To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

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.

SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Compiler Use guide] #pragma Extension Guide

This document explains the extended #pragma directives available in SuperH RISC engine C/C++ Compiler version 9. Some of these directives can reduce the program size or improve execution speed while others provide useful functions.

Table of contents

1. Extended #pragma Directives for Reducing Program Size and Improving Execution Speed

This chapter explains extended #pragma directives that can be effective in reducing program size and improving execution speed. Table 1-1 lists the extended #pragma directives explained in this chapter.

Table 1-1 Extended #pragma directives for improving performance

No.	#pragma	Explanation	Effectiveness on size	Effectiveness on speed	See section
1	#pragma abs16 #pragma abs20 #pragma abs28 #pragma abs32	Specifies address range	A+	B	1.1
2	#pragma inline	Performs inline expansion of functions	C	A+	1.2
3	#pragma inline_asm	Expands an assembly-language description function	C	A	1.3
4	#pragma regsave #pragma noregsave #pragma noregalloc	Generates or does not generate save and restore code at the start and end of functions	A	A	1.4
5	#pragma global_register	Allocates global variables to registers	B	B	1.5
6	#pragma gbr_base #pragma gbr_base1	Specifies GBR base variables	A+	A	1.6

A+: Very effective.
A: Effective.
B: Effective, but must be used with caution.
C: Lowers performance.

Note that the expanded assembly code examples in this document were obtained by specifying `code=asmcode` and `cpu=sh2`. This code might vary depending on the specification of the `cpu` option. It is also subject to change if the compiler is improved in the future. Accordingly, you should use these code examples for reference only.

1.1 Specifies address range

The `#pragma abs`*n* (*n*: `16`, `20`, `28`, or `32`) directive is a declaration that tells the compiler that the variable or function is in the *n*-bit address area. The default is the 32-bit address area.

For example, `#pragma abs16` specified for a variable or function means that the variable or function is placed in an address area that can be represented by 16 bits. Compared with the default 32-bit addressing, which uses four bytes for an address value, 16-bit addressing uses only two bytes, and therefore reduces program size. Address area specification for variables and functions that are referenced from many locations effectively reduces program size. Note that if a `#pragma abs`*n* directive is specified for a variable or function, the same `#pragma abs`*n* directive must be specified for all occurrences of the variable or function throughout the source code. For example, you must not specify `#pragma abs16` and `#pragma abs32` for separate occurrences of the same variable or function. Renesas recommends that you specify the `#pragma abs`*n* directive in a common header file.

You can specify `#pragma abs20` and `#pragma abs28` for only the SH-2A and SH2A-FPU microcomputers. For details, see *2.1 20-bit Long Immediate Load* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler use guide] SH-2A / SH2A-FPU*.

You can also use an advanced option to specify the address area. If you specify both the advanced option and the `#pragma` directive that specify the address area, the `#pragma` directive takes precedence. As an example of use, when the 16-bit address area is specified for a variable or function, you can use the `#pragma abs32` directive to change the address area to the 32-bit address area, which is the default.

Format:

#pragma abs16 (*identifier* [,*identifier*...])
#pragma abs20 (*identifier* [,*identifier*...])
#pragma abs28 (*identifier* [,*identifier*...])
#pragma abs32 (*identifier* [,*identifier*...])
***identifier*: variable name | function name**

Example:

When `abs16` is specified for a variable or function, the address storage area of the variable or function changes from a `.DATA.L` (4 bytes) to a `.DATA.W` (2 bytes).

Source code with #pragma abs16 not specified:

```
extern int x(void);
int y;
long z;
void f(void)
{
z = x() + y;
}
```

Expanded assembly code:

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11+2,R2    ; _x
          JSR         @R2
          NOP
          MOV.L       L11+6,R5    ; _y
          MOV.L       L11+10,R4   ; _z
          MOV.L       @R5,R1      ; y
          ADD         R1,R0
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R4      ; z
L11:
          .RES.W      1
          .DATA.L     _x
          .DATA.L     _y
          .DATA.L     _z
```

Source code with #pragma abs16 specified:

```
#pragma abs16 (x,y,z)
extern int x(void);
int y;
long z;
void f(void)
{
z  = x() + y;
}
```

Expanded assembly code:

```
_f:
          STS.L      _PR,@-R15
          MOV.W       L11,R2      ; _x
          JSR         @R2
          NOP
          MOV.W       L11+2,R5    ; _y
          MOV.W       L11+4,R4    ; _z
          MOV.L       @R5,R1      ; y
          ADD         R1,R0
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R4      ; z
L11:
          .DATA.W     _x
          .DATA.W     _y
          .DATA.W     _z
```

Note:

- If you specify `abs16`, `abs20`, `abs28`, or `abs32` for a variable or function, use the `#pragma section` directive to switch the section so that the section is placed in an address area that can be represented by the specified bit addressing during linkage.
- The following table shows the `#pragma abs`*n* directives and the address ranges in which a section can be placed.

Table 1-2 Address ranges in which the section can be placed

#pragma	Address range	
	Beginning	End
#pragma abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
#pragma abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
#pragma abs28	0x00000000	0x07FFFF7F *
	0xF8000000	0xFFFFFFFF
#pragma abs32	0x00000000	0xFFFFFFFF

* Note that the address is 0x07FFFF7F.

- If you specify `abs16`, place the section in either of the areas shown in the following figure.

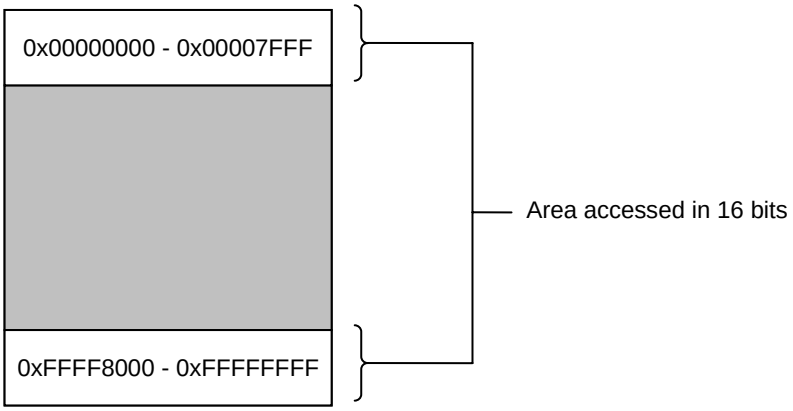


Figure 1-1

- If generation of position-independent code is specified (`pic=1` option) at compile time, function addresses are not generated for the specified bit value because addresses are referenced by relative access.

Source code:

```
#pragma abs16 (x,y,z)
extern int x(void);
int y;
long z;
void f(void)
{
    z = x() + y;
}
```

Expanded assembly code (pic=0 specified):

```
_f:
          STS.L       PR,@-R15
          MOV.W       L11,R2      ; _x
          JSR         @R2
          NOP
          MOV.W       L11+2,R5    ; _y
          MOV.W       L11+4,R4    ; _z
          MOV.L       @R5,R1      ; y
          ADD         R1,R0
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R4      ; z
L11:
          .DATA.W     _x
          .DATA.W     _y
          .DATA.W     _z
```

Expanded assembly code (pic=1 specified):

```
_f:
          STS.L       PR,@-R15
          MOV.L       L12+6,R3    ; H'FFFFFFFC+_x-L11
L11
          BSRF        R3
          NOP
          MOV.W       L12,R5      ; _y
          MOV.W       L12+2,R2    ; _z
          MOV.L       @R5,R1      ; y
          ADD         R1,R0
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R2      ; z
L12:
          .DATA.W     _y
          .DATA.W     _z
          .RES.W      1
          .DATA.L     H'FFFFFFFC+_x-L11
```

1.2 Performs inline expansion of functions

Inline expansion is a type of optimization that inserts the body of a function at the point at which the function is called. You can use inline expansion when you expect that it will reduce function call overhead, making the program smaller and allowing it to run faster. In particular, inline expansion can be very effective for functions that are called repeatedly in a loop. Since the compiler performs inline expansion before optimizing the source code, note that inline expansion for large functions increases program size, and might reduce the efficiency of compiler optimization. Inline expansion is more effective when it is performed for small functions that are called frequently.

Format:

#pragma inline [(]*function-name*[,...][)]

When you specify functions in a `#pragma inline` directive, make sure that the body of each function is defined after the directive.
The compiler also generates external definitions for the functions specified in a `#pragma inline` directive. If these external definitions are not necessary, specify `static` for the function declarations. Since the compiler does not generate the body of a static function when performing inline expansion, specifying `static` might reduce program size.
Whereas automatic inline expansion of the `inline` option specified for a function stops before the specified limit on increase in function size (expressed as a percentage) is reached, automatic inline expansion of the `#pragma inline` directive does not.
Note that, when `#pragma inline` is specified, inline expansion for a function does not occur if the function satisfies any of the following conditions:

– The function is defined before the `#pragma inline` directive.
– The function has variable parameters.
– The function is called via its address.

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
void main(void)
{
    int (*func_p)(int,int);

    func_p = func;

    x=func_p(10,20);
}
```

When the address of a function is assigned to a function-type pointer and the pointer is used to call the function, inline expansion does not occur.

Example: Source before and after inline expansion

Source code:

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
void main(void)
{
    x=func(10,20);
}
```

Source after expansion:

```
int x;
void main(void)
{
    int func_result;
    {
        int a_1=10, b_1=20;
        func_result=(a_1+b_1)/2;
    }
    x=func_result;
}
```

1.3 Expands an assembly-language description function

Inline expansion for assembly functions is effective when you want to use CPU instructions that C does not support or when you want to improve performance by coding the functions in assembly language rather than in C. In a C source file, you can code functions in assembly language by using the `#pragma inline_asm` directive to declare that the functions are written in assembly language. Such assembly functions are called *inline assembly functions*. In the `#pragma inline_asm` directive, you can also specify a `size=`*numeric-value* option to specify the size of an assembly function. Specifying this option might improve the efficiency of optimization.

Note that when you specify the size of a function in the `#pragma inline_asm` directive, you must make sure that the size is the same as or larger than the actual object size. If you specify a size smaller than the actual object size, operation of the compiler is not guaranteed.

Format:

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

Note the following when you code inline assembly functions:

- Make sure that each label is a local label (begins with a question mark (?) and consists of 16 or fewer characters).
- Do not code an instruction that automatically generates a literal pool. For details, see Example 2.
- Do not code an RTS (return) instruction at the end of a definition.
- Save and restore the contents of guaranteed registers.
  Save/restore operations must also be performed even for the registers specified in the `#pragma global_register` directive. Also note that the save/restore operations of the procedure register (PR) must be coded because the contents of the register are overwritten every time a function is called.

