprocessing to call the operation runtime function is necessary when it is used for an assembly- language source program.

2. The operation runtime function cannot be used with a C source program.

___addf.s

Addition of float type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of addition
r6	Right term of addition

[Return value]

r6	Result of addition

[Description]

Addition of single-precision floating-point.

___subf.s

Subtraction of float type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of subtraction
r6	Right term of subtraction

[Return value]

r6	Result of subtraction

[Description]

Subtraction of single-precision floating-point.

___mulf.s

Multiplication of float type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of multiplication
r6	Right term of multiplication

[Return value]

r6	Result of multiplication

[Description]

Multiplication of single-precision floating-point.

___divf.s

Division of float type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of division
r6	Right term of division

[Return value]

r6	Result of division

[Description]

Division of single-precision floating-point

___cmpf.s

Comparison of float type

[Classification]

Runtime library

[Argument(s)]

r8	Value of PSW just before the comparison
r7	Left term of comparison
r6	Right term of comparison

[Return value]

PSW	Result of comparison
r6	Same value as PSW after comparison

[Description]

Comparison of single-precision floating-point.
A result of the following combination is returned.

	Z flag	CY flag	S flag
Left term or right term is NaN	Undefined	Undefined	Undefined
Left term = right term = +∞	Undefined	Undefined	Undefined
Left term = right term = -∞	Undefined	Undefined	Undefined
Left term > right term	0	0	0
Left term = right term	1	0	0
Left term < right term	0	1	1

___fcmp.s

Comparison of float type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of comparison
r6	Right term of comparison

[Return value]

r6	Value of int type which shows a result of comparison

[Description]

Comparison of single-precision floating-point.
A result of the following combination is returned.

	Return Value
Left term or right term is NaN	1
Left term > right term	1
Left term = right term	0
Left term < right term	-1

___negf.s

Negate of float type

[Classification]

Runtime library

[Argument(s)]

r6	Value whose sign is to be reversed

[Return value]

r6	Value whose sign has been reversed

[Description]

Negate of single-precision floating-point.

___notf.s

Logical negation of float type

[Classification]

Runtime library

[Argument(s)]

r6	Value which dose logical negation

[Return value]

r6	Integer which did logical negation

[Description]

Logical negation of single-precision floating-point.
A return value is int type.

___addf.d

Addition of double type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of addition
r7:r6	Right term of addition

[Return value]

r7:r6	Result of addition

[Description]

Addition of double-precision floating-point.

___subf.d

Subtraction of double type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of subtraction
r7:r6	Right term of subtraction

[Return value]

r7:r6	Result of subtraction

[Description]

Subtraction of double-precision floating-point.

___mulf.d

Multiplication of double type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of multiplication
r7:r6	Right term of multiplication

[Return value]

r7:r6	Result of multiplication

[Description]

Multiplication of double-precision floating-point.

___divf.d

Division of double type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of division
r7:r6	Right term of division

[Return value]

r7:r6	Result of division

[Description]

Division of double-precision floating-point.

___fcmp.d

Comparison of double type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of comparison
r7:r6	Right term of comparison

[Return value]

r6	Value of int type which shows a result of comparison

[Description]

Comparison of double-precision floating-point.
A result of the following combination is returned.

	Return Value
Left term or right term is NaN	1
Left term > right term	1
Left term = right term	0
Left term < right term	-1

___negf.d

Negate of double type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value whose sign is to be reversed

[Return value]

r7:r6	Value whose sign has been reversed

[Description]

Negate of double-precision floating-point.

___notf.d

Logical negation of double type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value which does logical negation

[Return value]

r6	Integer which did logical negation

[Description]

Logical negation of double-precision floating-point.
A return value is int type.

___add.l

Addition of long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of addition
r7:r6	Right term of addition

[Return value]

r7:r6	Result of addition

[Description]

Addition of 64-bit integer.

___sub.l

Subtraction of long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of subtraction
r7:r6	Right term of subtraction

[Return value]

r7:r6	Result of subtraction

[Description]

Subtraction of 64-bit integer.

___mul.l

Multiplication of long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of multiplication
r7:r6	Right term of multiplication

[Return value]

r7:r6	Result of multiplication

[Description]

Multiplication of 64-bit integer.

___div.l

Division of signed long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of division
r7:r6	Right term of division

[Return value]

r7:r6	Result of division

[Description]

Division of signed 64-bit integer.
If the divisor of the expression is 0, a result of 0 will be returned.

___div.ul

Division of unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of division
r7:r6	Right term of division

[Return value]

r7:r6	Result of division

[Description]

Division of unsigned 64-bit integer.
If the divisor of the expression is 0, a result of 0 will be returned.

___mod.l

Remainder of long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of remainder
r7:r6	Right term of remainder

[Return value]

r7:r6	Result of remainder

[Description]

Remainder of signed 64-bit integer.
If the divisor of the expression is 0, a result of 0 will be returned.

___mod.ul

Remainder of unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of remainder
r7:r6	Right term of remainder

[Return value]

r7:r6	Result of remainder

[Description]

Remainder of unsigned 64-bit integer.
If the divisor of the expression is 0, a result of 0 will be returned.

___shl.l

Logical left shift of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Left term of logical left shift
r8	Right term of logical left shift

[Return value]

r7:r6	Result of logical left shift

[Description]

Logical left shift of 64-bit integer.
The operation is performed after masking the right term by 0x3F, regardless of its sign.

___shr.l

Logical right shift of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Left term of logical right shift
r8	Right term of logical right shift

[Return value]

r7:r6	Result of logical right shift

[Description]

Logical right shift of 64-bit integer.
The operation is performed after masking the right term by 0x3F, regardless of its sign.

___sar.l

Arithmetic right shift long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Left term of arithmetic right shift
r8	Right term of arithmetic right shift

[Return value]

r7:r6	Result of arithmetic right shift

[Description]

Arithmetic right shift 64-bit integer.
The operation is performed after masking the right term by 0x3F, regardless of its sign.

___inc.l

Increment of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value to increment

[Return value]

r7:r6	Result of increment

[Description]

Increment of 64-bit integer.

___dec.l

Decrement of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value to decrement

[Return value]

r7:r6	Result of decrement

[Description]

Decrement of 64-bit integer.

___not.l

Logical negation of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value which does logical negation

[Return value]

r7:r6	Result of logical negation

[Description]

Logical negation of 64-bit integer.

___neg.l

Negate of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value whose sign is to be reversed

[Return value]

r7:r6	Result of the sign reversed

[Description]

Negate of 64-bit integer.

___cmp.l

Comparison of long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of comparison
r7:r6	Right term of comparison

[Return value]

r6	Value of int type which shows a result of comparison

[Description]

Comparison of signed 64-bit integer.
A result of the following combination is returned.

	Return Value
Left term > right term	1
Left term = right term	0
Left term < right term	-1

___cmp.ul

Comparison of unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r9:r8	Left term of comparison
r7:r6	Right term of comparison

[Return value]

r6	Value of int type which shows a result of comparison

[Description]

Comparison of unsigned 64-bit integer.
A result of the following combination is returned.

	Return Value
Left term > right term	1
Left term = right term	0
Left term < right term	-1

___bext.l

Bit field extraction of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value to extract the bit field from
r8	Upper 16 bits: The width of the bit field to extract Lower 16 bits: The location of the bit field to extract

[Return value]

r7:r6	The extracted bit field value

[Description]

Bit extraction of signed 64-bit integer.

The value of the lower 16 bits of r8 masked by 0x3F is the position of the bottom bit to extract.

The value of the upper 16 bits of r8 masked by 0xFFFF is the bit width to extract. Note, however, that if this width combined with the extraction bit location would exceed 64 bits, then the extraction bit width is shrunk so that it will fit within 64 bits.

The extracted bit field value is returned as type long long with sign extension. The top bit of the bit field acts as the sign bit.

If the extraction bit width is 0, then 0 is returned.

___bext.ul

Bit field extraction of unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value to extract the bit field from
r8	Upper 16 bits: The width of the bit field to extract Lower 16 bits: The location of the bit field to extract

[Return value]

r7:r6	The extracted bit field value

[Description]

Bit field extraction of unsigned 64-bit integer.

The value of the lower 16 bits of r8 masked by 0x3F is the position of the bottom bit to extract.

The value of the upper 16 bits of r8 masked by 0xFFFF is the bit width to extract. Note, however, that if this width combined with the extraction bit location would exceed 64 bits, then the extraction bit width is shrunk so that it will fit within 64 bits.

The extracted bit field value is zero-extended and returned.

If the extraction bit width is 0, then 0 is returned.

___bins.l

Bit field insertion of long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Bit field insertion destination
r9:r8	Bit field insertion data
0[sp]	Upper 16 bits: Bit field insertion width Lower 16 bits: Bit field insertion location

[Return value]

r7:r6	long long value after insertion

[Description]

Bit field insertion of 64-bit integer.

The lower 16 bits of the value stored in 0[sp] is masked by 0x3F, and this value is used as the bit field insertion location.

The upper 16 bits of the value stored in 0[sp] is masked by 0xFFFF, and this value is used as the bit field insertion width. Note, however, that if this width combined with the insertion location would exceed 64 bits, then the insertion width is shrunk so that it will fit within 64 bits.

If the insertion bit width is 0, then the value of r7:r6 is returned as-is.

___cvt.ws

Conversion from long type to float type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from 32-bit integer to single-precision floating-point number.

___cvt.wd

Conversion from long type to double type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from 32-bit integer to double-precision floating-point number.

___cvt.uws

Conversion from unsigned long type to float type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from unsigned 32-bit integer to single-precision floating-point number.

___cvt.uwd

Conversion from unsigned long type to double type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from unsigned 32-bit integer to double-precision floating-point number.

___cvt.ls

Conversion from 64-bit integer to float type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from 64-bit integer to single-precision floating-point number.

___cvt.ld

Conversion from long long type to double type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from 64-bit integer to double-precision floating-point number.

___cvt.uls

Conversion from unsigned long long type to float type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from unsigned 64-bit integer to single-precision floating-point number.

___cvt.uld

Conversion from unsigned long long type to double type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from unsigned 64-bit integer to double-precision floating-point number.

___trnc.sw

Conversion from float type to long type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from single-precision floating-point number to 32-bit integer.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x80000000	0
Bigger than +0xFFFFFFFF	0
Others	Integer after conversion

___trnc.dw

Conversion from double type to long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from double-precision floating-point number to 32-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x80000000	0
Bigger than +0xFFFFFFFF	0
Others	Integer after conversion

___trnc.suw

Conversion from float type to unsigned long type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from single-precision floating-point number to unsigned 32-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x80000000	0
Bigger than +0xFFFFFFFF	0
Others	Integer after conversion

___trnc.duw

Conversion from double type to unsigned long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from double-precision floating-point number to unsigned 32-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x80000000	0
Bigger than +0xFFFFFFFF	0
Others	Integer after conversion

___trnc.sl

Conversion from float type to long long type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from single-precision floating-point number to 64-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x8000000000000000	0
Bigger than +0xFFFFFFFFFFFFFFFF	0
Others	Integer after conversion

___trnc.dl

Conversion from double type to long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from double-precision floating-point number to 64-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x8000000000000000	0
Bigger than +0xFFFFFFFFFFFFFFFF	0
Others	Integer after conversion

___trnc.sul

Conversion from float type to unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from single-precision floating-point number to unsigned 64-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x8000000000000000	0
Bigger than +0xFFFFFFFFFFFFFFFF	0
Others	Integer after conversion

___trnc.dul

Conversion from double type to unsigned long long type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from double-precision floating-point number to unsigned 64-bit integer.
Decimals are rounded toward 0.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN or ±∞	0
Smaller than -0x8000000000000000	0
Bigger than +0xFFFFFFFFFFFFFFFF	0
Others	Integer after conversion

___cvt.sd

Conversion from float type to double type

[Classification]

Runtime library

[Argument(s)]

r6	Value before conversion

[Return value]

r7:r6	Value after conversion

[Description]

Conversion from single-precision floating-point number to double-precision floating-point number.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN	NaN
±∞	±∞
Others	Value after conversion

___cvt.ds

Conversion from double type to float type

[Classification]

Runtime library

[Argument(s)]

r7:r6	Value before conversion

[Return value]

r6	Value after conversion

[Description]

Conversion from double-precision floating-point number to single-precision floating-point number.
A result of the following combination is returned.

Value before Conversion	Return Value
NaN	NaN
±∞	±∞
Others	Value after conversion

__mul

Multiplication of int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of multiplication
r6	Right term of multiplication

[Return value]

r6	Result of multiplication

[Description]

Multiplication of signed integer.

__mulu

Multiplication of unsigned int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of multiplication
r6	Right term of multiplication

[Return value]

r6	Result of multiplication

[Description]

Multiplication of unsigned integer.

__div

Division of int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of division
r6	Right term of division

[Return value]

r6	Result of division

[Description]

Division of signed integer.
If there is a division by zero, then 0 is returned.

__divu

Division of unsigned int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of division
r6	Right term of division

[Return value]

r6	Result of division

[Description]

Division of unsigned integer.
If there is a division by zero, then 0 is returned.

__mod

Remainder of int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of remainder
r6	Right term of remainder

[Return value]

r6	Result of remainder

[Description]

Remainder of signed integer.
If there is a division by zero, then 0 is returned.