```
extern void sub(void);

#pragma inline_asm (func(size=0x14))
static void func(void)
{
    .IMPORT _sub
    STS.L PR,@-R15
    MOV.L ?LOCAL1,R0
    JSR @R0
    NOP
    LDS.L @R15+,PR
    BRA   ?LOCAL2
    NOP
    .ALIGN 4
?LOCAL1:
    .DATA.L _sub
?LOCAL2:
}

void g(void)
{
    func(void);
}
```

Save/restore operations must be coded when a function is called.

A C source file that includes assembly functions must be output in an assembly file format (`code=asmcode`).

Option settings in High-Performance Embedded Workshop (*Renesas IDE* hereafter):

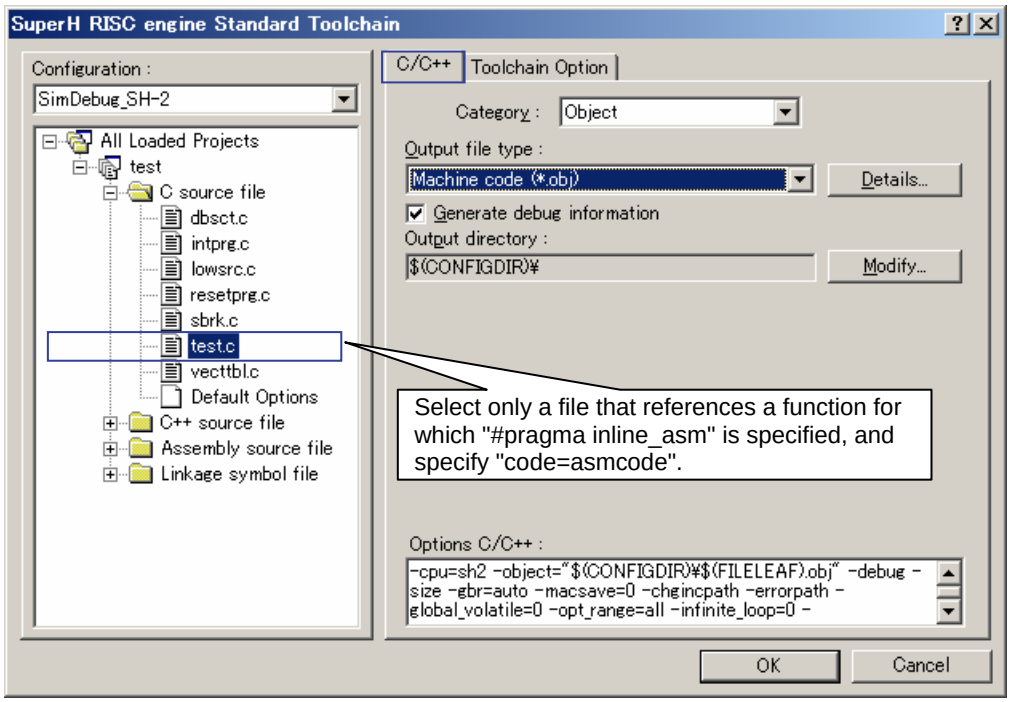


Figure 1-2

If a function specified in the `#pragma inline_asm` directive causes a compilation error when the output of compiler debug information is enabled, the C source program line information is output in Renesas IDE. You cannot use the line information to jump to the assembly program location that caused the error. If you disable output of the compiler debug information, the assembly program line information is displayed in Renesas IDE. Renesas recommends that you disable output of the compiler debug information when you debug a function specified in the `#pragma inline_asm` directive.

Note that the interface between functions must comply with the generation rules for the C or C++ compiler (see Table 1-3 and Table 1-4).

Table 1-3 General rules for assigning arguments in C

Assignment rules		
Arguments passed via registers		Arguments passed via stacks
Registers for storing arguments	Supported data types	
R4 - R7	`char`, `unsigned char`, `bool`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float` (when the CPU is other than SH-2E, SH2A-FPU, SH-4, or SH-4A), pointer, pointer to a data member, and references	• Arguments other than those passed via registers • Arguments that cannot be stored in registers R4 to R7 because other arguments have already been stored in these registers • Arguments that cannot be stored in registers FR4 (DR4) to FR11 (DR10) because other arguments have already been stored in these registers • Arguments of types `long long` and `unsigned long long` • Arguments of types `__fixed`, `long __fixed`, `__accum`, and `long __accum`
FR4 - FR11 #1	When the CPU is SH-2E: • `float` • `double` (with the `double=float` option specified) When the CPU is SH2A-FPU, SH-4, or SH-4A: • `float` (with the `fpu=double` option specified) • `double` (with the `fpu=single` option specified)	
DR4 - DR10 #2	When the CPU is SH2A-FPU, SH-4, SH-4A: • `double` (without the `fpu=single` option specified) • `float` (with the `fpu=double` option specified)	

Notes: #1: SH-2E, SH2A-FPU, SH-4, and SH-4A registers used for single-precision floating-point numbers
#2: SH2A-FPU, SH-4, and SH-4A registers used for double-precision floating-point numbers

Table 1-4 Return value types and setting location in C programs

Return value types	Setting location
(singed) char, unsigned char, (singed) short, unsigned short, (singed) int, unsigned int, long, unsigned long, pointers, bool, references, pointers to data members	R0: 32 bits The contents of the upper three bytes of (signed) char, or unsigned char and the contents of the upper two bytes of (signed) short or unsigned short are not guaranteed. However, when the rtnext option is specified, sign extension is performed for (signed) char or (signed) short type, and zero extension is performed for unsigned char or unsigned short type.
float	FR0: 32 bits (1) For CPU is SH-2E • Return value is float type. • Return value is double type and double=float is specified. (2) For SH2A-FPU,SH-4, or SH-4A • Return value is float type and fpu=double is not specified. • Return value is floating-point type and fpu=single is specified.
double and long double	Return value setting area (memory) For SH2A-FPU, SH-4, or SH-4A • Return value is double type and fpu=single is not specified. • Return value is floating-point type and fpu=double is specified.
Structure, union, and class types, and pointers to function members	Return value setting area (memory).
(signed) long long and unsigned long long	Return value setting area (memory).
__fixed, long __fixed, __accum, and long __accum	Return value setting area (memory).

Example 1: Specifying an inline assembly function

Source code:

```
/* Inline function definition */
/* FILE: inlasm.h */
#pragma inline_asm(rev4b(size=0x04))
static unsigned long rev4b(unsigned long p)
/* Function is declared as static */
{
  ; In a definition, comment lines begin with a semicolon, as in assembly language.
    SWAP.W R4,R0
    SWAP.B R0,R0
  ; Do not specify an RTS instruction at the end of the definition.
}
#pragma inline_asm(ovf)
static unsigned long ovf(void)
{
?LABEL001 ; In an inline assembly function, use local labels.
  ; Local label: String beginning with ? and consisting of 16 or fewer characters
    MOV R4,R0
      :
    CMP/EQ #1,R0
    BT ?LBABEL001
}
```

Example 2: Notes on automatic generation of a literal pool

Incorrect source code:

```
#pragma inline_asm(f)
/* Incorrect inline assembly function */
```

Correct source code 1:

```
#pragma inline_asm(f(size=0x0c))
/* Correct inline assembly function */
```

```
static unsigned long f(void)
{
    MOV.L #H'f0000000,R0

  ; This code causes the assembler to
  ; automatically generate a literal pool.
  ; As a result, the compiler-generated code
  ; might not be aligned correctly.
}
```

```
static unsigned long f(void)
{
    MOV.L ?LOCAL1,R0 ; Data is referenced from
                     ; the local label.
    BRA  ?LOCAL2 ; Jumps over data definitions.
    NOP
    .ALIGN 4
?LOCAL1:
    .DATA.L H'F0000000
?LOCAL2:
}
```

Correct source code 2:

```
#pragma inline_asm(f(size=0x06))
/* Correct inline assembly function */
static unsigned long f(void)
{
     MOV         #-16,R0    ; H'FFFFFFF0
     SHLL8       R0
     SHLL16      R0
; No data is referenced from labels.
}
```

Example 3: Optimization with the "size" option specified

Evaluation at branches is further optimized by specifying the size.

Source code ("size" not specified)

```
#include <machine.h>
extern int a;

#pragma inline_asm (func)
static int func(void)
{
   NOP
}

void g(void)
{
    if (a) {
        func();
    }
    if (a) {
        nop();
    }
}
```

Expanded assembly code:

```
_g:
          MOV.L       L16+2,R6    ; _a
          MOV.L       @R6,R2      ; a
          TST         R2,R2
          BF          L20
          MOV.L       L16+6,R3    ; L13
          JMP         @R3
          NOP
L20:
          BRA         L15
          NOP
L16:
          .RES.W      1
          .DATA.L     _a
          .DATA.L     L13
L15:
   NOP
          .ALIGN      4
          MOV.L       L18+2,R6    ; _a
          BRA         L17
          MOV.L       @R6,R2      ; a
L18:
          .RES.W      1
          .DATA.L     _a
L17:
          TST         R2,R2
          BT          L13
```

Source code ("size=0x20" specified)

```
#include <machine.h>
extern int a;

#pragma inline_asm (func(size=0x20))
static int func(void)
{
   NOP
}

void g(void)
{
    if (a) {
        func();
    }
    if (a) {
        nop();
    }
}
```

Expanded assembly code:

```
_g:
          MOV.L       L15+2,R6    ; _a
          MOV.L       @R6,R2      ; a
          TST         R2,R2
          BT          L13
   NOP
          .ALIGN      4
          MOV.L       L15+2,R6    ; _a
          MOV.L       @R6,R2      ; a
          TST         R2,R2
          BT          L13
          NOP
L13:
          RTS
          NOP
L15:
          .RES.W      1
          .DATA.L     _a
```

```
            NOP
L13:
            RTS
            NOP
```

1.4 Generates or does not generate save and restore code at the start and end of functions

Program execution speed and efficiency of ROM usage can be improved by deleting the register save operation at function entry points and the register restore operation at function exit points. You can use the `#pragma noregsave`, `#pragma noregalloc`, and `#pragma regsave` directives for fine-grained control of the saving and restoring of the guaranteed registers listed in Table 1-5.

Table 1-5 Guaranteed registers that can be controlled by using "#pragma noregsave", "#pragma noregalloc", and "#pragma regsave"

Register	Explanation
R8 - R14	--
FR12 - FR15	SH-2E, SH2A-FPU, SH-4, and SH-4A registers used for single-precision floating-point numbers
DR12 and DR14	SH2A-FPU, SH-4, and SH-4A registers used for double-precision floating-point numbers

Specifying `#pragma noregsave` for frequently executed functions can reduce program size and improve execution speed.

(1) The `#pragma noregsave` directive specifies that guaranteed registers are not saved and restored at the entry and exit points of functions.
(2) The `#pragma noregalloc` directive is used to create an object that does not save/restore guaranteed registers at function entry/exit points, and does not allocate guaranteed registers across function calls.
(3) The `#pragma regsave` directive is used to create an object which saves and restores guaranteed-registers at function entry/exit points, and does not allocate guaranteed registers.
(4) `#pragma regsave` and `#pragma noregalloc` can be specified simultaneously for the same function. Such overlapping specifications causes an object to be created in which all guaranteed-registers are saved and restored at the function entry/exit points, and no guaranteed registers are allocated across function calls.

Table 1-6 Operation of "#pragma regsave", "#pragma noregsave", and "#pragma noregalloc"

#pragma	Register save/restore operations	Register use
#pragma noregsave	Guaranteed registers are not saved and restored.	Guaranteed registers are used.
#pragma noregalloc	Guaranteed registers are not saved and restored.	Guaranteed registers are not used across multiple function calls.
#pragma regsave	Guaranteed registers are saved and restored.	Guaranteed registers are not used across multiple function calls. The frequency of using guaranteed registers within one function is low.
#pragma regsave + #pragma noregalloc	Guaranteed registers are saved and restored.	Guaranteed registers are not used across multiple function calls. The frequency of using guaranteed registers within one function is high.

A function for which `#pragma noregsave` is specified might not operate correctly if it is called from ordinary functions. Make sure that a function for which `#pragma` is specified is called from one of the following types of functions:

- Function for which `#pragma regsave` is specified
- Function for which `#pragma noregalloc` is specified and that is called from a function for which `#pragma regsave` is specified

Before allowing a function for which the `#pragma noregsave`, `#pragma noregalloc`, or `#pragma regsave` directive is specified to be called, make sure that the directive is specified for all instances of the function throughout the project. Renesas recommends that you specify the directive in a common header file.

Example 1:

For a function that does not return to the caller, the contents of registers do not need to be saved and restored. You can therefore reduce object size and improve execution speed by specifying #pragma noregsave for such a function.

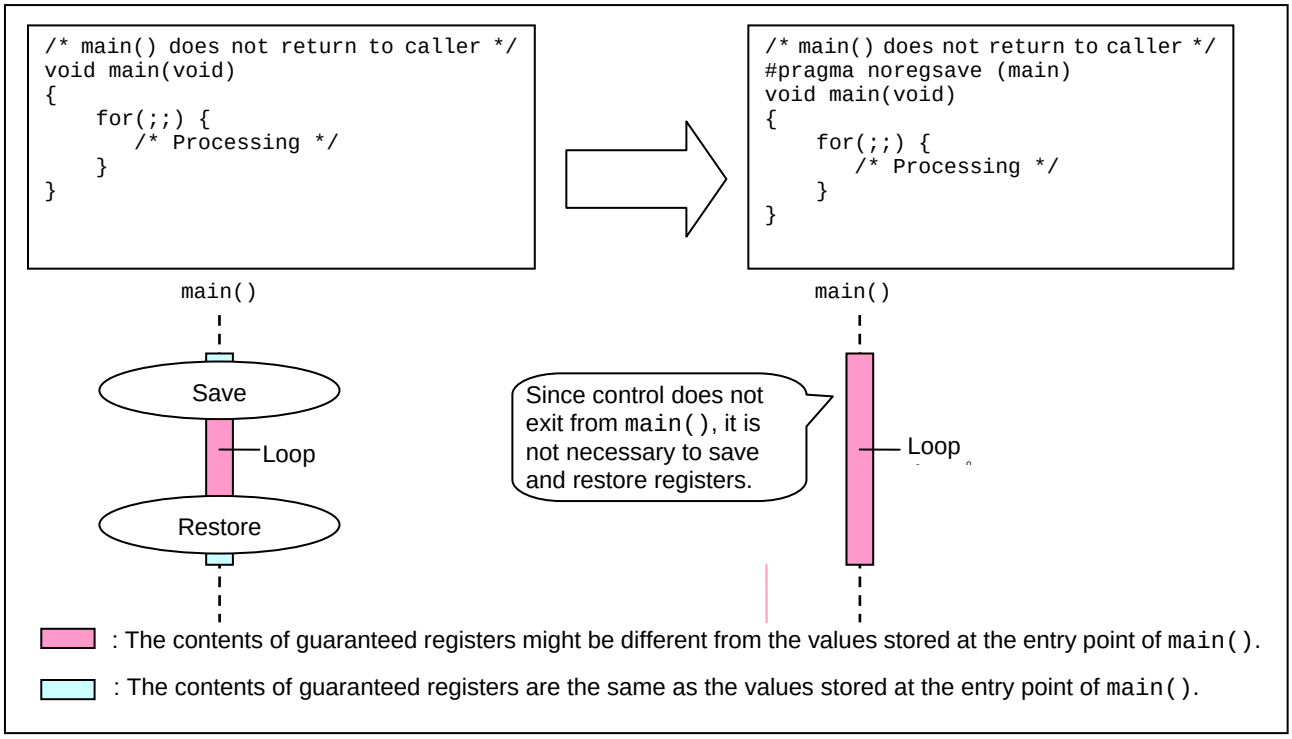


Figure 1-3

Example 2:

If functions b1() and b2() use guaranteed registers, the contents of these registers are saved at the entry points of the functions and restored at the exit points of the functions. For example, if function a() frequently calls functions b1() and b2(), the guaranteed registers are also saved and restored frequently, lowering efficiency.

If function a() does not use guaranteed registers, functions b1() and b2(), the guaranteed registers need not be saved and restored. If #pragma regsave is specified for function a(), the contents of guaranteed registers are saved at the entry point and restored at the exit point of the function, but the guaranteed registers are not used across multiple function calls. In addition, if #pragma noregsave is specified for functions b1() and b2(), the guaranteed registers are never saved and restored for these functions. When #pragma directives are specified in this way, guaranteed registers are saved and restored only for function a().

You can optimize the location of register save and restore operations as described above to decrease their frequency.

Specifying #pragma noregsave together with #pragma regsave results in effective optimization because the frequency of using guaranteed registers within a function increases. If you specify #pragma regsave, you should also specify #pragma noregsave.

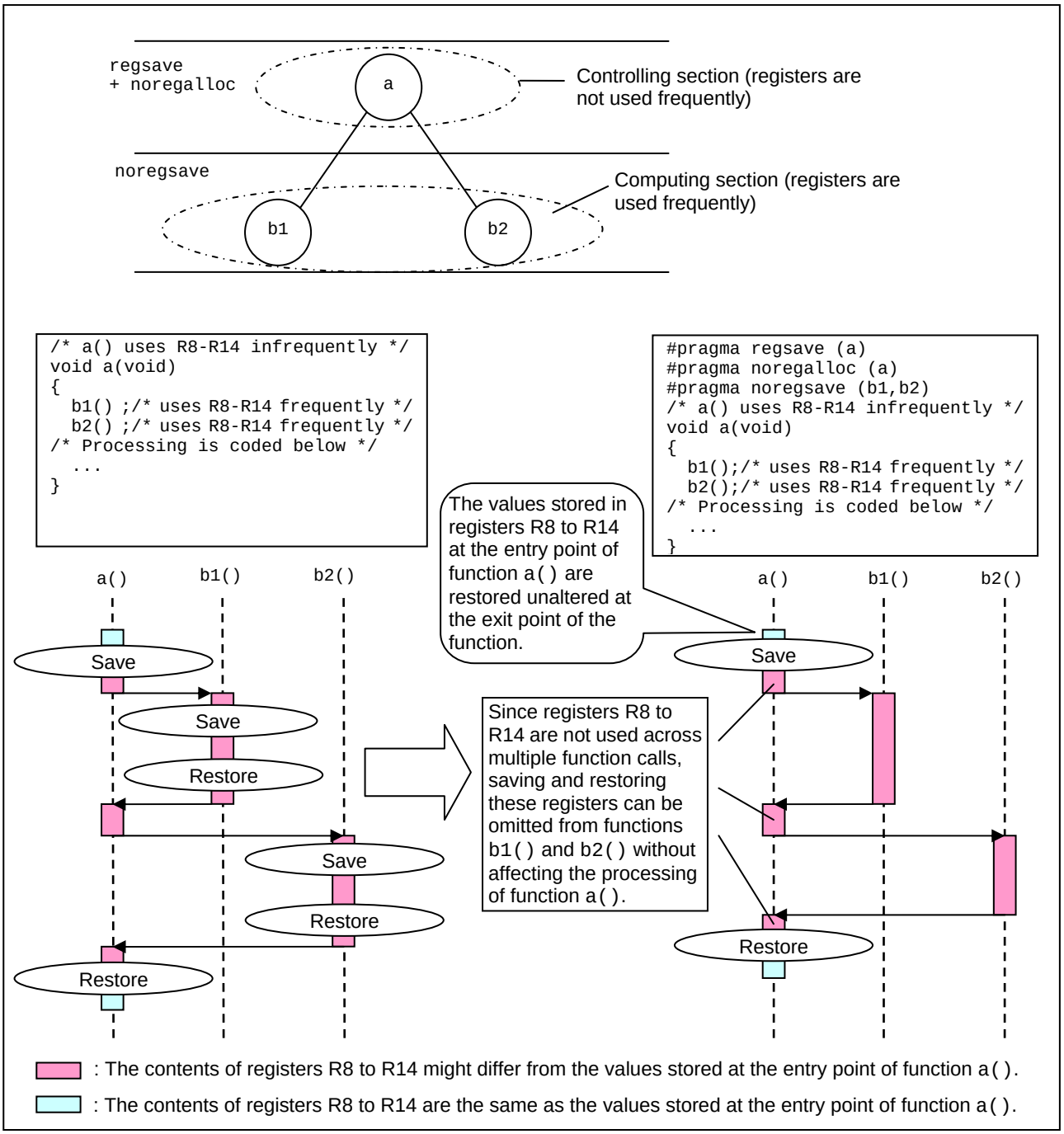

regsave
+ noregalloc
a
Controlling section (registers are not used frequently)
noregsave
b1
b2
Computing section (registers are used frequently)
/* a() uses R8-R14 infrequently */
void a(void)
{
b1() ;/* uses R8-R14 frequently */
b2() ;/* uses R8-R14 frequently */
/* Processing is coded below */
...
}
#pragma regsave (a)
#pragma noregalloc (a)
#pragma noregsave (b1,b2)
/* a() uses R8-R14 infrequently */
void a(void)
{
b1();/* uses R8-R14 frequently */
b2();/* uses R8-R14 frequently */
/* Processing is coded below */
...
}
The values stored in registers R8 to R14 at the entry point of function a() are restored unaltered at the exit point of the function.
a()
b1()
b2()
Save
Save
Restore
Save
Restore
Restore
Since registers R8 to R14 are not used across multiple function calls, saving and restoring these registers can be omitted from functions b1() and b2() without affecting the processing of function a().