__modu

Remainder of unsigned int type

[Classification]

Runtime library

[Argument(s)]

r7	Left term of remainder
r6	Right term of remainder

[Return value]

r6	Result of remainder

[Description]

Remainder of unsigned integer.
If there is a division by zero, then 0 is returned.

6.4.14 Function pre/post processing runtime functions

Function pre/post processing runtime function is a routine that is automatically called by the process of the CX prologue/epilogue functions. Similar to "operation runtime function", function pre/post processing runtime function is not described in the C source or assembler source.

The V850Ex core uses the CALLT instruction to call the function pre/post processing runtime function. The code efficiency can be enhanced by calling these functions from the table of the CALLT instruction.

Calling the function pre/post processing runtime function is valid when:

- An optimization option other than "-Ospeed" (execution speed priority optimization) is specified.
- The compiler option "-Xpro_epi_runtime=on" is specified.

Table 6-31. Function pre/post Processing Runtime Functions

Function/Macro Name	Outline
___Epush250, ___Epush251, ___Epush252, ___Epush253, ___Epush254, ___Epush260, ___Epush261, ___Epush262, ___Epush263, ___Epush264, ___Epush270, ___Epush271, ___Epush272, ___Epush273, ___Epush274, ___Epush280, ___Epush281, ___Epush282, ___Epush283, ___Epush284, ___Epush290, ___Epush291, ___Epush292, ___Epush293, ___Epush294, ___Epushlp0, ___Epushlp1, ___Epushlp2, ___Epushlp3, ___Epushlp4	Prologue processing of functions
___push2000, ___push2001, ___push2002, ___push2003, ___push2004, ___push2040, ___push2100, ___push2101, ___push2102, ___push2103, ___push2104, ___push2140, ___push2200, ___push2201, ___push2202, ___push2203, ___push2204, ___push2240, ___push2300, ___push2301, ___push2302, ___push2303, ___push2304, ___push2340, ___push2400, ___push2401, ___push2402, ___push2403, ___push2404, ___push2440, ___push2500, ___push2501, ___push2502, ___push2503, ___push2504, ___push2540, ___push2600, ___push2601, ___push2602, ___push2603, ___push2604, ___push2640, ___push2700, ___push2701, ___push2702, ___push2703, ___push2704, ___push2740, ___push2800, ___push2801, ___push2802, ___push2803, ___push2804, ___push2840, ___push2900, ___push2901, ___push2902, ___push2903, ___push2904, ___push2940, ___pushlp00, ___pushlp01, ___pushlp02, ___pushlp03, ___pushlp04, ___pushlp40	Prologue processing of functions
___Epop250, ___Epop251, ___Epop252, ___Epop253, ___Epop254, ___Epop260, ___Epop261, ___Epop262, ___Epop263, ___Epop264, ___Epop270, ___Epop271, ___Epop272, ___Epop273, ___Epop274, ___Epop280, ___Epop281, ___Epop282, ___Epop283, ___Epop284, ___Epop290, ___Epop291, ___Epop292, ___Epop293, ___Epop294, ___Epoplp0, ___Epoplp1, ___Epoplp2, ___Epoplp3, ___Epoplp4	Epilogue processing of function
___pop2000, ___pop2001, ___pop2002, ___pop2003, ___pop2004, ___pop2040, ___pop2100, ___pop2101, ___pop2102, ___pop2103, ___pop2104, ___pop2140, ___pop2200, ___pop2201, ___pop2202, ___pop2203, ___pop2204, ___pop2240, ___pop2300, ___pop2301, ___pop2302, ___pop2303, ___pop2304, ___pop2340, ___pop2400, ___pop2401, ___pop2402, ___pop2403, ___pop2404, ___pop2440, ___pop2500, ___pop2501, ___pop2502, ___pop2503, ___pop2504, ___pop2540, ___pop2600, ___pop2601, ___pop2602, ___pop2603, ___pop2604, ___pop2640, ___pop2700, ___pop2701, ___pop2702, ___pop2703, ___pop2704, ___pop2740, ___pop2800, ___pop2801, ___pop2802, ___pop2803, ___pop2804, ___pop2840, ___pop2900, ___pop2901, ___pop2902, ___pop2903, ___pop2904, ___pop2940, ___poplp00, ___poplp01, ___poplp02, ___poplp03, ___poplp04, ___poplp40	Epilogue processing of function

6.5 Library Consumption Stack List

This section explains stack consumption amount of all function included in library.

6.5.1 Standard library

Stack consumption amount (Unit: Byte) of all function included in standard library are shown below.

(1) Functions with variable arguments

Table 6-32. Functions with Variable Arguments

Function/Macro Name	Stack Consumption Amount
va_start	0
va_end	0
va_arg	0

(2) Character string functions

Table 6-33. Character String Functions

Function/Macro Name	Stack Consumption Amount
memchr	0
memcmp	0
bcmp	0
memcpy	0
bcopy	0
memmove	0
memset	0

(3) Memory management functions

Table 6-34. Memory Management Functions

Function/Macro Name	Stack Consumption Amount
index	0
strpbrk	0
rindex	0
strrchr	0
strchr	0
strstr	0
strspn	0
strcspn	0
strcmp	0
strncmp	0
strcpy	0

Function/Macro Name	Stack Consumption Amount
strncpy	0
strcat	0
strncat	0
strtok	0
strlen	0
strerror	0

(4) Character conversion functions

Table 6-35. Character Conversion Functions

Function/Macro Name	Stack Consumption Amount
toupper	0
_toupper	0
tolower	0
_tolower	0
toascii	0

(5) Character classification functions

Table 6-36. Character Classification Functions

Function/Macro Name	Stack Consumption Amount
isalnum	0
isalpha	0
isascii	0
isupper	0
islower	0
isdigit	0
isxdigit	0
iscntrl	0
ispunct	0
isspace	0
isprint	0
isgraph	0

(6) Standard I/O functions

Table 6-37. Standard I/O Functions

Function/Macro Name	Stack Consumption Amount
fread	40
getc	0
fgetc	0
fgets	0
fwrite	28
putc	0
fputc	0
fputs	0
getchar	0
gets	0
putchar	0
puts	0
sprintf	284
fprintf	276
vsprintf	268
printf	276
vfprintf	256
vprintf	264
sscanf	244
fscanf	236
scanf	236
ungetc	0
rewind	0
perror	288

(7) Standard utility functions

Table 6-38. Standard Utility Functions

Function/Macro Name	Stack Consumption Amount
abs	0
labs	0
llabs	0
bsearch	40
qsort	76
div	0

Function/Macro Name	Stack Consumption Amount
ldiv	0
lldiv	36
itoa	36
ltoa	36
ultoa	44
lltoa	88
ulltoa	76
ecvt	168
ecvtf	124
fcvt	168
fcvtf	124
gcvt	232
gcvtf	232
atoi	64
atol	64
atoll	72
strtol	72
strtoul	64
strtoll	72
strtoull	72
atoff	140
atof	140
strtodf	140
strtod	140
calloc	20
malloc	12
realloc	24
free	16
rand	0
srand	0

(8) Non-local jump functions

Table 6-39. Non-Local Jump Functions

Function/Macro Name	Stack Consumption Amount
longjmp	0
setjmp	0

6.5.2 Mathematical library

Stack consumption amount (Unit: Byte) of all function included in mathematical library are shown below.

(1) Mathematical functions

Table 6-40. Mathematical Functions

Function/Macro Name	Stack Consumption Amount
j0f	68
j1f	68
jnf	88
y0f	80
y1f	80
ynf	100
erff	64
erfcf	64
expf	40
exp	56
logf	44
log	68
log2f	44
log10f	44
log10	68
powf	52
pow	88
sqrtf	72
sqrt	52
cbrtf	44
cbrt	64
ceilf	0
ceil	0
fabsf	0
fabs	0
floorf	0
floor	4
fmodf	32
fmod	56
frexpf	32
frexp	44
ldexpf	32

Function/Macro Name	Stack Consumption Amount
ldexp	44
modff	0
modf	0
gammaf	52
hypotf	44
matherrf (matherr)	0
matherrd	0
cosf	40
cos	64
sinf	40
sin	68
tanf	52
tan	80
acosf	52
acos	68
asinf	52
asin	68
atanf	48
atan	68
atan2f	52
atan2	80
coshf	40
cosh	60
sinhf	40
sinh	60
tanhf	48
tanh	60
acoshf	44
asinhf	44
atanhf	44

6.5.3 Initialization library

Stack consumption amount (Unit: Byte) of all function included in initialization library are shown below.

(1) Initialization peripheral devices function

Table 6-41. Initialization Library

Function/Macro Name	Stack Consumption Amount
hdwinit	0

6.5.4 ROMization library

Stack consumption amount (Unit: Byte) of all function included in ROMization library are shown below.

(1) Copy functions

Table 6-42. Copy Functions

Function/Macro Name	Stack Consumption Amount
_rcopy	24
_rcopy1	24
_rcopy2	20
_rcopy4	20

6.5.5 Multi-core library

Stack consumption amount (Unit: Byte) of all function included in multi-core library are shown below.

(1) Pseudo "main" functions for multi-core

Table 6-43. Pseudo "main" Functions for Multi-core

Function/Macro Name	Stack Consumption Amount
main_pe2	0
main_pe3	0
main_pe4	0
main_pe5	0
main_pe6	0
main_pe7	0
main_pe8	0
main_pe9	0
main_pe10	0
main_pe11	0
main_pe12	0
main_pe13	0
main_pe14	0

Function/Macro Name	Stack Consumption Amount
main_pe15	0
main_pe16	0
main_pe17	0
main_pe18	0
main_pe19	0
main_pe20	0
main_pe21	0
main_pe22	0
main_pe23	0
main_pe24	0
main_pe25	0
main_pe26	0
main_pe27	0
main_pe28	0
main_pe29	0
main_pe30	0
main_pe31	0

6.5.6 Runtime library

Stack consumption amount (Unit: Byte) of all function included in runtime library are shown below.

(1) Operation runtime functions

Table 6-44. Operation Runtime Functions

Function/Macro Name	Stack Consumption Amount
___addf.s	84
___subf.s	84
___mulf.s	84
___divf.s	84
___cmpf.s	84
___fcmp.s	84
___negf.s	92
___notf.s	92
___addf.d	96
___subf.d	96
___mulf.d	96
___divf.d	140
___fcmp.d	72
___negf.d	108

Function/Macro Name	Stack Consumption Amount
___notf.d	84
___add.l	0
___sub.l	0
___mul.l	0
___div.l	32
___div.ul	16
___mod.l	40
___mod.ul	20
___shl.l	4
___shr.l	4
___sar.l	4
___inc.l	0
___dec.l	0
___not.l	0
___neg.l	0
___cmp.l	4
___cmp.ul	0
___bext.l	8
___bext.ul	8
___bins.l	12
___cvt.ws	16
___cvt.wd	8
___cvt.uws	16
___cvt.uwd	8
___cvt.ls	20
___cvt.ld	24
___cvt.uls	8
___cvt.uld	12
___trnc.sw	4
___trnc.dw	4
___trnc.suw	4
___trnc.duw	4
___trnc.sl	12
___trnc.dl	12
___trnc.sul	12
___trnc.dul	12
___cvt.sd	4

Function/Macro Name	Stack Consumption Amount
___cvt.ds	12
__mul	12
__mulu	12
__div	20
__divu	16
__mod	20
__modu	16

(2) Function pre/post processing runtime functions

Table 6-45. Function Pre/Post Processing Runtime Functions

Function/Macro Name	Stack Consumption Amount
___Epush250	0
___Epush251	0
___Epush252	0
___Epush253	0
___Epush254	0
___Epush260	0
___Epush261	0
___Epush262	0
___Epush263	0
___Epush264	0
___Epush270	0
___Epush271	0
___Epush272	0
___Epush273	0
___Epush274	0
___Epush280	0
___Epush281	0
___Epush282	0
___Epush283	0
___Epush284	0
___Epush290	0
___Epush291	0
___Epush292	0
___Epush293	0
___Epush294	0

Function/Macro Name	Stack Consumption Amount
___Epushlp0	0
___Epushlp1	0
___Epushlp2	0
___Epushlp3	0
___Epushlp4	0
___Epop250	0
___Epop251	0
___Epop252	0
___Epop253	0
___Epop254	0
___Epop260	0
___Epop261	0
___Epop262	0
___Epop263	0
___Epop264	0
___Epop270	0
___Epop271	0
___Epop272	0
___Epop273	0
___Epop274	0
___Epop280	0
___Epop281	0
___Epop282	0
___Epop283	0
___Epop284	0
___Epop290	0
___Epop291	0
___Epop292	0
___Epop293	0
___Epop294	0
___Epoplp0	0
___Epoplp1	0
___Epoplp2	0
___Epoplp3	0
___Epoplp4	0
___push2000	0
___push2001	0

Function/Macro Name	Stack Consumption Amount
___push2002	0
___push2003	0
___push2004	0
___push2040	0
___push2100	0
___push2101	0
___push2102	0
___push2103	0
___push2104	0
___push2140	0
___push2200	0
___push2201	0
___push2202	0
___push2203	0
___push2204	0
___push2240	0
___push2300	0
___push2301	0
___push2302	0
___push2303	0
___push2304	0
___push2340	0
___push2400	0
___push2401	0
___push2402	0
___push2403	0
___push2404	0
___push2440	0
___push2500	0
___push2501	0
___push2502	0
___push2503	0
___push2504	0
___push2540	0
___push2600	0
___push2601	0
___push2602	0