: The contents of registers R8 to R14 might differ from the values stored at the entry point of function a().
: The contents of registers R8 to R14 are the same as the values stored at the entry point of function a().

Figure 1-4

Example 3:

When functions `b1()` and `b2()` use guaranteed registers, the contents of these registers are saved at the entry points of the functions and restored at the exit points of the functions. For example, when functions `b1()` and `b2()` are also frequently called from function `c2()`, which is called from function `a()`, the guaranteed registers are also saved and restored frequently, lowering the efficiency of program execution.
If functions `a()` and `c2()` do not use the same guaranteed registers, functions `b()` and `b2()` do not require that these registers be saved and restored. If you specify function `a()` in `#pragma regsave` and `#pragma noregalloc` directives, the contents of the guaranteed registers are saved at the entry point and restored at the exit point of function `a()`, and the guaranteed registers used by function `a()` are not used by other functions. If you specify function c2() in the `#pragma noregalloc` directive, the guaranteed registers used by function c2() are not used by other functions. If you specify functions `b1()` and `b2()` in the `#pragma regsave` directive, you can suppress saving and restoring of the guaranteed registers used by these functions.
When the directive settings are specified as described above, guaranteed registers are saved and restored for function `a()`, but are not saved and restored for functions `b1()` and `b2()`. Accordingly, you can reduce the frequency of register save and restore operations by changing the register save and restore locations.

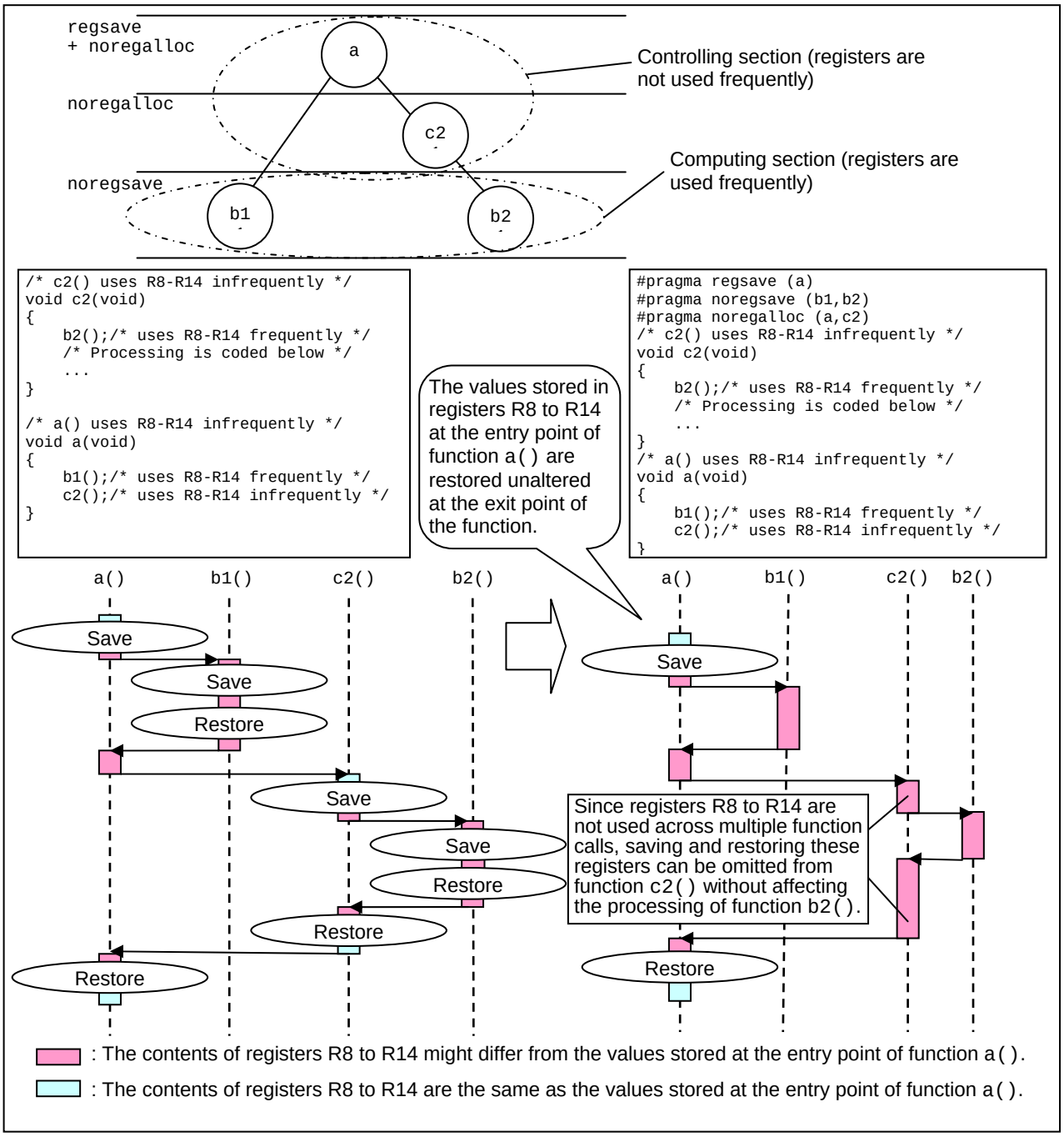

regsave
+ noregalloc
noregalloc
noregsave
a
c2
b1
b2
Controlling section (registers are not used frequently)
Computing section (registers are used frequently)
/* c2() uses R8-R14 infrequently */
void c2(void)
{
b2();/* uses R8-R14 frequently */
/* Processing is coded below */
...
}
/* a() uses R8-R14 infrequently */
void a(void)
{
b1();/* uses R8-R14 frequently */
c2();/* uses R8-R14 infrequently */
}
#pragma regsave (a)
#pragma noregsave (b1,b2)
#pragma noregalloc (a,c2)
/* c2() uses R8-R14 infrequently */
void c2(void)
{
b2();/* uses R8-R14 frequently */
/* Processing is coded below */
...
}
/* a() uses R8-R14 infrequently */
void a(void)
{
b1();/* uses R8-R14 frequently */
c2();/* uses R8-R14 infrequently */
}
The values stored in registers R8 to R14 at the entry point of function a() are restored unaltered at the exit point of the function.
a()
b1()
c2()
b2()
Save
Save
Restore
Save
Save
Restore
Restore
Restore
Since registers R8 to R14 are not used across multiple function calls, saving and restoring these registers can be omitted from function c2() without affecting the processing of function b2().
: The contents of registers R8 to R14 might differ from the values stored at the entry point of function a().
: The contents of registers R8 to R14 are the same as the values stored at the entry point of function a().

Figure 1-5

1.5 Allocates global variables to registers

You can use the `#pragma global_register` directive to allocate global variables to registers. By allocating global variables to registers, you can reduce the number of load and store instructions.

You must specify the `#pragma global_register` directive in all files. Use the `preinclude` option to specify a header file that contains the `#pragma global_register` declaration. This option specifies the directive in all files. For details about how to specify the option settings in Renesas IDE, see Figure 1-6.

For the standard library as well, you must specify the `#pragma global_register` directive. Use the `preinclude` option to specify a header file that contains the `#pragma global_register` declaration. For details about how to specify the option settings in Renesas IDE, see Figure 1-7.

Format:

#pragma global_register [(]*variable-name*=*register-name*[,...][)]

- For *variable-name*, you can specify a global variable of the integer, float, or pointer type. If the CPU is not SH2A-FPU, SH-4, or SH-4A, you can specify a global variable of type double when the `double=float` option is specified.
- For *register-name*, you can specify R8 to R14, FR12 to FR15 (when the CPU is SH-2E, SH2A-FPU, SH-4, or SH-4A), DR12, or DR14 (when the CPU is SH2A-FPU, SH-4, or SH-4A).
  - Types of variables that can be allocated to registers FR12 to FR15
    (i) SH-2E
    - `float` type
    - `double` type (when the `double=float` option is specified)

    (ii) SH2A-FPU, SH-4, and SH-4A
    - `float` type (when the `fpu=double` option is not specified)
    - `double` type (when the `fpu=single` option is not specified)
  - Types of variables that can be allocated to registers DR12 to DR14
    (i) SH2A-FPU, SH-4, and SH-4A
    - `float` type (when the `fpu=double` option is specified)
    - `double` type (when the `fpu=single` option is not specified)
- You can neither set initial values for the variables nor use the variables for address reference.
- If the `#pragma global_register` directive settings are the same for all files and the library, correct operation cannot be guaranteed.
- You can specify only static data members. You cannot specify non-static data members.

Option settings in Renesas IDE:

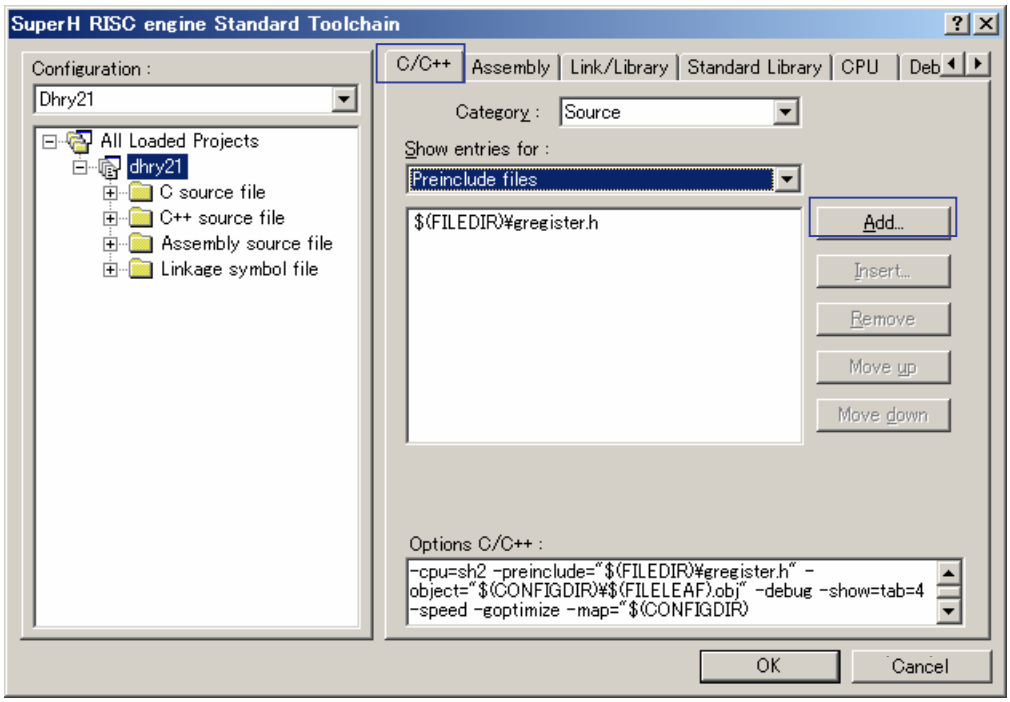


Figure 1-6

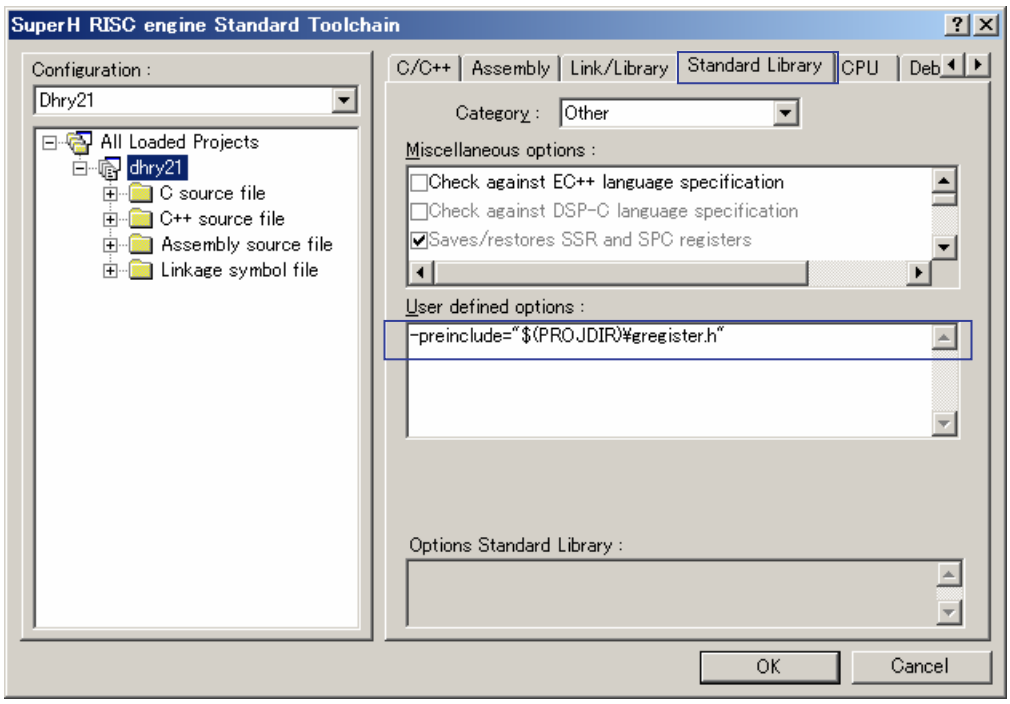


Figure 1-7

Example:

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

int x;
char *y;

void func1(void)
{
    x++;
}

void func2(void)
{
    *y=0;
}

void func(int a)
{
    x = a;
    func1();
    func2();
}
Expanded assembly code:

_func1:
          RTS
          ADD         #1,R13
_func2:
          MOV         #0,R2       ; H'00000000
          RTS
          MOV.B       R2,@R14     ; *(y)
_func:
          STS.L       PR,@-R15
          BSR         _func1
          MOV         R4,R13
          BRA         _func2
          LDS.L       @R15+,PR
```

1.6 Specifies GBR base variables

A GBR base variable is a variable for which a GBR base is specified and which can be accessed GBR relative access code. The use of a relative address means that the variable address does not have to be loaded, resulting in a smaller object.

Note, however, that the number and size of variables that can be used as GBR base variables are limited. Specify as GBR base variables only those variables that will be used frequently.

Format:

#pragma gbr_base (*variable-name* [,*variable-name*...])
#pragma gbr_base1 (*variable-name* [,*variable-name*...])

The `#pragma gbr_base` directive allocates the specified variables in the $G0 section, which occupies relative byte positions 0 to 127 from the address indicated by GBR.

The `#pragma gbr_base1` directive allocates the specified variables in the $G1 section, which occupies relative byte positions 128 to 1020 from the address indicated by GBR. However, in this section, the maximum byte position at which a variable can be allocated depends on the data type. For a variable of type `char` or `unsigned char`, the maximum byte position is 255. For a variable of type `unsigned short`, the maximum byte position is 510. For a variable of type `int`, `unsigned int`, `long`, `unsigned long`, `float`, or `double`, the maximum byte position is 1020.

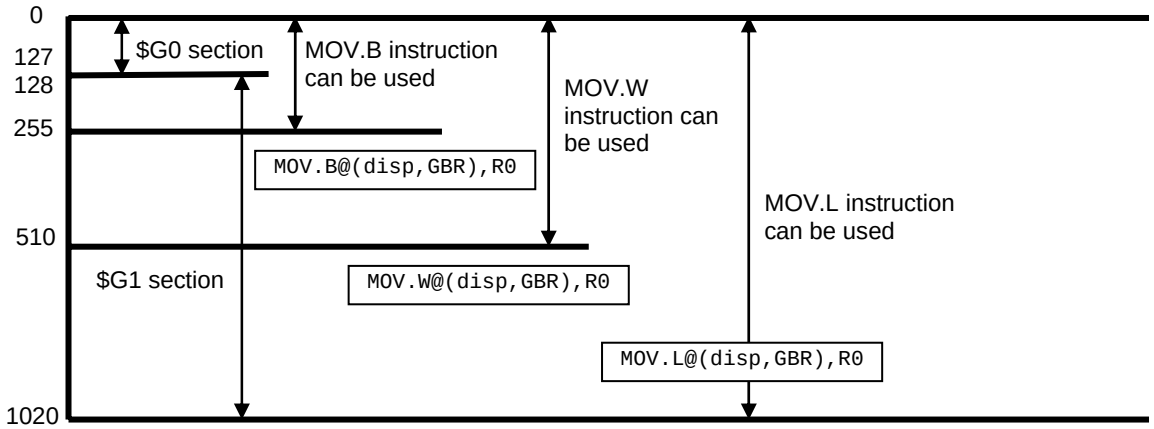


Figure 1-8

Variables that are frequently accessed or used for bitwise operations should be allocated in the $G0 section if at all possible. An object that accesses data in the $G0 section is faster and smaller than an object that accesses data in the $G1 section (see Example 2 below).

The `#pragma gbr_base` and `#pragma gbr_base1` directives allocate the specified variables in the appropriate sections in the order in which the variables are declared. Since alternately declaring variables of different sizes wastes space, as much as possible, try to declare variables of the same size together.

If GBR base variables overflow the appropriate area, the following error occurs during linkage:

```
L2330 (E)Relocation size overflow
```

If this error occurs, delete specification of the relevant variables from the `#pragma gbr_base` or `#pragma gbr_base1` directive.

To use the `#pragma gbr_base` directive, use a linkage editor to set the $G0 section. To use the `#pragma gbr_base1` directive, use a linkage editor to set the $G0 and $G1 sections. When you set the $G1 section, make sure that the address of the section is the start address of the $G0 section + 0x80.

Note that the same specification of the `#pragma gbr_base` and `#pragma gbr_base1` directives should be used throughout the project. The `preinclude` option is useful for ensuring the same specification throughout the project.

The variables specified in the `#pragma gbr_base` or `#pragma gbr_base1` directive are allocated in either the $G0 or $G1 section whether or not an initial value is specified. When the compiler generates an object, the compiler treats the $G0 and $G1 sections as initialized data. The initialized data (variables) has initial values. Although the initial values

must be prepared in the ROM area, the data must be stored in the RAM area because the data might change during program execution.

Therefore, after an initial value is set in the ROM area, the initial value must be copied from the ROM area to the RAM area. To set an initial value in the ROM area and to access data with an address in the RAM area, you must use a linkage editor to specify the appropriate ROM support option. For the $G0 and $G1 sections, you must also add processing that copies the initial data from the ROM area to the RAM area, and use a linkage editor to specify the appropriate ROM support option. For details, see *4.Memory Initialization* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Introduction guide] Sample file Guide for SH-1, SH-2, and SH-2A*.

Make sure that the address of the RAM area for the $G1 section is the address of the RAM area for the $G0 section + 0x80.

Before you can use GBR base variables, you must set the start address of the RAM area for copying the $G0 section in the GBR register. You can use intrinsic function `set_gbr()` to perform this operation.

The following is an example of code that copies the $G0 section in the ROM area to the $G1 section in the RAM area.

```
Initialization program:

#include <machine.h>

/* Functions executed before main */
void PowerON_Reset(void)
{
        ...
set_gbr(__sectop("$RG0")); /* Sets the start address of $RG0 section in the GBR register */
        ...
}
```

The following explains the tasks required before you can use the GBR base.

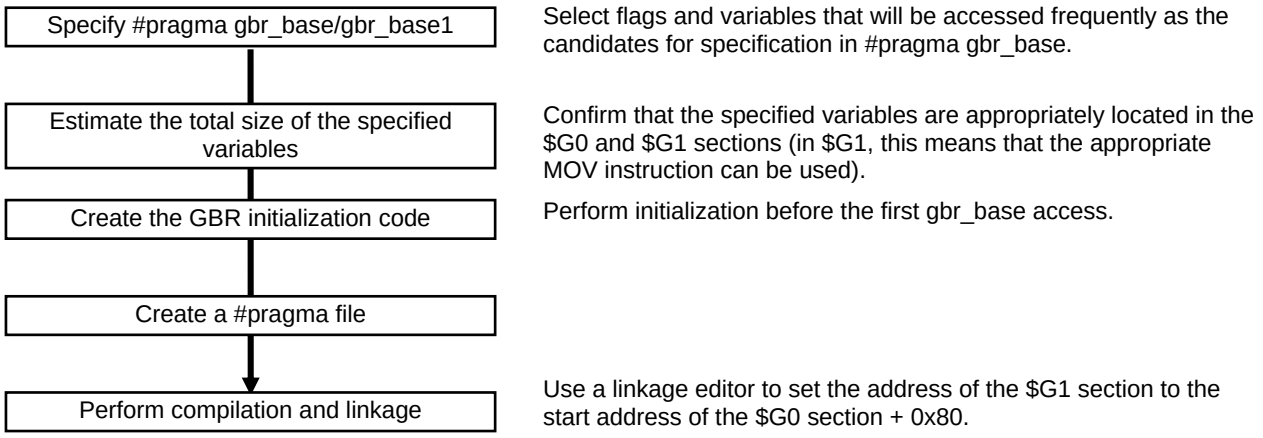


Figure 1-9

If you specify the `#pragma gbr_base` or `#pragma gbr_base1` directive, specify the `gbr=user` option (Figure 1-10). If you do not specify the option, a warning is output for the directive and specification of the directive is ignored.