Function/Macro Name	Stack Consumption Amount
___push2603	0
___push2604	0
___push2640	0
___push2700	0
___push2701	0
___push2702	0
___push2703	0
___push2704	0
___push2740	0
___push2800	0
___push2801	0
___push2802	0
___push2803	0
___push2804	0
___push2840	0
___push2900	0
___push2901	0
___push2902	0
___push2903	0
___push2904	0
___push2940	0
___pushlp00	0
___pushlp01	0
___pushlp02	0
___pushlp03	0
___pushlp04	0
___pushlp40	0
___pop2000	0
___pop2001	0
___pop2002	0
___pop2003	0
___pop2004	0
___pop2040	0
___pop2100	0
___pop2101	0
___pop2102	0
___pop2103	0

Function/Macro Name	Stack Consumption Amount
___pop2104	0
___pop2140	0
___pop2200	0
___pop2201	0
___pop2202	0
___pop2203	0
___pop2204	0
___pop2240	0
___pop2300	0
___pop2301	0
___pop2302	0
___pop2303	0
___pop2304	0
___pop2340	0
___pop2400	0
___pop2401	0
___pop2402	0
___pop2403	0
___pop2404	0
___pop2440	0
___pop2500	0
___pop2501	0
___pop2502	0
___pop2503	0
___pop2504	0
___pop2540	0
___pop2600	0
___pop2601	0
___pop2602	0
___pop2603	0
___pop2604	0
___pop2640	0
___pop2700	0
___pop2701	0
___pop2702	0
___pop2703	0
___pop2704	0

Function/Macro Name	Stack Consumption Amount
___pop2740	0
___pop2800	0
___pop2801	0
___pop2802	0
___pop2803	0
___pop2804	0
___pop2840	0
___pop2900	0
___pop2901	0
___pop2902	0
___pop2903	0
___pop2904	0
___pop2940	0
___poplp00	0
___poplp01	0
___poplp02	0
___poplp03	0
___poplp04	0
___poplp40	0

6.5.7 Libraries used in V850E2V3-FPU

Stack consumption amount (Unit: Byte) of all function included in libraries used in V850E2V3-FPU are shown below.

(1) Functions used in V850E2V3-FPU

Table 6-46. Functions Used in V850E2V3-FPU

Function/Macro Name	Stack Consumption Amount
expf	212
exp	272
logf	140
log	204
log10f	120
log10	204
powf	180
pow	356
sqrtf	24
sqrt	36
ceilf	20

Function/Macro Name	Stack Consumption Amount
ceil	36
floorf	20
floor	36
fmodf	100
fmod	176
frexpf	56
frexp	84
ldexpf	136
ldexp	136
modff	36
modf	68
cosf	220
cos	352
sinf	220
sin	352
tanf	84
tan	176
acosf	80
acos	472
asinf	72
asin	384
atanf	108
atan	288
atan2f	160
atan2	360
coshf	268
cosh	352
sinhf	268
sinh	352
tanhf	272
tanh	380

CHAPTER 7 STARTUP

This chapter explains the startup routine.

7.1 Outline

In order to execute the program by C language, ROMization process for embedding in system and the program that starts the user program (main function) is needed. This program is called as startup routine.

In order to excute the programm creatd by user, startup routine corresponding to that programm must be created. CubeSuite provides, object module file of startup routine that includes the necessary process which needs to be executed before execution of the program as well it provides startup routine which user can change as per his system requirements.

Remark Multi-core programming requires a startup routine for multi-core programs.

7.2 File Contents

Startup routine that CubeSuite supplies is as follows:

Table 7-1. Startup Routine Samples

Storage Location	File Name	Contents
Version Folder\lib\850e	cstart.asm	Startup routine sample for V850Ex core
	cstartN.asm	For not ROMization Startup routine sample for V850Ex core
	cstartM.asm	Startup routine sample for multi-core
	cstartMN.asm	For not ROMization Startup routine sample for multi-core

To create a new startup routine, copy the above sample and add it to the project. And then edit it.

If the startup routine is not added to the project, the CX automatically links a default startup routine (object). The files to be linked result from compiling (assembling) sample startup routines "cstart.asm" and "cstartN.asm".

These objects are assembled with the assembler options "-Xcommon=v850e" and can be used commonly in the V850 microcontrollers.

7.3 Startup Routine

Startup routine is the routine that is to be executed after V850 is reset and before the execution of main function. Basically, it carries out the initialization after system is reset. Specifically, it (startup routine) carries out following things:

- Setting RESET handler when reset is input
- Setting of register mode of startup routine
- Securing stack area and setting stack pointer
- Securing argument area for main function
- Setting text pointer (tp)
- Setting global pointer (gp)
- Setting element pointer (ep)
- Initializing peripheral I/O registers that must be initialized before execution of main function

- Initializing user target that must be initialized before execution of main function
- Clearing sbss area to 0
- Clearing bss area to 0
- Clearing sebss area to 0
- Clearing tibss.byte area to 0
- Clearing tibss.word area to 0
- Clearing sibss area to 0
- Setting of CTBP value for function pre/post processing runtime function
- Setting of programmable peripheral I/O register value
- Setting r6 and r7 as argument of main function
- Branching to main function (when not using real-time OS)
- Branching to initialization routine of real-time OS (when using real-time OS)
- V850E2V3 multi-core startup routine

Of course, there are processes which are not required by system, those can be omitted.
Also, except these processes if there are some more process that user may want to execute, these can be described.
The description example indicated on after 7.3.1 assumes and is explaining various cases.
Therefore there is a possibility different from a CubeSuite offers startup routine description.
These processes, basically are needed to be described by assembler instructions.

7.3.1 Setting RESET handler when reset is input

Describing the process to be performed when a reset (reset interrupt) is input. Execution branches to the handler address 0x0 when a reset is input in the V850. Therefore, allocate an instruction that branches to the beginning of the startup routine to address 0x0. Resetinterrupt cannot be described by # pragma interrupt specification on C language, therefore it describes by the assembler instruction. Description is as follows.

```
RESET   .cseg   TEXT
        jr      __start
__start:
```

Use the .cseg directive to allocate an instruction to the handler address. If the above description is made, the "jr __ start" instruction is allocated to the handler address of RESET.

If the jr instruction cannot reach the destination, i.e., if "__ start" is not within ± 2 Mbytes from address 0x0, use the jmp instruction as follows.

```
RESET   .cseg   TEXT
        mov     #__start, lp
        jmp     [lp]
__start:
```

In this case, one register is used. The lp (r31) register is used in the above example. Any general-purpose register whose contents can be lost at this point can be used. The lp (r31) register in which the return address from a function is stored is not used when a reset is input. Therefore, it is safe to use the lp (r31) register.

The description of the .cseg directive does not always have to be in the startup routine.

In the example symbol for startup routine is "__start", however, it can be any other name.

7.3.2 Setting of register mode of startup routine

Describe the setting of the register mode in the startup routine described with assembler instructions.

However, this setting is necessary only when the 22-register mode or 26-register mode is used for the overall system. It is not necessary to describe this setting when the 32-register mode is specified.

[At 22-register mode]

```
$ REG_MODE 22
```

[At 26-register mode]

```
$ REG_MODE 26
```

[At universal register mode]

```
$ REG_MODE common
```

If this setting is not described, the linker outputs the following warning message.

```
W0565308: input files have different register modes.
        use "-Xregmode_info" option for more information.
```

7.3.3 Securing stack area and setting stack pointer

Secure the stack area used by the system and set the stack pointer (SP = r3) at the end of this area. When a real-time OS is used, however, the stack specified here is used until execution branches to the initialization routine of the real-time OS.

In other words, it is hardly used or not used at all. If a large stack area is secured, therefore, the RAM area is wasted. Check if the stack is used before execution branches to the initialization routine of the real-time OS. Interrupts must be especially noted. It seems, however, that the startup routine is mostly executed with interrupts disabled.

The stack area is secured as follows.

```
STACKSIZE       .set    0x200
                .dseg   BSS
mov     #__stack + STACKSIZE, sp
```

This is an example of securing a 0x200-byte stack in the .bss area. The contents of the stack are allocated to a bss attribute area because they do not have an initial value. Of course, they can be allocated to the sbss area, but the size of the stack that can be allocated to the sbss area is limited because the sbss area is accessed with a single gp-relative instruction. It is recommended to allocate the stack contents to the bss area if the stack size is great, as it may be better to allocate other variables to the sbss area.

Change the value written to the .set instruction to change the stack size to be secured. The CX generates codes on the assumption that the sp is at a 4-byte boundary when it references the memory relatively with the stack pointer (sp). Therefore, be sure to allocate the stack pointer at a 4-byte boundary. If necessary, use the directive ".align 4", and Make the number specified by the ".set" instruction a multiple of 4.

The stack has a serious effect on the operation of the system. If the stack area runs short, the stack size exceeds the secured area and the stack contents are lost, which may cause a system hang-up. Estimate the stack size to be used by functions using stack usage tracer included with the CX, and secure a sufficient stack size.

7.3.4 Securing argument area for main function

In ANSI C specifications, main function format is defined as "int main(void) { ... }" having no parameters or, as the main function with two parameters "int main(int argc, char *argv[]) { ... }".

argc of the function having two parameters is a value that is not negative and indicates the total number of parameters. argv indicates an array of pointers to argument character strings. argv[argc] is NULL (vacant pointer). If argc is 1 or more, argv[0] to argv[argc - 1] are pointers to character strings.

Secure the areas for argc and argv in the startup routine. Securing method is as shown below.

```
        .dseg   DATA
        .align  4
__argc:
        .db4    0
__argv:
        .db4    #.L16
.L16:
        .db     0
        .db     0
        .db     0
        .db     0
```

This area has initialization definition, therefore it is allocated to "data attribute area".

The above area is not necessary if the main function is defined in the format: int main(void) { ... }.

The used RAM area can be reduced by deleting the above area.

Actually, processing that sets arguments (r6 and r7) of the main function is performed immediately before the main function. If r6 and r7 are not used in the startup routine, the processing can be executed immediately after the above program. See "7.3.18 Setting r6 and r7 as argument of main function" for the processing to be set.

7.3.5 Setting text pointer (tp)

The text pointer (tp) is a pointer prepared to implement referencing (PIC: Position Independent Code) independent of the position at which the text area of an application, i.e., program code is allocated when the program code is referenced. For example, if it is necessary to reference a specific location in the code during program execution, the CX outputs the code to be accessed in tp-relative mode.

Since the code is output on the assumption that tp is correctly set, tp must be correctly set in the startup routine.

The text pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the text pointer is described as follows.

```
__tp_TEXT@%TP_SYMBOL {TEXT};
```

The text pointer value is the beginning of the TEXT segment, and is in "__ tp_TEXT".

Describe as follows to set tp in the startup routine.

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

7.3.6 Setting global pointer (gp)

External variables or data defined in an application are allocated to the memory. The global pointer (gp) is a pointer prepared to implement referencing independent of location position (PID: Position Independent Data) when the variables or data allocated to the memory are referenced. The CX outputs a code for the section that is to be accessed in gp-relative mode.

Since the code is output on the assumption that gp is correctly set, gp must be correctly set in the startup routine.

The global pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the global pointer is described as follows.

```
__gp_DATA@%GP_SYMBOL {DATA};
```

The gp symbol value can be defined at the beginning of "data segment" of the DATA segment as shown above, or offset from a text symbol.

Using the second method, the gp symbol value is determined by adding value of tp and offset value from tp. In other words, a code that is independent of location can be generated. To copy a program code and data used by that code to the RAM area simultaneously and execute them, the value of gp can be acquired immediately if the start address of the copy destination is known. In this case, the symbol directive is described as follows.

```
__tp_TEXT@%TP_SYMBOL {TEXT};
__gp_DATA@%GP_SYMBOL &__tp_TEXT {DATA};
```

The global pointer value is "__ tp_TEXT to which the value of __ gp_DATA is added", and the value to be added, i.e., offset value, is stored in "__ gp_DATA". Therefore, describe as follows to set gp in the startup routine.

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

This sets the correct value of the global pointer to gp.

7.3.7 Setting element pointer (ep)

Of the external variables or data defined in an application, those that are allocated to the following sections are accessed from the element pointer (ep) in relative mode.

- sedata/sebss section
- sidata/sibss section
- tidata.byte/tibss.byte section
- tidata.word/tibss.word section

If these sections exist, the CX outputs a code to access these areas in ep-relative mode.

Since the code is output on the assumption that ep is correctly set, ep must be correctly set in the startup routine.

The element pointer value is determined during linking, and is in a symbol defined by a symbol directive that is described in the link directive file. For example, suppose that the symbol directive of the element pointer is described as follows.

```
__ep_DATA@%EP_SYMBOL;
```

The element pointer value is the beginning of the SIDATA segment by default, and its value is in "__ ep_DATA". Therefore, describe as follows to set ep in the startup routine.

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

Reference the absolute address of __ ep_DATA and set that value to ep.

7.3.8 Initializing peripheral I/O registers that must be initialized before execution of main function

When the external RAM is initialized by the startup routine, the external memory must first be set to the peripheral I/O; otherwise the memory area cannot be accessed and initialized. In addition, initialize the peripheral I/O registers that must be set for executing the startup routine.