Note that if you select **User(gnenrate logic operation)** from the **GBR relative operation** drop-down list, the `gbr=user` and `logic_gbr` options are specified. If these options are specified, GBR relative logical operation instructions might be used for operations that do not use GBR base variables.

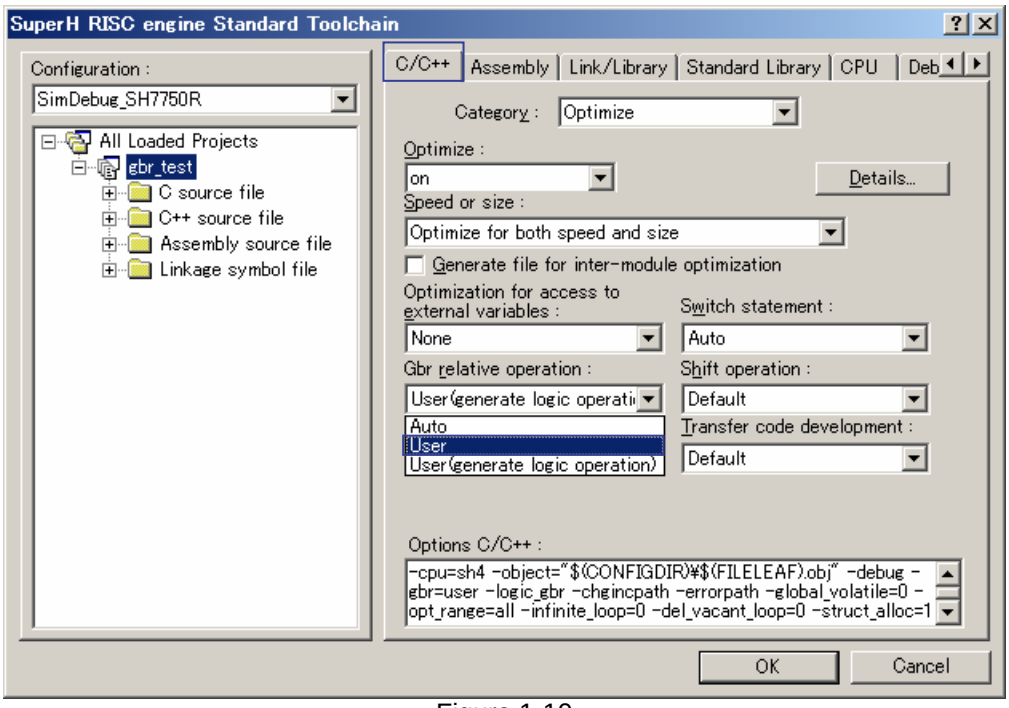


Figure 1-10

Example 1:

The following is an example of code generated when the `gbr=user` compiler option is specified.

Source code with #pragma gbr_base not specified:

```
struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

struct {
    char  c;
    short s;
    long  l;
} x, y;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;

    if (bitf.c) {
        bitf.d = 1;
    } else {
        bitf.e = 1;
    }

    x.c = y.c;
    x.s = y.s;
    x.l = y.l;
}
```

Source code with #pragma gbr_base specified:

```
#pragma gbr_base  (bitf)
#pragma gbr_base1 (x,y)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

struct {
    char  c;
    short s;
    long  l;
} x, y;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;

    if (bitf.c) {
        bitf.d = 1;
    } else {
        bitf.e = 1;
    }

    x.c = y.c;
    x.s = y.s;
    x.l = y.l;
}
```

```
Expanded assembly code:
_f:
    MOV.L       L14,R5      ; _bitf
    MOV.B       @R5,R0      ; (part of)bitf
    AND         #191,R0
    OR          #128,R0
    TST         #32,R0
    BT/S        L12
    MOV.B       R0,@R5      ; (part of)bitf
    BRA         L13
    OR          #16,R0
L12:
    OR          #8,R0
L13:
    MOV.L       L14+4,R7    ; _y
    MOV.B       R0,@R5      ; (part of)bitf
    MOV.L       L14+8,R6    ; _x
    MOV.B       @R7,R1      ; y.c
    MOV.W       @(2,R7),R0  ; y.s
    MOV.L       @(4,R7),R4  ; y.l
    MOV.B       R1,@R6      ; x.c
    MOV.W       R0,@(2,R6)  ; x.s
    RTS
    MOV.L       R4,@(4,R6)  ; x.l
L14:
    .DATA.L     _bitf
    .DATA.L     _y
    .DATA.L     _x
    .SECTION    B,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .RES.B      1
    .RES.B      1
    .RES.W      1
_x:                         ; static: x
    .RES.L      2
_y:                         ; static: y
    .RES.L      2
```

```
Expanded assembly code:
_f:
    MOV         #_bitf-(STARTOF $G0),R0
    AND.B       #191,@(R0,GBR); (part of)bitf
    OR.B        #128,@(R0,GBR); (part of)bitf
    MOV.B       @(_bitf-(STARTOF $G0),GBR),R0
                                ; (part of)bitf
    TST         #32,R0
    BT          L12
    BRA         L13
    OR          #16,R0
L12:
    OR          #8,R0
L13:
    MOV.B       R0,@(_bitf-(STARTOF $G0),GBR)
                                ; (part of)bitf
    MOV.B       @(_y-(STARTOF $G0),GBR),R0; y.c
    MOV.B       R0,@(_x-(STARTOF $G0),GBR); x.c
    MOV.W       @(_y-(STARTOF $G0)+2,GBR),R0; y.s
    MOV.W       R0,@(_x-(STARTOF $G0)+2,GBR); x.s
    MOV.L       @(_y-(STARTOF $G0)+4,GBR),R0; y.l
    RTS
    MOV.L       R0,@(_x-(STARTOF $G0)+4,GBR); x.l
    .SECTION    $G0,DATA,ALIGN=4
_bitf:          ; static: bitf
    .DATAB.B    1,0
    .SECTION    $G1,DATA,ALIGN=4
_x:             ; static: x
    .DATAB.L    2,0
_y:             ; static: y
    .DATAB.L    2,0
```

Example 2 (comparison of gbr_base and gbr_base1):

The following is an example of code generated when the `gbr=user` compiler option is specified. If `#pragma gbr_base` is specified to allocate variables in the $G0 section, literal access does not occur.

```
Source code with #pragma gbr_base1 specified:
#pragma gbr_base1 (bitf)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;
}
Expanded assembly code:

_f:
    MOV.W       L11,R0 ; _bitf-(STARTOF $G0)
    AND.B       #191,@(R0,GBR);(part of)bitf
    RTS
    OR.B        #128,@(R0,GBR);(part of)bitf
L11:
    .DATA.W     _bitf-(STARTOF $G0)
    .SECTION    $G1,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .DATAB.B    1,0
```

```
Source code with #pragma gbr_base specified:
#pragma gbr_base  (bitf)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;
}
Expanded assembly code:

_f:
    MOV         #_bitf-(STARTOF $G0),R0
    AND.B       #191,@(R0,GBR); (part of)bitf
    RTS
    OR.B        #128,@(R0,GBR); (part of)bitf
    .SECTION    $G0,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .DATAB.B    1,0
```

2. Other Useful Extended #pragma Directives

This chapter explains extended `#pragma` directives that provide benefits other than an improvement in performance. Table 2-1 lists the extended `#pragma` directives explained in this chapter.

Table 2-1 Other useful #pragma directives

No.	#pragma	Explanation	See section
1	#pragma section	Switches sections	2.1
2	#pragma bit_order	Switches the order of bit fields	2.2
3	#pragma pack #pragma unpack	Specifies the boundary alignment value for structures, unions, and classes	2.3

2.1 Switches sections

You can use the `#pragma section` directive to replace section names output by the compiler.

For example, you might want to allocate modules in separate sections, such as internal and external RAM. In this case, assign names to these sections, and use a linkage editor to specify the addresses in the sections at which you want to allocate the modules.

If you specify the `#pragma section` directive without specifying a section name, the default section name is used.

Format:

#pragma section [{*name*|*numeric-value*}]

Table 2-2 lists the default section names and the format of the new names created by the `#pragma section` directive.

Table 2-2 Default section names and format of replacement names

	Section	Specification	Default name	New name
1	Program section	#pragma section *XX*	P	P*XX*
2	Const section		C	C*XX*
3	Data section		D	D*XX*
4	Uninitialized data section		B	B*XX*

Instead of the `#pragma section` directive, you can also use the `section` option to change the default section names.