Register setting can be described with assembler instructions, or execution may once branch from the startup routine to a C function and register setting can be described in this function. If it is described in C, reading and substitution in the peripheral I/O can be described in a visually simple way. For example, when creating the C function "void reset(void)" and calling it from the startup routine, describe the following instruction in the startup routine.

```
jarl    _reset, lp
```

Differences between assembler instruction description and C description are shown below using the following examples. An instruction that substitutes "1" in P0 (port 0) is described in an assembler source (use r 10) and as a C source is as follows.

[Assembler source]

```
mov     1, r10
st.b    r10, P0
```

[C source]

```
#pragma ioreg
P0 = 1;
```

The external memory setting differs depending on the device. See the Relevant Device's Hardware User's Manual of each device.

With a clock generation function, the "internal system clock" that is supplied to each unit built in the V850 needs to be generated. In this case, the clock needs to be multiplied by a PLL (Phase locked loop) synthesizer before use. In other words, the clock must be correctly set to the frequency used; otherwise the clock operates slower or faster than the assumed operation speed.

Regarding the default value of the PLL, usually, the multiplication value is small and the operation frequency is low. These also apply to the startup routine. If the clearing of the memory area that is explained in "7.3.10 Clearing sbss area to 0" and later sections is executed while the operating frequency is low, it takes a lot of time to complete the execution. Therefore, it is recommended that the PLL be set during the early stages of the startup routine.

Aside from the above settings, set the following settings: the "system wait control register (VSWC)", the" command register (PRCMD)", and, if necessary, the "watch dog timer (WDT)". For the correct settings, see the Relevant Device's Hardware User's Manual.

7.3.9 Initializing user target that must be initialized before execution of main function

Describe the necessary initialization processing for the user target, if any, in the startup routine.

The processing can be described with assembler language source or execution may once branch from the startup routine to a C function and the processing can be described in this function.

7.3.10 Clearing sbss area to 0

Initialize the sbss area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sebss area to zero.

This processing is not necessary if the sbss section has not been created or if it is not necessary to clear the sbss area to zero.

Use symbols "__ssbss" and "__esbss" reserved for the CX to clear the sbss area. The meaning of each symbol is as follows.

Table 7-2. Symbols of sbss Area

Symbol Name	Meaning
__ssbss	Symbol indicating start of sbss area
__esbss	Symbol indicating end of sbss area

The values (addresses) of these symbols are determined during linking. The program that clears the sbss area using these symbols is as follows.

```
        .extern __ssbss, 4
        .extern __esbss, 4
        mov     #__ssbss, r13
        mov     #__esbss, r12
        cmp     r12, r13
        jnl     .L11
.L12:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L12
.L11:
```

This program clears the sbss area to zero in 4-byte units.

7.3.11 Clearing bss area to 0

Initialize the bss area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the bss area to zero.

This processing is not necessary if the bss section has not been created or if it is not necessary to clear the bss area to zero.

Use symbols "__sbss" and "__ebss" reserved for the CX to clear the bss area. The meaning of each symbol is as follows.

Table 7-3. Symbols of bss Area

Symbol Name	Meaning
__sbss	Symbol indicating start of bss area
__ebss	Symbol indicating end of bss area

The values (addresses) of these symbols are determined during linking. The program that clears the bss area using these symbols is as follows.(This program clears the bss area to zero in 4-byte units.)

```
        .extern __sbss, 4
        .extern __ebss, 4
        mov     #__sbss, r13
        mov     #__ebss, r12
        cmp     r12, r13
        jnl     .L14
.L15:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L15
.L14:
```

7.3.12 Clearing sebss area to 0

Initialize the sebss area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sebss area to zero.

This processing is not necessary if the sebss section has not been created or if it is not necessary to clear the sebss area to zero.

Use symbols "__ssebss" and "__esebss" reserved for the CX to clear the sebss area. The meaning of each symbol is as follows

Table 7-4. Symbols of sebss Area

Symbol Name	Meaning
__ssebss	Symbol indicating start of sebss area
__esebss	Symbol indicating end of sebss area

The values (addresses) of these symbols are determined during linking. The program that clears the sebss area using these symbols is as follows.(This program clears the sebss area to zero in 4-byte units.)

```
        .extern __ssebss, 4
        .extern __esebss, 4
        mov     #__ssebss, r13
        mov     #__esebss, r12
        cmp     r12, r13
        jnl     .L17
.L18:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L18
.L17:
```

7.3.13 Clearing tibss.byte area to 0

Initialize the tibss.byte area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the tibss.byte area to zero.

This processing is not necessary if the tibss.byte section has not been created or if it is not necessary to clear the tibss.byte area to zero.

Use symbols "__stibss.byte" and "__etibss.byte" reserved for the CX to clear the tibss.byte area. The meaning of each symbol is as follows.

Table 7-5. Symbols of tibss.byte Area

Symbol Name	Meaning
__stibss.byte	Symbol indicating start of tibss.byte area
__etibss.byte	Symbol indicating end of tibss.byte area

The values (addresses) of these symbols are determined during linking. The program that clears the tibss.byte area using these symbols is as follows.(This program clears the tibss.byte area to zero in 1-byte units.)

```
        .extern __stibss.byte, 4
        .extern __etibss.byte, 4
        mov     #__stibss.byte, r13
        mov     #__etibss.byte, r12
        cmp     r12, r13
        jnl     .L20
.L21:
        st.b    r0, [r13]
        add     1, r13
        cmp     r12, r13
        jl      .L21
.L20:
```

7.3.14 Clearing tibss.word area to 0

Initialize the tibss.word area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the tibss.word area to zero.

This processing is not necessary if the tibss.word section has not been created or if it is not necessary to clear the tibss.word area to zero.

Use symbols "__stibss.word" and "__etibss.word" reserved for the CX to clear the tibss.word area. The meaning of each symbol is as follows

Table 7-6. Symbols of tibss.word Area

Symbol Name	Meaning
__stibss.word	Symbol indicating start of tibss.word area
__etibss.word	Symbol indicating end of tibss.word area

The values (addresses) of these symbols are determined during linking. The program that clears the tibss.word area using these symbols is as follows.

```
        .extern __stibss.word, 4
        .extern __etibss.word, 4
        mov     #__stibss.word, r13
        mov     #__etibss.word, r12
        cmp     r12, r13
        jnl     .L23
.L24:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L24
.L23:
```

7.3.15 Clearing sibss area to 0

Initialize the sibss area, one of the bss attribute areas that do not have an initial value.

Since the memory contents are undefined after the V850 is reset, it is recommended to clear the sibss area to zero.

This processing is not necessary if the sibss section has not been created or if it is not necessary to clear the sibss area to zero.

Use symbols "__ssibss" and "__esibss" reserved for the CX to clear the sibss area. The meaning of each symbol is as follows.

Table 7-7. Symbols of sibss Area

Symbol Name	Meaning
__ssibss	Symbol indicating start of sibss area
__esibss	Symbol indicating end of sibss area

The values (addresses) of these symbols are determined during linking. The program that clears the sibss area using these symbols is as follows.(This program clears the sibss area to zero in 4-byte units.)

```
        .extern __ssibss, 4
        .extern __esibss, 4
        mov     #__ssibss, r13
        mov     #__esibss, r12
        cmp     r12, r13
        jnl     .L26
.L25:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L25
.L26:
```

7.3.16 Setting of CTBP value for function pre/post processing runtime function

This setting is necessary when the function pre/post processing runtime function is used.

Since the CALLT instruction is used when the function pre/post processing runtime function is called, the value of CTBP necessary for the CALLT instruction must be set at the beginning of the function table of the function pre/post processing runtime function.

The function pre/post processing runtime function is used in the following case.

- If compiler option "-Xpro_epi_runtime=on" and "-Ospeed" is set.

If a compiler option other than "-Ospeed" is specified for optimization, "-Xpro_epi_runtime=on" is automatically specified.

Start symbol of function table of function pre/post processing runtime function is as follows.

- ___PROLOG_TABLE

Describe the following code using this symbol.

```
mov     #___PROLOG_TABLE, r12
ldsr    r12, 20
```

CTBP is system register 20. Set a value to it using the ldsr instruction.

7.3.17 Setting of programmable peripheral I/O register value

BPC must be set when using a V850 microcontrollers product in which programmable peripheral I/O registers are provided and using a programmable peripheral I/O register.

For example, the peripheral area select control register of the V850E/IA1 is configured as follows.

Figure 7-1. BPC Register

15 0

PA15	0	PA13	PA12	PA11	PA10	PA9	PA8	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

Table 7-8. BPC Register

Bit Position	Bit Name	Meaning
15	PA15	Enables or disables use of programmable peripheral I/O area. 0: Use of programmable peripheral I/O area disabled. 1: Use of programmable peripheral I/O area enabled.
13-0	PA13-PA0	Sets address of programmable peripheral I/O area.

To set a value to BPC, processing to write a value to the BPC register must be described in the startup routine.

In the case of the V850E/IA1, PA15 is set to 1, and a programmable peripheral I/O area address is set to PA13 to PA0. Set the BPC register, for example, to set the address of the programmable peripheral I/O area to 0x1234 as follows.

```
mov     0x9234, r13
st.h    r13, BPC
```

Because PA15 must be set to 1, set BPC to the logical sum (OR) of 0x1234 and 0x8000.

See the Relevant Device's Architecture User's Manual of each device for details of the programmable peripheral I/O registers.

7.3.18 Setting r6 and r7 as argument of main function

If the main function is defined to have two parameters as follows "int main (int argc, char *argv[ ]) { /* ... */ }", processing that sets a value to the arguments (r6 and r7) must be performed before execution branches to the main function. See "7.3.4 Securing argument area for main function" for how to secure an area.

This processing is not necessary for an application using a real-time OS because the main function is not created.

Processing to set a value to r6 and r7 is as follows.

```
ld.w    $__argc, r6
movea   $__argv, gp, r7
```

The argument area of the main function is allocated to the .data section, so describe an access code in gp- relative mode.

7.3.19 Branching to main function (when not using real-time OS)

When the processing necessary for the startup routine has been completed, execute an instruction that branches to the main function.

However, this processing is not necessary for an application using a real-time OS because the main function is not created. Instead, an instruction that branches to the initialization routine of the realtime OS is necessary. See "7.3.20 Branching to initialization routine of real-time OS (when using real-time OS)" for the details.

Describe the following code to branch to the main function.

```
jarl    _main, lp
```

When the main function has been executed, execution returns to the 4 bytes subsequent to this branch instruction.
The following instruction can also be used if it is known that execution does not return.

```
jr      _main
```

```
mov     #_main, lp
jmp     [lp]
```

The entire 32-bit space can be accessed using the jmp instruction.

When the "jarl_main, lp" instruction is used, execution returns after the main function is executed. It is recommended to take appropriate action to prevent deadlock from occurring when execution returns.

7.3.20 Branching to initialization routine of real-time OS (when using real-time OS)

In an application using a real-time OS, execution branches to the initialization routine when the processing that must be performed by the startup routine has been completed. In an application not using a real-time OS, execution branches to the main function. See "7.3.19 Branching to main function (when not using real-time OS)".

[If RX850V4 is used]

```
        .extern __kernel_sit
        .extern __kernel_start
        mov     #__kernel_sit, r6
        mov     #__kernel_start, r11
        jarl    __jump_kernel_start, lp
__boot_error:
        jbr     __boot_error
__jump_kernel_start:
        jmp     [r11]
```

See the User's Manual of each real-time OS for details.

7.3.21 V850E2V3 multi-core startup routine

Initialize the common module area by the following procedure.

```
__start:
        ld.hu   PEID, r10
        add     -1, r10
        shl     2, r10

        mov32   0xFFFF6900, r11                 /* Get address of MEV0 register */
        mov     1, r12

        caxi    [r11], r0, r12              /* Select PE to initialize common module */
        bnz     .Lsleep                         /* Put other PEs to sleep */

        jarl    _hdwinit, lp

        mov     -1, r7
        mov32   #__S_romp, r6                   /* Initialize common module */
        jarl    __rcopy, lp

        mov32   #___PROLOG_TABLE, r12
        ldsr    r12, 20

        st.w    r0, MEV0                        /* Restore other PEs */
.Lwakeup:
        ld.w    #__table.__ssbss[r10], r6
        ld.w    #__table.__esbss[r10], r7       /* Initialize each PE */
        jarl    __zeroclrw, lp

        mov32   #__exit, lp
        ld.w    #__table._main[r10], r10        /* Branch to main() of each PE module */
        jmp     [r10]
__exit:
        br      __exit
.Lsleep:
        ld.w    MEV0, r12
        cmp     r0, r12                         /* Wait for MEV register to reach 0 */
        bz      .Lwakeup
        br      .Lsleep
```

7.4 Coding Example

This section shows an example of the startup routine.