Format of the `section` option:

```
SEction = <sub>[,...]
   <sub>:  {   Program  =  section-name   |
               Const    =  section-name   |
               Data     =  section-name   |
               Bss      =  section-name   }
```

Option settings in Renesas IDE:

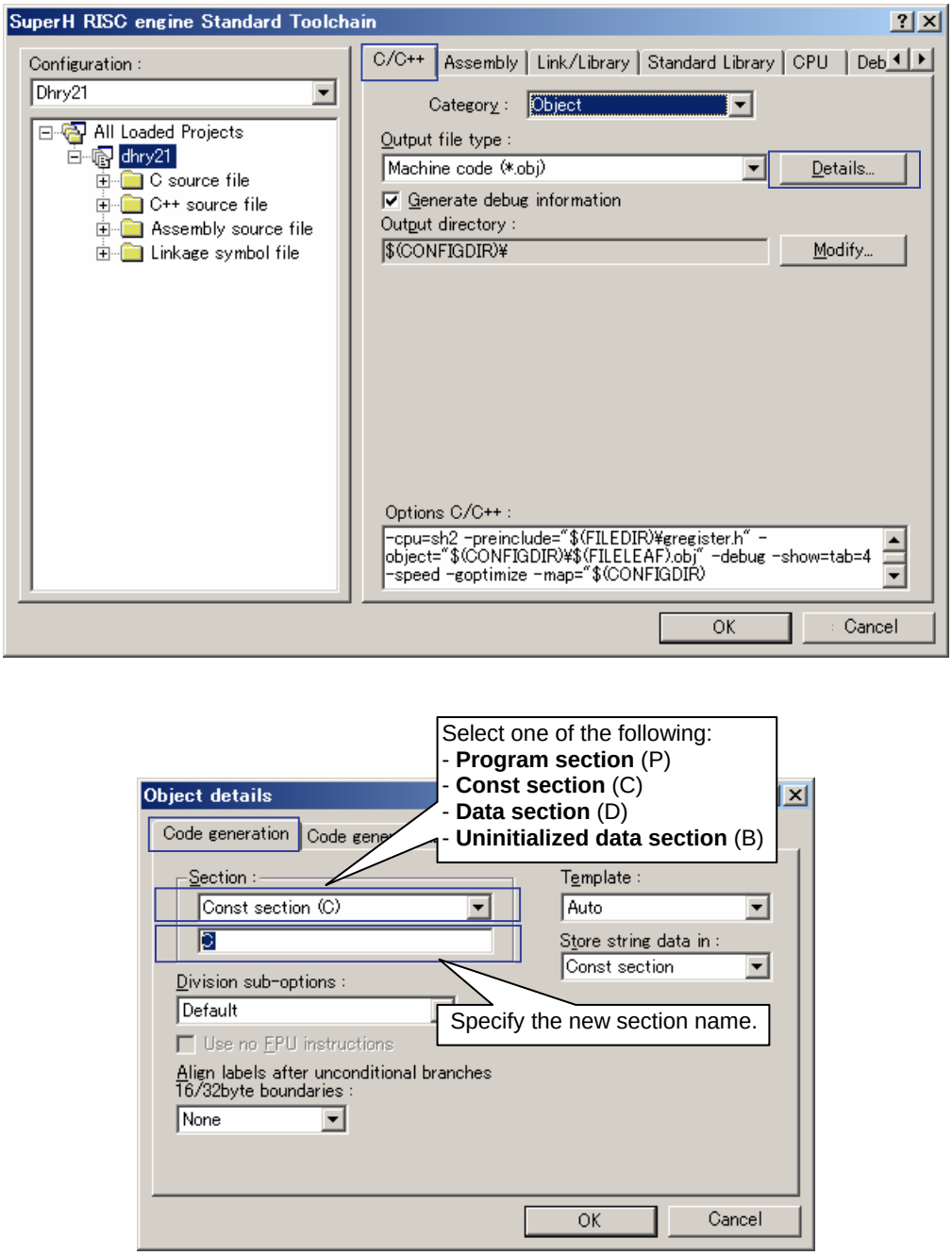


Figure 2-1

Example 1:

Source code:

```
#pragma section abc

int a;         /* a is allocated in section Babc */
const int c=1; /* c is allocated in section Cabc */
void f(void)   /* f is allocated in section Pabc */
{
    a=c;
}

#pragma section

int b;        /* b is allocated in section B */
void g(void)  /* g is allocated in section P */
{
    b=c;
}
```

Example 2:

When `section=program=PX,const=CX,bss=BX` is specified:

Source code:

```
#pragma section abc

int a;         /* a is allocated in section BXabc */
const int c=1; /* c is allocated in section CXabc */
void f(void)   /* f is allocated in section PXabc */
{
    a=c;
}

#pragma section

int b;        /* b is allocated in section BX */
void g(void)  /* g is allocated in section PX */
{
    b=c;
}
```

2.2 Switches the order of bit fields

You can use the `#pragma bit_order` directive to change the order of bit fields. Since the bit field allocation rule might differ depending on the microcomputer, you can use this functionality to improve portability of programs between different microcomputers.

Instead of using the `#pragma bit_order` directive, you can use an option to change the order of bit fields. If you specify both the `#pragma bit_order` directive and its equivalent option, the directive takes precedence.

Format:

#pragma bit_order [{left|right}]

When `left` is specified, bit field members are assigned from the upper-bit side. When `right` is specified, members are assigned from the lower-bit side. The default is `left`. However, if neither `left` nor `right` is specified in the directive when the option is specified, the option takes effect in the subsequent lines.

Example 1:

This example illustrates how the `#pragma bit_order left` and `#pragma bit_order right` directives allocate members.

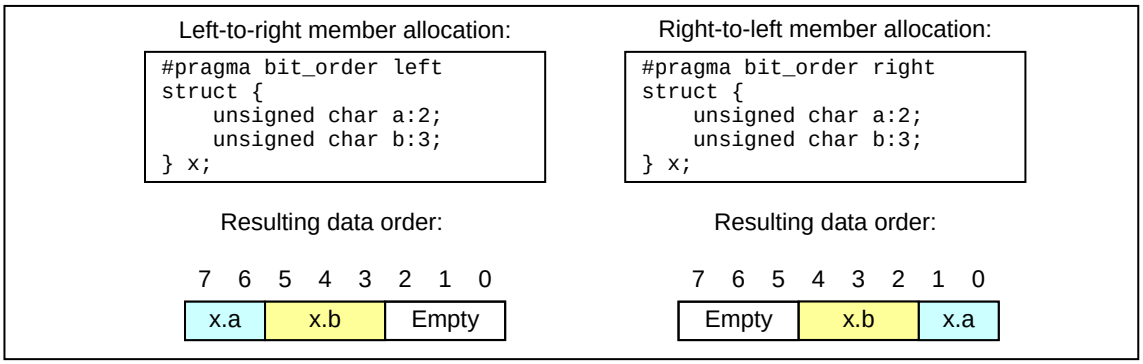


Figure 2-2

Example 2:

If type specifiers of the same size are specified in succession, members are allocated in the same area to the extent possible.

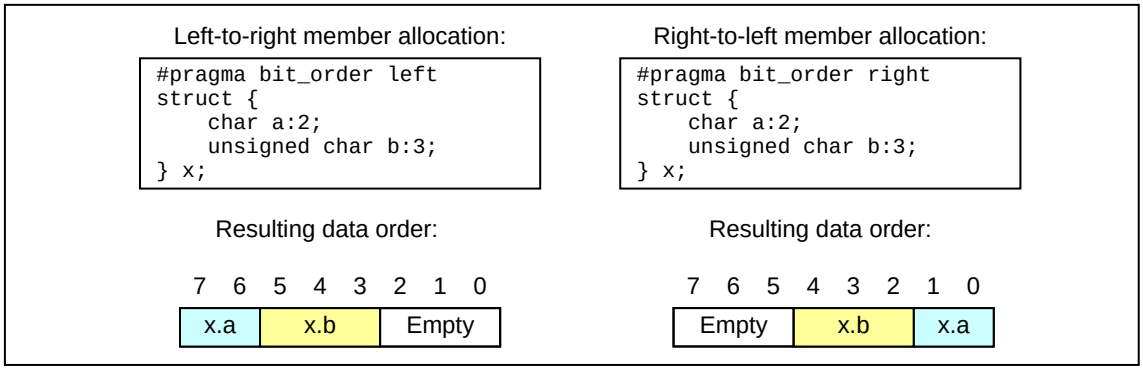


Figure 2-3

Example 3:

If type specifiers of different sizes are specified in succession, members are allocated in different areas.

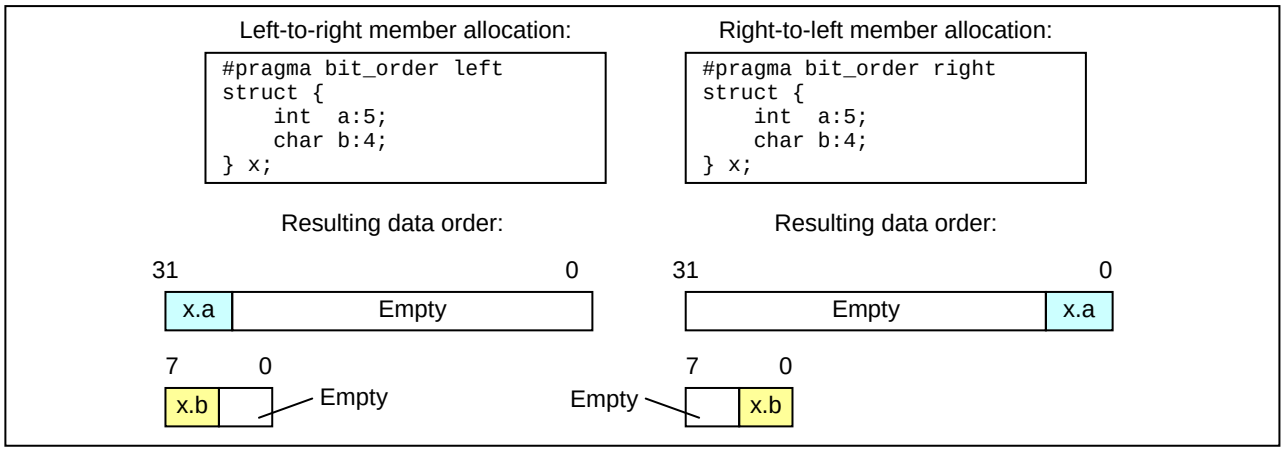


Figure 2-4

Example 4:

Even when type specifiers of the same size are specified in succession, if the remaining area becomes smaller than the next bit field member, the remaining area is not used and the member is allocated in the next area.

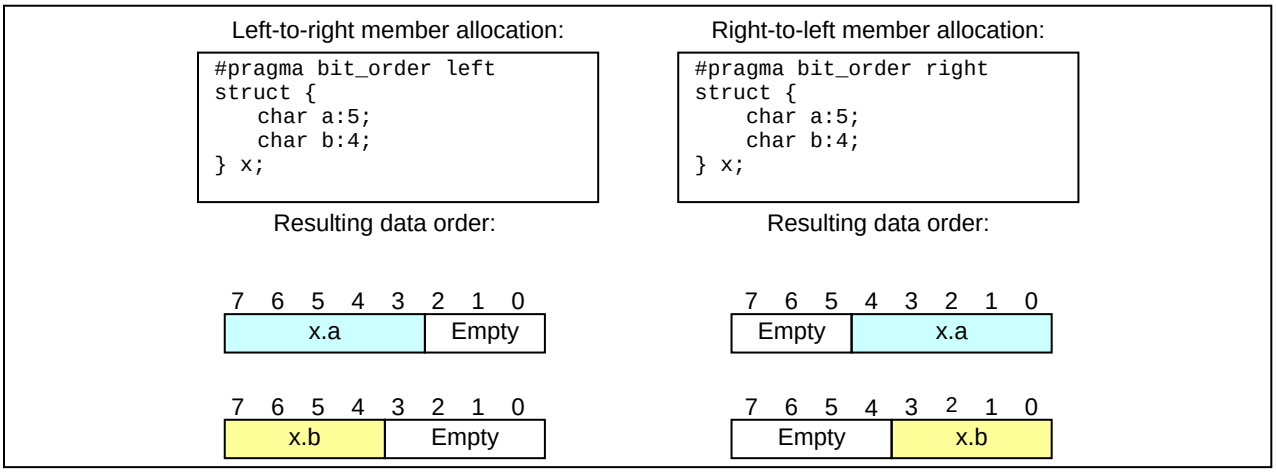


Figure 2-5

Example 5:

If a bit field member with a bit width of 0 appears, the subsequent members are forcibly allocated in the next area.