Table 7-9. Examples of Startup Routine

```
#------------------------------------------------------------------
# external label declaration 1 of symbol reserved for the CX (For tp, gp, ep)
#------------------------------------------------------------------
        .extern __tp_TEXT, 4
        .extern __gp_DATA, 4
        .extern __ep_DATA, 4
#------------------------------------------------------------------
# external label declaration 2 of symbol reserved for the CX (For bss attribute section
# initialization)
# Section deleted if there is a section not used.
# If the section to be used is not determined, write all sections and suppress the
# assemble error of the startup routine that occurs due to addition/deletion of sections.
#------------------------------------------------------------------
        .extern __ssbss, 4
        .extern __esbss, 4
        .extern __sbss, 4
        .extern __ebss, 4
        .extern __ssebss, 4
        .extern __esebss, 4
        .extern __stibss.byte, 4
        .extern __etibss.byte, 4
        .extern __stibss.word, 4
        .extern __etibss.word, 4
        .extern __ssibss, 4
        .extern __esibss, 4
#------------------------------------------------------------------
# external label declaration of symbol reserved for the CX
# Declare start address of function table as external label when
# using function pre/post processing runtime function
#------------------------------------------------------------------
        .extern ___PROLOG_TABLE
#------------------------------------------------------------------
# external label declaration of main function
#------------------------------------------------------------------
        .extern _main
#------------------------------------------------------------------
# argument area of the main function (Unnecessary if void main(void) type is used)
#------------------------------------------------------------------
        .dseg   DATA
        .align  4
__argc:
```

```
        .db4    0
__argv:
        .db4    #.L16
.L16:
        .db     0
        .db     0
        .db     0
        .db     0
#---------------------------------------------------------------------
# The following is dummy data for section generation.
# This dummy data is used to clear the bss attribute section that appears later to zero.
#
# The start symbol and end symbol are generated if data exists in the corresponding section
# during linking.  However, if the section that is to be used is not yet decided, an
# assemble error of startup routine occurs each time when section is added or deleted by
# rewriting the link directive file.  To avoid this, generate the start and end symbols of a
# section by allocating dummy data to the section.
# The bss attribute section is not described because data is allocated by a stack generation
# code and dummy data does not have to be created in that section.
#
# If the section to be used is determined, delete this dummy data and the zero clear routine
# except the necessary part of the routine, this can eliminate waste and enhance the code
# efficiency.
#---------------------------------------------------------------------
.sbss   .dseg   sbss
        .ds(0)
.sebss  .dseg   sebss
        .ds(0)
.tibss.byte     .dseg   tibss.byte
        .ds(0)
.tibss.word     .dseg   tibss.word
        .ds(0)
.sibss  .dseg   sibss
        .ds(0)
#---------------------------------------------------------------------
# securing stack
# securing 0x200 bytes in bss area
#---------------------------------------------------------------------
STACKSIZE       .set    0x200
        .dseg   BSS
#---------------------------------------------------------------------
# reset handler
# describing instructions allocated in reset handler
#---------------------------------------------------------------------
RESET   .cseg   TEXT
```

```
        jr      __start
#------------------------------------------------------------------------
# startup routine entity
#------------------------------------------------------------------------
        .cseg   text
        .align  4
        .public __start
        .public __exit
        .public __startend
__start:
#------------------------------------------------------------------------
# It is assumed that __gp_DATA is set by a symbol directive that uses a relative value
# from tp.  Therefore, gp adds the value of __gp_DATA to tp.
#------------------------------------------------------------------------
        mov     #__tp_TEXT, tp
        mov     #__gp_DATA, gp
        add     tp, gp
        mov     #__stack + STACKSIZE, sp
        mov     #__ep_DATA, ep
#------------------------------------------------------------------------
# Clearing sbss section to zero
# Delete this description to reduce the code if the sbss attribute section is not used.
#------------------------------------------------------------------------
        mov     #__ssbss, r13
        mov     #__esbss, r12
        cmp     r12, r13
        jnl     .L11
.L12:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L12
.L11:
#------------------------------------------------------------------------
# Clearing bss section to zero
# Delete this description to reduce the code if the bss section is not used.
#------------------------------------------------------------------------
        mov     #__sbss, r13
        mov     #__ebss, r12
        cmp     r12, r13
        jnl     .L14
.L15:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
```

```
        jl      .L15
.L14:
#------------------------------------------------------------------
# Clearing sebss section to zero
# Delete this description to reduce the code if the sebss section is not used.
#------------------------------------------------------------------
        mov     #__ssebss, r13
        mov     #__esebss, r12
        cmp     r12, r13
        jnl     .L17
.L18:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L18
.L17:
#------------------------------------------------------------------
# Clearing tibss.byte section to zero
# Delete this description to reduce the code if the tibss.byte section is not used.
#------------------------------------------------------------------
        mov     #__stibss.byte, r13
        mov     #__etibss.byte, r12
        cmp     r12, r13
        jnl     .L20
.L21:
        st.b    r0, [r13]
        add     1, r13
        cmp     r12, r13
        jl      .L21
.L20:
#------------------------------------------------------------------
# Clearing tibss.word section to zero
# Delete this description to reduce the code if the tibss.word section is not used
#------------------------------------------------------------------
        mov     #__stibss.word, r13
        mov     #__etibss.word, r12
        cmp     r12, r13
        jnl     .L23
.L24:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L24
.L23:
#------------------------------------------------------------------
```

```
# Clearing sibss section to zero
# Delete this description to reduce the code if the sibss section is not used
#------------------------------------------------------------------------
        mov     #__ssibss, r13
        mov     #__esibss, r12
        cmp     r12, r13
        jnl     .L26
.L25:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl     .L25
.L26:
#------------------------------------------------------------------------
# setting of function pre/post processing runtime function
# The start address of the library function table is set to CTBP (system register #20).
# All except for V850Ex delete this description.
#------------------------------------------------------------------------
        mov     #___PROLOG_TABLE, r12
        ldsr    r12, 20
#------------------------------------------------------------------------
# programmable peripheral I/O register setting
# Delete this description if a V850 not having programmable peripheral I/O registers.
# Shown below is an example where the BPC register value (set address) is 0x1234.
# The logical sum of 0x1234 (address) and 0x8000 (use of programmable peripheral I/O) is
# set to BPC.
#------------------------------------------------------------------------
PIOADDR .set    0x12340000
USEBPC  .set    0x8000
        mov     USEBPC | (PIOADDR >> 14), r13
        st.w    r13, BPC
#------------------------------------------------------------------------
# setting argument of main function to r6 and r7
#------------------------------------------------------------------------
        ld.w    $__argc, r6
        movea   $__argv, gp, r7
#------------------------------------------------------------------------
# branching to main function
#------------------------------------------------------------------------
        jarl    _main, lp
#------------------------------------------------------------------------
# processing when main function returns
#------------------------------------------------------------------------
__exit:
        br      __exit
__startend:
```

CHAPTER 8 ROMIZATION

This chapter describes an outline of the ROMization procedure, operation method, etc.

8.1 Outline

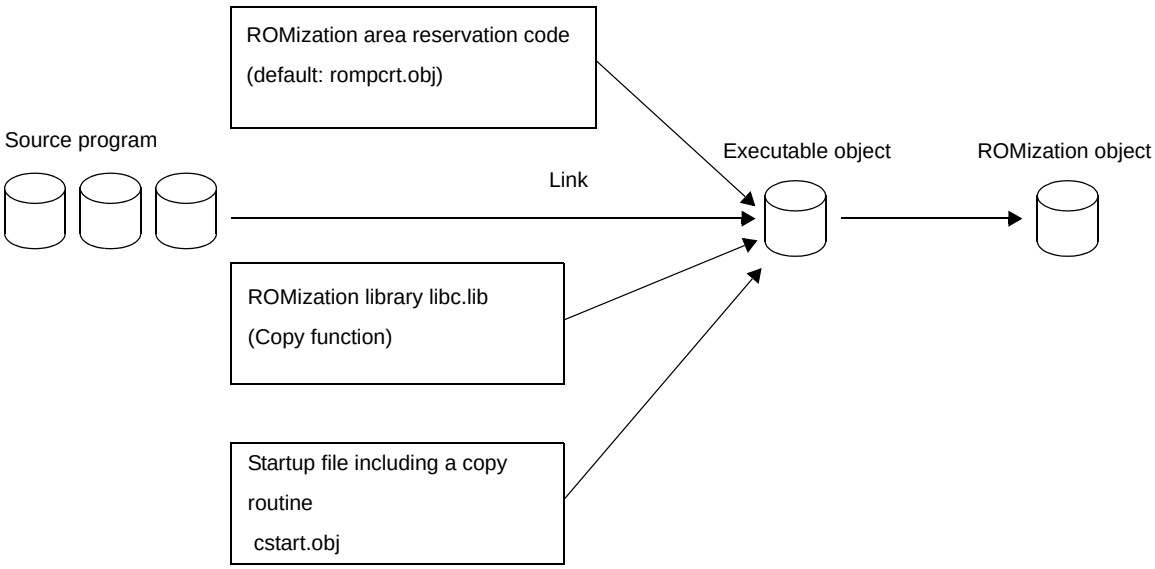
When a variable is declared globally within a program, the variable is allocated to the data-attribute section in RAM if the variable has a initial value, or to the bss-attribute section if it does not have a initial value. When the variable has a initial value, that initial value is also stored in RAM. In addition, program code may be stored in the internal RAM area to speed up applications.

In the case of an embedded system, if a debug tool such as an in-circuit emulator is used, executable modules can be downloaded and executed just as they are in the allocation image. However, if the program is actually written to the target system's ROM area before being executed, the initial value information that has been allocated to the data-attribute section and the program code that has been allocated to a RAM area must be deployed in RAM prior to execution. In other words, data that is residing in RAM must be deployed in ROM, and this means that data must be copied from ROM to RAM before the corresponding application is executed.

The ROMization is to pack information of defaults on a variable of a data-attribute section and the program arranged on the RAM in a single section. This section is allocated in ROM and the initial value information or program code it contains can be easily deployed in RAM by calling the copy function that is provided by the CX.

The following figure shows an outline of the operation flow in creating objects for ROMization.

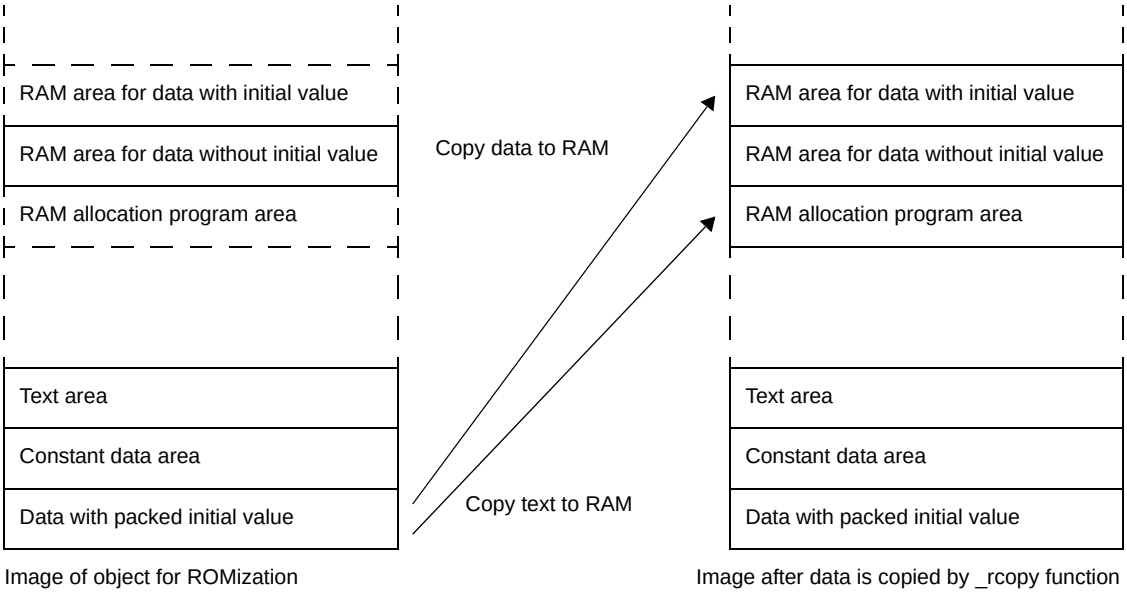
Figure 8-1. Creation of Object for ROMization



When ROMization objects are created as shown in the "Table 8-1. Copy Functions", execution of the _rcopy copies the data to be allocated to RAM from the packed ROM section.

An image of this operation is shown below.

Figure 8-2. Image of Processing Before and After Copy Function Call



The default values for the section name and the section's start address (label name) required for the ROMization object are as follows.

- Name of packed section -> rompsec section
- Start address (l name) of rompsec section -> __S_romp

The function used to copy from the rompsec section to the RAM area is as follows.

- Copy function -> _rcopy, _rcopy1, _rcopy2, _rcopy4

This function is stored in the library "libc.lib" which is in the *Version Folde* \ lib850e folder.

__S_romp is a label that is defined by "rompcrt.obj" in the *Version Folde*\ lib850e folder (the corresponding source file is rompcrt.asm). The rompcrt.obj object module file is used as it automatically creates a rompsec section immediately after (at the 4-byte alignment position) the .text-attribute section. __S_romp becomes the label indicating the start address of that rompsec section.

In addition to this method for automatically creating a rompsec section, it is also possible to independently create and allocate a program corresponding to the rompcrt.asm source file.

During ROMization, once the object for ROMization has been created, it is converted into a hexadecimal file and written to ROM.

If the application does not include any data that requires packing, there is no need to create a ROMization object. Instead, the object created by the linker can be converted directly into a hexadecimal file.

If the object module files resolved for relocation include symbol information and debug information, the CX creates a ROMization object module file without deleting them. Therefore, the debugger can debug the source even with a ROMization object module file.

8.2 rompsec Section

This section explains a rompsec section.

8.2.1 Types of sections to be packed

The default setting for the object that can be packed in a rompsec section is "data allocated to sections having a write-enabled attribute[Note]". In addition, "any section that has either the text attribute or const attribute" can be specified for packing by specifying the -Xrompsec_text option.

Note bss attribute sections and sbss attribute sections that are writeable but which clearly do not have initial values are not packed.

Specific examples of packing targets are listed below.

- Reserved sections (.data, .sdata, .sedata, .sidata, .tidata, .tidata.byte, .tidata.word)
- Sections created with arbitrary names specifying a sdata and data attribute by the .section definition directive in the assembly language program.
- Sections allocated to the internal instruction RAM (can't be packed, when V850E2 core device is specified).

Note, however, that if any user-specified sections with either the text attribute or const attribute are not packed and if the above-listed sections are not in an executable module, there is no need to create a ROMization object.

See the link map file to determine whether or not the reserved sections (.data, .sdata, .sedata, .sidata, .tidata, .tidata.byte, .tidata.word) exist.

It can be confirmed that a rompsec section is created in place of a .data section, .sdata section, sections allocated to an internal RAM (including interrupt handler sections), and the like, by referencing the map file which is generated by the ROMization processing.

Therefore, when sections allocated to the internal instruction RAM (including interrupt handler sections) are packed, the program requires four-byte alignment of the start address of each section.

Additionally, the internal offset in the rompsec section is also 4-byte aligned, so a padding area is created, and this is added to the size of the rompsec section.

8.2.2 Size of rompsec section

This section describes the memory area size to be reserved for the rompsec section.

When creating the ROMization module, note the size of the rompsec section as well as the internal ROM capacity of the target CPU and the address range and size of the target system's ROM area.

Describe the link directive file carefully to prevent the rompsec section from overlapping other sections.

Remark See "8.3 Creating ROMized Load Module File" for details about the code example of the link directive file.

Formulas used to calculate the size of the rompsec section are shown below.

size-of-rompsec-section (byte, in decimal numbers)
= 8 + 12 * number-of-ROMization-sections + total-size-of-ROMization-sections + padding-size[Note]

Note The padding size is 0 to 3 bytes per section, depending on the alignment condition of the section subject to ROMization.

For example, if .data and .sdata sections exist, the size of each is 1002 bytes and 1000 bytes, and the alignment condition of each section is 4 bytes, the size of the rompsec section is as follows.

8 + 12 * 2 + 1002 + 1000 + 2 = 2036 (bytes)

8.2.3 rompsec section and link directive

The CX links the ROMization area reservation code file (rompcrt.obj) last to add the rompsec section immediately after the .text section when performing ROMization.

Therefore, the rompsec section does not have to be allocated by the following link directive.

The link directive taking ROMization processing into consideration is shown below.

```
SCONST : !LOAD ?R {                    # Allocates SCONST, CONST, and TEXT to internal ROM
        .sconst = $PROGBITS ?A .sconst;
};

CONST : !LOAD ?R {
        .const = $PROGBITS ?A .const;
};

TEXT : !LOAD ?RX {
        .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
        .text = $PROGBITS ?AX .text;
        rompsec = $PROGBITS ?AX rompsec # Allocates .text to end of internal ROM
};

DATA : !LOAD ?RW V0x100000 {           # Allocates DATA to external RAM
        .data = $PROGBITS ?AW;
        .sdata = $PROGBITS ?AWG;
        .sbss = $NOBIT ?AWG;
        .bss = $NOBIT ?AW;
};

SIDATA : !LOAD ?RW V0xFFE000 {         # Allocates SIDATA to internal RAM
        .sidata = $PROGBITS ?AW .sidata;
        .sibss = $NOBITS ?AWG .sibss;
};

__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

If the rompsec section exceeds the internal ROM area, the message is output and the processing is stopped.

Remark By specifying the -Xromize_check_off=rom_less option, the internal ROM area may be ignored.
By specifying the -Xromize_check_off option, it is possible to continue processing, while outputting a message.

The above check is not performed if the rompsec section is allocated to the end of the external ROM area. Check the memory map file to see if the sections fit in ROM.

Remark The memory map file can be output by specifying the -Xmap option.

If it is necessary to allocate the rompsec section in the middle of ROM, check the area where the rompsec section is to be allocated as follows, from the size and allocation address of the rompsec section, and specify an appropriate address for the segment immediately after the rompsec section.

Figure 8-3. Link Directive Taking ROMization Processing into Consideration (Size Considered)

```
#Allocates SCONST, CONST, and TEXT to internal ROM
SCONST: !LOAD ?R {
        .sconst = $PROGBITS ?A .sconst;
};
#Allocates .text in middle of internal ROM
#rompsec between TEXT and CONST
TEXT:   !LOAD ?RX {
        .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
        .text = $PROGBITS ?AX .text;
        rompsec = $PROGBITS ?AX rompsec;
};
#Allocates CONST to end of internal ROM by specifying address taking size into consideration
CONST:  !LOAD ?R Vx3f800 {
        .const = $PROGBITS ?A .const;
};
#Allocates DATA to external RAM
DATA:   !LOAD ?RX V0x100000 {
        .data = $PROGBITS ?AW;
        .sdata = $PROGBITS ?AWG;
        .sbss = $NOBIT ?AWG;
        .bss = $NOBIT ?AW;
};
#Allocates SIDATA to internal RAM
SIDATA: !LOAD ?RX V0xFFE000 {
        .sidata = $PROGBITS ?AW .sidata;
        .sibss = $NOBIT ?AWG .sibss;
};
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

8.3 Creating ROMized Load Module File

This section explains how to create the ROMized load module.

8.3.1 Procedure for creating ROMized load module (default)

This section describes the method that uses the ROMization area reservation code file (rompcrt.obj) that is provided by default.

(1) Calling the copy function

In the startup routine, add code to start the copy function _rcopy() with the necessary arguments, and create the object module file.

In the CX, this code is included in the standard startup routine, so it has no particular meaning if you perform ROMization by default.

Even if you suppress ROMization via the -Xno_romize option, this has no meaning because it switches to the startup file entered by the driver.

Note, however, that if _rcopy2/_rcopy4 was used instead of _rcopy, then if you wish to specify a section to copy, you must overwrite the startup routine, and create cstart.asm from cstart.obj.

Figure 8-4. Example of Using Copy Function _rcopy (cstrat.asm)

```
.extern _hdwinit
.extern __S_romp
.extern __rcopy
        :
jarl    _hdwinit, lp
        :
mov32   #__S_romp, r6
mov     -1, r7
jarl    __rcopy, lp
```

Remarks 1. See "8.4 Copy Functions" for details about copy functions.

2. When overwriting the startup routine, change the startup routine to be used specifying the -Xno_startup or -Xstartup options.

(2) Specify the allocation of the rompsec section

The cx links the ROMization area reservation code file (rompcrt.obj) last to add the rompsec section immediately after the .text section when performing ROMization.

Therefore, the rompsec section does not have to be allocated by the link directive.

Remark See "8.2.3 rompsec section and link directive" for details.

(3) Area secured for rompsec section

Secure memory area for the "romspec" section, and create an object module file indicating its start address.

This also has no meaning, because the standard ROMized load module is linked by default.

In the example above, the label "__S_romp" generates code indicating the first (4-byte aligned) address exceeding the end of the text section (section name defined in the code above) in the object module file, as an absolute address.

(4) Linking

The driver controls linking so that files are specified in the following order.

- cstart.obj (Object module file of startup routine)
- Object module file specified by user
- libc.lib (Standard library including hdwinit function and _rcopy function)
- rompcrt.obj (ROMization area reservation code file)

A memory area for the rompsec section is secured immediately after the ".text" section by linking[Note] the rompcrt.obj last.

Remark If the -Xrescan option is specified, the library file will be linked after rompcrt.obj, and an error will occur during ROMization. In such a case, explicitly secure a rompsec section area.
See "8.2.3 rompsec section and link directive" for details.

(5) ROMization

ROMization generates an object file with a "romspec" section, instead of a section with a data or sdata attribute (indicating that it is to be allocated to ROM with an initial value), or a section allocated to internal instruction RAM (all sections specified for allocation to internal instruction RAM via a link directive, such as an interrupt handler section).

(6) Hex processing

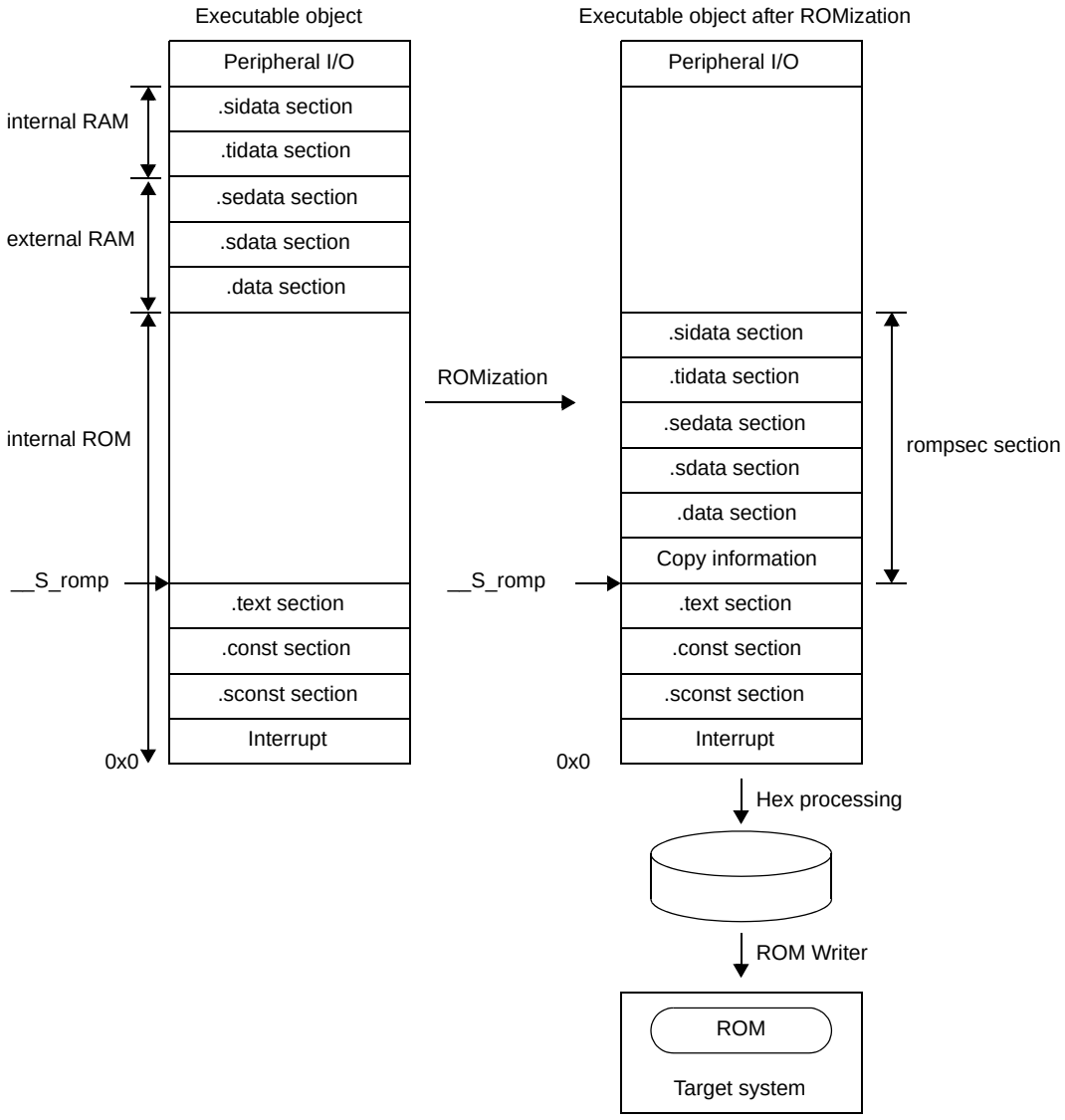
Hex processing creates hex data.

The hex process is called automatically from the driver. You can specify the output file name by specifying the -Xhex option.

(7) Downloading to ROM

Load the created hex data to the ROM of the target system.

Figure 8-5. ROMization Image (Default)



8.3.2 Procedure for creating ROMized load module (customize)

This section describes the method for independently creating the ROMization area reservation code file and determining the desired start address and allocation position of the .rompack section.

(1) Describe ROMization area reservation code file

Describe the code corresponding to default ROMization area reservation code "rompcrt.asm".
In this section, it is assumed that the source file name of the ROMization area reservation code file is "rompack.asm" and the name of the symbol indicating the start of the ROMization area is "__rompack".
In addition, it is assumed that the section containing this symbol is ".rompack section".
In this case, the code in "rompack.asm" appears as follows.

Example rompack.asm

```
.rompack        .cseg   text
        .align  4
        .public __rompack, 4
__rompack:
```

After describing rompack.asm, it's assembled and object file rompack.obj of ROMization area reservation code file is generated.

(2) Call a copy function

Call a copy function within the startup routine.

Example Call copy function "_rcopy"

```
.extern _hdwinit
.extern __rompack
.extern __rcopy
        :
jarl    _hdwinit, lp
        :
mov32   #__rompack, r6
mov     -1, r7
jarl    __rcopy, lp
```

Remarks 1. See "8.4 Copy Functions" for details about copy functions.

2. When using other than the standard startup routine, change the startup routine to be used specifying the -Xno_startup or -Xstartup options.

(3) Specify the allocation of the rompack section

Define the created .rompack section in the link directive.
The allocation location of the .rompack section can be determined arbitrarily by specifying an address simultaneously.
To specify ROMPACK as the segment containing the .rompack section and to allocate that segment to at address 0x3000, enter the following link directive.

```
TEXT:    !LOAD ?RX V0x1000 {
        .text = $PROGBITS ?AX .text;
};

ROMPACK: !LOAD ?RX V0x3000 {
        .rompack = $PROGBITS ?AX .rompack;
};
                :
```

Estimate the .rompack section's size using the formula described in "8.2.2 Size of rompsec section" to avoid the ROMPACK segment's allocation address from overlapping with adjacent segments and reflect the size to the link directive file.

(4) Specify the ROMization area reservation code file

Specify ROMization area reservation code file "rompack.obj" by the -Xrompcrt option.

(5) Specify the start label of the rompsec section

Specify "__rompack" as the parameter of the -Xrompsec_start option.
This will generate code indicating the same addresses for the __rompack label and the .rompack section.

(6) Linking

The driver controls linking so that files are specified in the following order.

- cstart.obj (Object module file of startup routine)
- Object module file specified by user
- libc.lib (Standard library including hdwinit function and _rcopy function)
- rompack.obj (ROMization area reservation code file)

A memory area for the .rompack section is secured immediately after the ".text" section by linking[Note] the rompack.obj last.

Remark If the -Xrescan option is specified, the library file will be linked after rompack.obj, and an error will occur during ROMization. In such a case, explicitly secure a .rompack section area.
See "8.2.3 rompsec section and link directive" for details.

(7) ROMization

ROMization generates an object file with a ".rompack" section, instead of a section with a data or sdata attribute (indicating that it is to be allocated to ROM with an initial value), or a section allocated to internal instruction RAM (all sections specified for allocation to internal instruction RAM via a link directive, such as an interrupt handler section).

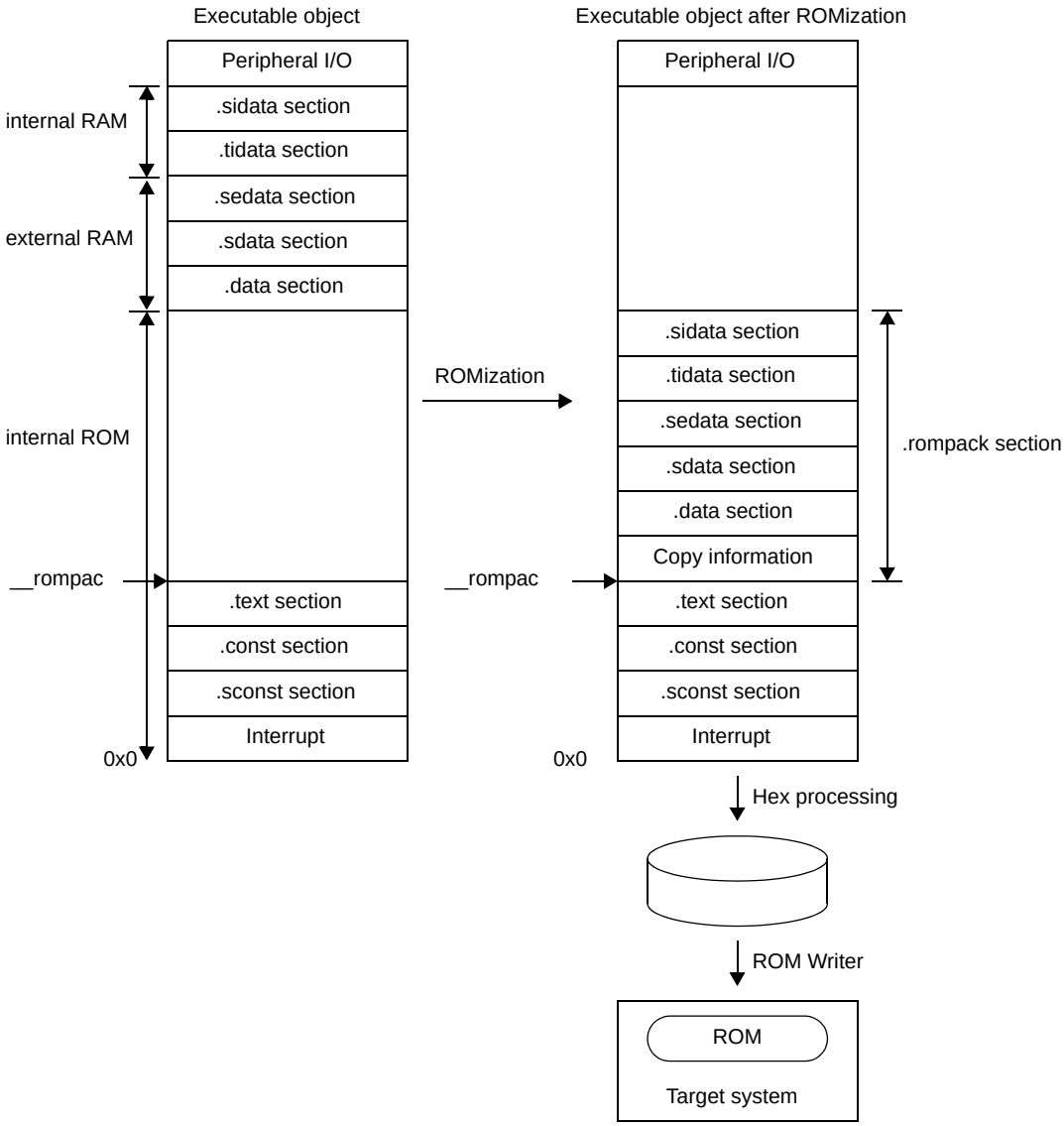
(8) Hex processing

Hex processing creates hex data.
The hex process is called automatically from the driver. You can specify the output file name by specifying the -Xhex option.

(9) Downloading to ROM

Load the created hex data to the ROM of the target system.

Figure 8-6. ROMization Image (Customize)



8.4 Copy Functions

This section describes the copy functions necessary for the program to be stored in ROM.

Table 8-1. Copy Functions

Function Name	Function
_rcopy	Copies Packing data in the unit of 1 byte to RAM (Same as _rcopy1)
_rcopy1	Copies Packing data in the unit of 1 byte to RAM (Same as _rcopy)
_rcopy2	Copies Packing data in the unit of 2 bytes to RAM
_rcopy4	Copies Packing data in the unit of 4 bytes to RAM

Use 1-byte, 2-byte, or 4-byte transfer, depending on the specification of the RAM at the transfer destination.

_rcopy

Copies default data or RAM text[Note] (1 byte).

Note Data section with initial value which is to be allocated to RAM, and text section for internal RAM.

[Classification]

ROMization library

[Syntax]

int _rcopy(const unsigned long * *label*, long *number*);
extern const unsigned long _S_romp[Note];

Note _S_romp is the start address of the packing data.

[Return value]

0	Normal completion (if copied correctly)
-1	Abnormal termination (if not copied correctly)

[Description]

_rcopy(&label, number) copies the initial value data of section number *number* to be copied, or text to be allocated to RAM, to the RAM area 1 byte at a time, based on the information in the rompsec section allocated starting at the address following the address indicated by *label*. If -1 is specified as *number*, all sections in the rompsec section are copied. Section number *number* is a positive number that starts from 1.

By default, sections are allocated in the order in which they appear in the input file.

[Caution]

- _rcopy copies data in accordance with the information generated by the ROMization. When this function is executed, processing which adds an offset value to address of the copy destination can't be done.
- No data is copied if data may be overwritten as a result of copying.
- Specify a global label having an absolute value or an absolute address as the first argument of _rcopy, *label*. If any other value or address is specified, the result is not guaranteed.
- The copy is not performed unless the 4 bytes following the address indicating the *label* contain a magic number indicating that the object was generated via ROMization.
- The section number to be specified as the *number* is a positive number.
  See the memory map file for the relation between the section name and section number.
- The copy is not performed if "*number*" is a section number, or any value other than "-1".
- If there is more than one RAM, and multiple copy routines are used separately, specifying "-1" in *number* will send all of the functions to each copy routine multiple times. The copy will thus not be performed correctly, due to section-alignment and other issues. In this case, specify a section number for "*number*", rather than "-1".
- If "-1" is specified in *number*, the copy is performed in section-number order. If there are any sections that are not copied due to the problem above, a value of -1 will be returned, and sections after the problem section will not be copied.
- The _rcopy and _rcopy1 functions are identical in feature.

[Example]

```
main(){
        _rcopy( &_S_romp, -1 );
}
```

_rcopy1

Copies default data or RAM text[Note] (1 byte).

Note Data section with initial value which is to be allocated to RAM, and text section for internal RAM.

[Classification]

ROMization library

[Syntax]

int _rcopy1(const unsigned long * *label*, long *number*);
extern const unsigned long _S_romp[Note];

Note _S_romp is the start address of the packing data.

[Return value]

0	Normal completion (if copied correctly)
-1	Abnormal termination (if not copied correctly)

[Description]

_rcopy1(&label, number) copies the initial value data of section number *number* to be copied, or text to be allocated to RAM, to the RAM area 1 byte at a time, based on the information in the rompsec section allocated starting at the address following the address indicated by *label*. If -1 is specified as *number*, all sections in the rompsec section are copied. Section number *number* is a positive number that starts from 1.

By default, sections are allocated in the order in which they appear in the input file.

[Caution]

- _rcopy1 copies data in accordance with the information generated by the ROMization. When this function is executed, processing which adds an offset value to address of the copy destination can't be done.
- No data is copied if data may be overwritten as a result of copying.
- Specify a global label having an absolute value or an absolute address as the first argument of _rcopy, *label*. If any other value or address is specified, the result is not guaranteed.
- The copy is not performed unless the 4 bytes following the address indicating the *label* contain a magic number indicating that the object was generated via ROMization.
- The section number to be specified as the *number* is a positive number.
  See the memory map file for the relation between the section name and section number.
- The copy is not performed if "*number*" is a section number, or any value other than "-1".
- If there is more than one RAM, and multiple copy routines are used separately, specifying "-1" in *number* will send all of the functions to each copy routine multiple times. The copy will thus not be performed correctly, due to section-alignment and other issues. In this case, specify a section number for "*number*", rather than "-1".
- If "-1" is specified in *number*, the copy is performed in section-number order. If there are any sections that are not copied due to the problem above, a value of -1 will be returned, and sections after the problem section will not be copied.
- The _rcopy1 and _rcopy functions are identical in feature.

_rcopy2

Copies default data or RAM text[Note] (2 bytes).

Note Data section with initial value which is to be allocated to RAM, and text section for internal RAM.

[Classification]

ROMization library

[Syntax]

int _rcopy2(const unsigned long * *label*, long *number*);
extern const unsigned long _S_romp[Note];

Note _S_romp is the start address of the packing data.

[Return value]

0	Normal completion (if copied correctly)
-1	Abnormal termination (if not copied correctly)

[Description]

_rcopy2(&label, number) copies the initial value data of section number number to be copied, or text to be allocated to RAM, to the RAM area 2 bytes at a time, based on the information in the rompsec section allocated starting at the address following the address indicated by *label*. If -1 is specified as *number*, all sections in the rompsec section are copied. Section number *number* is a positive number that starts from 1.

By default, sections are allocated in the order in which they appear in the input file.

[Caution]

- The copy will not be performed unless the start address of the copy source (offset in the "romspec" section) and the start address of the copy destination are 2-byte aligned.
- If the size of the section to copy is not a multiple of 2, then the padding area immediately after the end of the section is copied in addition to the final odd byte. 2-byte align the section that follows, or copy in ascending address order, so that following sections are not overwritten.
- _rcopy2 copies data in accordance with the information generated by the ROMization. When this function is executed, processing which adds an offset value to address of the copy destination can't be done.
- No data is copied if data may be overwritten as a result of copying.
- Specify a global label having an absolute value or an absolute address as the first argument of _rcopy2, *label*. If any other value or address is specified, the result is not guaranteed.
- The copy is not performed unless the 4 bytes following the address indicating the *label* contain a magic number indicating that the object was generated via ROMization.
- The section number to be specified as the *number* is a positive number.
  See the memory map file for the relation between the section name and section number.
- The copy is not performed if "*number*" is a section number, or any value other than "-1".
- If there is more than one RAM, and multiple copy routines are used separately, specifying "-1" in *number* will send all of the functions to each copy routine multiple times. The copy will thus not be performed correctly, due to section-alignment and other issues. In this case, specify a section number for "*number*", rather than "-1".

- If "-1" is specified in *number*, the copy is performed in section-number order. If there are any sections that are not copied due to the problem above, a value of -1 will be returned, and sections after the problem section will not be copied.

_rcopy4

Copies default data or RAM text[Note] (4 bytes).

Note Data section with initial value which is to be allocated to RAM, and text section for internal RAM.

[Classification]

ROMization library

[Syntax]

int _rcopy4(const unsigned long * *label*, long *number*);
extern const unsigned long _S_romp[Note];

Note _S_romp is the start address of the packing data.

[Return value]

0	Normal completion (if copied correctly)
-1	Abnormal termination (if not copied correctly)

[Description]

_rcopy4(&label, number) copies the initial value data of section number number to be copied, or text to be allocated to RAM, to the RAM area 4 bytes at a time, based on the information in the rompsec section allocated starting at the address following the address indicated by *label*. If -1 is specified as *number*, all sections in the rompsec section are copied. Section number *number* is a positive number that starts from 1.

By default, sections are allocated in the order in which they appear in the input file.

[Caution]

- The copy will not be performed unless the start address of the copy source (offset in the "romspec" section) and the start address of the copy destination are 4-byte aligned.
- If the size of the section to copy is not a multiple of 4, then the padding area immediately after the end of the section is copied in addition to the final odd byte. 4-byte align the section that follows, or copy in ascending address order, so that following sections are not overwritten.
- s_rcopy4 copies data in accordance with the information generated by the ROMization. When this function is executed, processing which adds an offset value to address of the copy destination can't be done.
- No data is copied if data may be overwritten as a result of copying.
- Specify a global label having an absolute value or an absolute address as the first argument of _rcopy4, *label*. If any other value or address is specified, the result is not guaranteed.
- The copy is not performed unless the 4 bytes following the address indicating the *label* contain a magic number indicating that the object was generated via ROMization.
- The section number to be specified as the *number* is a positive number.
  See the memory map file for the relation between the section name and section number.
- The copy is not performed if "*number*" is a section number, or any value other than "-1".
- If there is more than one RAM, and multiple copy routines are used separately, specifying "-1" in *number* will send all of the functions to each copy routine multiple times. The copy will thus not be performed correctly, due to section-alignment and other issues. In this case, specify a section number for "*number*", rather than "-1".

- If "-1" is specified in *number*, the copy is performed in section-number order. If there are any sections that are not copied due to the problem above, a value of -1 will be returned, and sections after the problem section will not be copied.

CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER

This chapter explains how to handle arguments when a program is called by the CX.

9.1 Method of Accessing Arguments and Automatic Variables

(1) Argument passed to assembler function

The CX stores 4-word arguments in argument registers r6 to r9 and arguments in excess of 4 words in the stack frame of the calling function. Reference each stored value when using an argument value in an assembler function.

If the assembler function returns a structure, the CX stores 3-word arguments in argument registers r7 to r9 and arguments in excess of 3 words in the stack frame of the calling function. Note the argument storage location because the address where a return value is stored is stored in r6 register.

An argument value in a C function is the value itself that is specified as an argument. The operation of the C function is not affected even if this value is changed in an assembler function.

(2) Argument passed to C function

The CX stores 4-word arguments in argument registers r6 to r9 and arguments in excess of 4 words in the stack frame of the calling function. Store the arguments in excess of 4 words upward from the address indicated by SP.

If the C function returns a structure, the CX stores 3-word arguments in argument registers r7 to r9 and arguments in excess of 3 words in the stack frame of the calling function. And the address where a return value is stored is stored in r6 register.

9.2 Method of Storing Return Value

(1) Return value returned from assembler function

The CX generates codes on the assumption that the return value of a function is stored in the r10 register. Store the value returned from an assembler function in r10.

If the function returns a structure, the return value, i.e., the structure, is stored in the stack frame of the calling function.

(2) Return value returned from C function

The CX generates codes on the assumption that the return value of a function is stored in the r10 register. Reference the r10 register when using the value returned from a C function.

If the function returns a structure, a value is stored in an area for the return value of the calling function, and a code that passes the address of that area as an argument is output. An area for the return value must be allocated in advance on the calling side.

9.3 Calling of Assembly Language Routine from C Language

This section explains the points to be noted when calling an assembler function from a C function.

(1) Identifier

If external names, such as functions and external variables, are described in the C source by the CX, they are prefixed with "_" (underscore) when they are output to the assembler.

Table 9-1. Identifier

C	Assembler
func1 ( )	_func1

Prefix "_" to the identifier when defining functions and external variables with the assembler and remove "_" when referencing them from a C function.

(2) Stack frame

The CX generates codes on the assumption that the stack pointer (SP) always indicates the lowest address of the stack frame. Therefore, the address area lower than the address indicated by SP can be freely used in the assembler function after branching from a C source to an assembler function. Conversely, if the contents of the higher address area are changed, the area used by a C function may be lost and the subsequent operation cannot be guaranteed. To avoid this, change SP at the beginning of the assembler function before using the stack.
At this time, however, make sure that the value of SP is retained before and after calling.
When using a register variable register in an assembler function, make sure that the register value is retained before and after the assembler function is called. In other words, save the value of the register variable register before calling the assembler function, and restore the value after calling.
The register for register variable that can be used differ depending on the register mode.

Table 9-2. Registers for Register Variables

Register Modes	Register for Register Variable
22-register mode	r25, r26, r27, r28, r29
26-register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

(3) Return address passed to C function

The CX generates codes on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to an assembler function, the return address of the function is stored in lp. Execute the jmp [lp] instruction to return to a C function.

9.4 Calling of C Language Routine from Assembly Language

This section explains the points to be noted when calling a C function from an assembler function.

(1) Stack frame

The CX generates codes on the assumption that the stack pointer (SP) always indicates the lowest address of the stack frame. Therefore, set SP so that it indicates the higher address of an unused area of the stack area before branching from an assembler function to a C function. This is because the stack frame is allocated towards the lower addresses.

(2) Work register

The CX retains the values of the register for register variable before and after a C function is called but does not retain the values of the work registers. Therefore, do not leave a value that must be retained assigned to a work register.

The register for register variable and work registers that can be used differ depending on the register mode.

Table 9-3. Registers for Register Variables

Register Modes	Register for Register Variable
22-register mode	r25, r26, r27, r28, r29
26-register mode	r23, r24, r25, r26, r27, r28, r29
32-register mode	r20, r21, r22, r23, r24, r25, r26, r27, r28, r29

Table 9-4. Work Register

Register Modes	Work Register
22-register mode	r10, r11, r12, r13, r14
26-register mode	r10, r11, r12, r13, r14, r15, r16
32-register mode	r10, r11, r12, r13, r14, r15, r16, r17, r18, r19

(3) Return address returned to assembler function

The CX generates codes on the assumption that the return address of a function is stored in link pointer lp (r31). When execution branches to a C function, the return address of the function must be stored in lp.

Execution is generally branched to a C function using the jarl instruction.

9.5 Reference of Argument Defined by Other Language

The method of referring to the variable defined by the assembly language on the C language is shown below.

Example Programming of C Language]

```
extern  char    c;
extern  int     i;

void subf() {
        c = 'A';
        i = 4;
}
```

The CX assembler performs as follows.

```
        .public  _i
        .public  _c
        .dseg   SDATA
_i:
        .db4    0x0
_c:
        .db     0x0
```

CHAPTER 10 CAUTIONS

This chapter explains the points to be noted when using the CX.

10.1 Delimiting Folder/Path

Both "\" and "/" are regarded as the delimiters of a folder.

10.2 Mixing with K&R Format in Function Declaration/Definition

If the K&R format and ANSI standard format exist together in the declaration and definition of a function, an error may occur on compilation by the CX as a result of argument expansion processing in the K&R format.

For example, a function is declared according to the ANSI standard in the example below, but the function is defined in the K&R format. Consequently, the types of the arguments do not match, and the CX outputs a "function redeclaration" error.

Example Error

```
void    func(int a, int b, float c);
/*Declared in ANSI standard format.*/
/*Third argument is declared as float type.*/
  :
void func(a, b, c)
int     a, b;
float   c;
{
    /*Defined in K&R format.*/
    /*Third argument is the expanded default of K&R and so becomes double type.*/
      :
}
```

In the above example, compilation is performed normally if the K&R format is uniformly used by specifying "void func();" for the function declaration, or if the ANSI standard format is used by specifying "void func(int a, int b, float c)" for the function definition.

Note, however, that use of the ANSI standard format is recommended in the CX.

10.3 Output of Other Than Position-Independent Codes

Basically, the CX outputs codes not dependent on positions (position-independent codes). However, it outputs the following codes in response to the "initialization statement with an initial value other than a numeric value for a pointer type variable other than an automatic variable".

Example

```
[Description of C Language]
char    *ptr = "test\n";
```

```
[Output codes
LL20    .ds     (6)
LL20:
        .db     "test\n\0"
        .align  4
        .public _ptr, 4
_ptr:
        .db4    #LL20   --Absolute address reference of label
```

10.4 Library File Search by Specifying Option

The CX does not display a message even if a specified library file has not been found as a result of a library file search-Note initiated by an option (-L or -l). However, if the library file name has been directly specified on the command line or in the command file, a message is displayed.

Note If the -L option is not specified, the standard folder (*Version Folder*\lib850e) is searched.

Example

```
> cx -Cf3507 a.c usr.lib
```

```
F0560001: can not open input file"usr.lib".
```

10.5 Volatile Qualifier

When a variable is declared with the volatile qualifier, the variable is not optimized and optimization for assigning the variable to a register is no longer performed. When a variable with volatile specified is manipulated, a code that always reads the value of the variable from memory and writes the value to memory after the variable is manipulated is output. The access width of the variable with volatile specified is not changed.

A variable for which volatile is not specified is assigned to a register as a result of optimization and the code that loads the variable from the memory may be deleted. When the same value is assigned to variables for which volatile is not specified, the instruction may be deleted as a result of optimization because it is interpreted as a redundant instruction. The volatile qualifier must be specified especially for variables that access a peripheral I/O register, variables whose value is changed by interrupt servicing, or variables whose value is changed by an external source. When a peripheral I/O register is accessed using the #pragma ioreg directive, however, the CX internally outputs a code for which volatile is specified. Therefore, volatile declaration is not necessary.

The following problem may occur if volatile is not specified where it should.

- The correct calculation result cannot be obtained.
- Execution cannot exit from a loop if the variable is used in a for loop.

If it is clear that the value of a variable with volatile specified is not changed from outside in a specific section, the code can be optimized by assigning the unchanged value to a variable for which volatile not specified and referencing it, which may increase the execution speed.

Example Source and output code if volatile is not specified

If volatile is not specified for "variable a", "variable b", and "variable c", these variables are assigned to registers and optimized. For example, even if an interrupt occurs in the meantime and the variable value is changed by the interrupt, the changed value is not reflected.

```
int a;
int b;
int c;

void func(void) {
        if(a <= 0) {
                b++;
        } else {
                c++;
        }
        b++;
        c++;
}
```

```
_func:
        #@B_PROLOGUE
        #@E_PROLOGUE
        ld.w    $_a, r12
        cmp     r0, r12
        jgt     .L2
        ld.w    $_b, r11
        ld.w    $_c, r10
        add     1, r11
        jbr     .L3
.L2:
        ld.w    $_c, r10
        ld.w    $_b, r11
        add     1, r10
.L3:
        addi    1, r11, r13
        st.w    r13, $_b
        addi    1, r10, r14
        st.w    r14, $_c
        #@B_EPILOGUE
        jmp     [lp]
        #@E_EPILOGUE
```

Example Source and output code if volatile is specified

If volatile is specified for "variable a", "variable b", and "variable c", a code that always reads the values of these variables from memory and writes them to memory after the variables are manipulated is output. For example, even if, an interrupt occurs in the meantime and the values of the variables are changed by the interrupt, the result in which the change is reflected can be obtained. (In this case, interrupts may have to be disabled while the variables are manipulated, depending on the timing of the interrupt.)

When volatile is specified, the code size increases compared with when volatile is not specified because the memory has to be read and written.

```
volatile int    a;
volatile int    b;
volatile int    c;

void func(void) {
        if(a <= 0) {
                b++;
        } else {
                c++;
        }
        b++;
        c++;
}
```

```
func:
        #@B_PROLOGUE
        #@E_PROLOGUE
        ld.w    $_a, r10
        cmp     r0, r10
        jgt     .L2
        ld.w    $_b, r11
        add     1, r11
        st.w    r11, $_b
        jbr     .L3
.L2:
        ld.w    $_c, r12
        add     1, r12
        st.w    r12, $_c
.L3:
        ld.w    $_b, r13
        add     1, r13
        st.w    r13, $_b
        ld.w    $_c, r14
        add     1, r14
        st.w    r14, $_c
        #@B_EPILOGUE
        jmp     [lp]
        #@E_EPILOGUE
```

10.6 Extra Brackets in Function Declaration

If extra brackets "( )" are described in the function declaration, ANSI-C prescribes their handling as shown below, but the CX outputs an error.

Example

```
typedef int Int;

void    f1((Int));
```

[Prescription in ANSI-C]

In a parameter declaration, a single type definition name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

The above example is therefore interpreted according to ANSI-C.

```
void    f(int (*)(int));
```

If the code includes extra brackets, delete the unnecessary brackets as shown below.

Example

```
typedef int Int;

void    f1(Int);
```

APPENDIX A INDEX

S

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Oct 01, 2010	-	First Edition issued

CubeSuite Ver.1.40 User's Manual: Coding for CX Compiler

Publication Date: Rev.1.00 Oct 1, 2010

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.0

CubeSuite Ver.1.40

Renesas Electronics Corporation

R20UT0259EJ0100