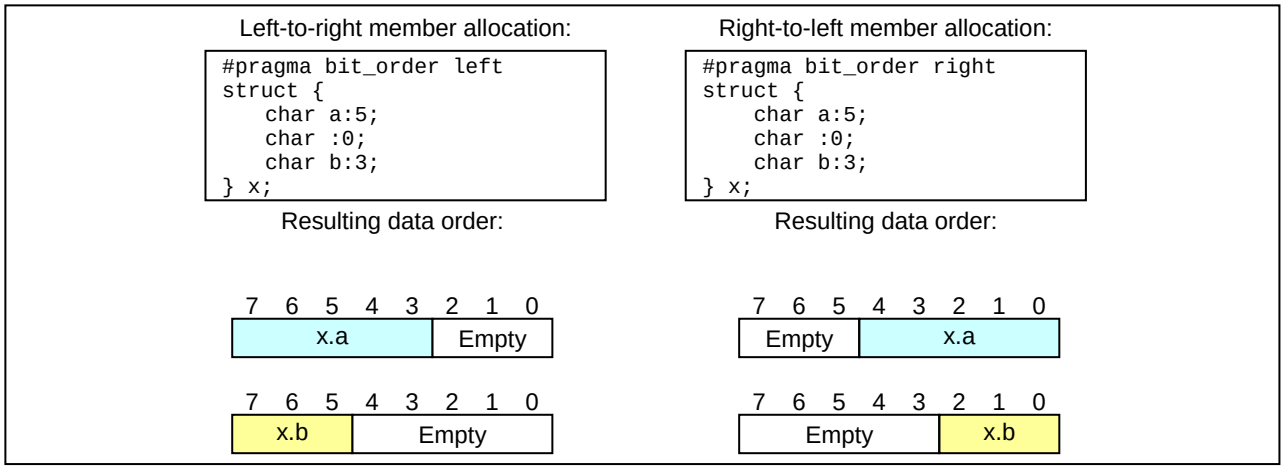


Figure 2-6

Example 6:

The default byte order is big endian. Some CPUs provide an option for changing the endian byte order. If little endian is specified (`endian=little`), bit field members in each area are allocated in reverse order of the big-endian order.

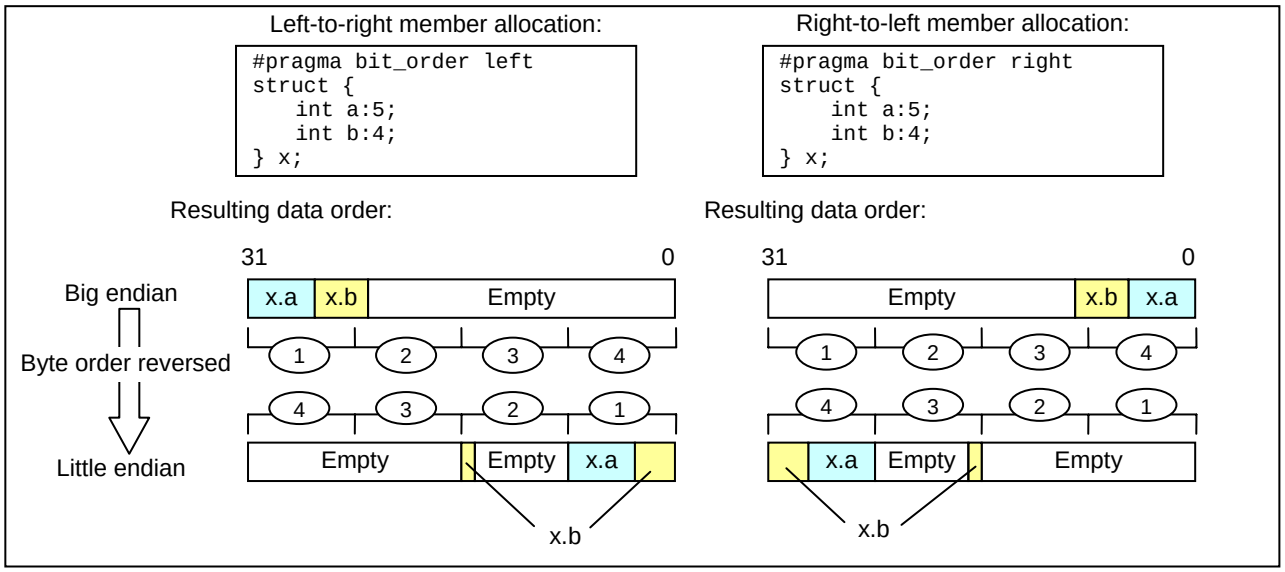


Figure 2-7

Note:

Care is required for the structure for I/O registers in the `iodefine.h` file created in Renesas IDE. If you use the `#pragma bit_order` directive or its equivalent option to specify that allocation of bit field members is to start with the lowest-order bit, the structure members will not indicate correct addresses.

Specify `#pragma bit_order left` at the beginning of the `iodefine.h` file and `#pragma bit_order` at the end of the file.

```
/* iodefine.h */
#pragma bit_order left

    ...

#pragma bit_order
```

2.3 Specifies the boundary alignment value for structures, unions, and classes

For SH microcomputers, when one instruction is used to access a 4-byte data item, the data item must be allocated at an address that is a multiple of four. Similarly, when one instruction is used to access a 2-byte data item, the data item must be allocated at an address that is a multiple of two. Data items allocated in this manner are aligned on the applicable byte boundary.

For example, when data items are allocated at addresses that are a multiple of four, the data items are aligned on a 4-byte boundary. For details about the byte boundary for each data type, see *10.1.2 Internal Data Representation* in the manual *Compiler User Manual*.

If 1-byte, 2-byte, and 4-byte members all exist in a structure, union, or class, each member uses its respective byte boundary, in which case padding areas might arise between members.

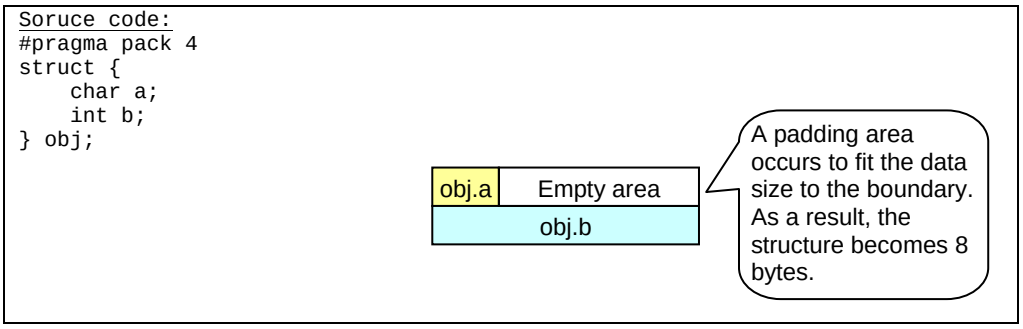


Figure 2-8

In some types of programs, such as communication programs, you might not want structures to have padding areas. In these cases, you can specify the `#pragma pack 1` directive to align structure members on a 1-byte boundary. No padding areas are created in a structure when members are aligned on a 1-byte boundary. Note, however, that because all members in the structure are accessed on a byte basis (byte access), program size might increase. Also note that a member in the structure cannot be accessed by using a pointer. If you attempt to do so, a warning is output.

Instead of using the `#pragma pack` directive, you can also use the `pack` option, which allows you to specify the byte boundary for the structures in each file. If you specify both the `#pragma pack` directive and the `pack` option, the directive takes precedence.

Format:

#pragma pack {1|4}
#pragma unpack

The following table explains how these directives specify the byte boundary.

Table 2-3 Byte boundary for the members of a structure, union, or class

Extention and Member Type	#pragma pack 1	#pragma pack 4	#pragma unpack (or not specified)
[unsigned]char	1	1	1
[unsigned]short or __fixed	1	2	Same as the `pack` option
[unsigned]int, [unsigned]long, [unsigned]long long, long __fixed, __accum, long, __accum, floating-point type, or pointer type	1	4	Same as the `pack` option
Structure, union, or class whose byte boundary is 1	1	1	1
Structure, union, or class whose byte boundary is 2	1	2	Same as the `pack` option
Structure, union, or class whose byte boundary is 4	1	4	Same as the `pack` option

Example:

This example shows how structure members are allocated.

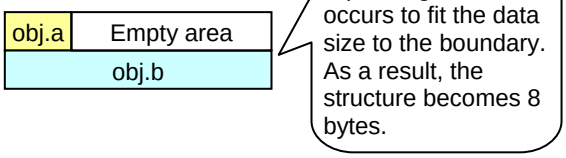
Source code with "#pragma pack 4" specified:

```
#pragma pack 4
struct {
    char a;
    int b;
} obj;

int func(void)
{
    return obj.b;
}
```

Expanded assembly code:

```
_func:
          MOV.L       L11+2,R6   ; _obj
          RTS
          MOV.L       @(4,R6),R0 ; obj.b
L11:
          .RES.W      1
          .DATA.L     _obj
          .SECTION    B,DATA,ALIGN=4
_obj:                            ; static: obj
          .RES.L      2
```



A padding area occurs to fit the data size to the boundary. As a result, the structure becomes 8 bytes.

Source code with "#pragma pack 1" specified:

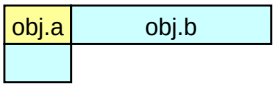
```
#pragma pack 1
struct {
    char a;
    int b;
} obj;

int func(void)
{
    return obj.b;
}
```

Expanded assembly code:

```
_func:
          MOV.L       L11+2,R3   ; H'00000001+_obj
          MOV.B       @R3+,R2    ; (part of)obj.b
          MOV.B       @R3+,R7    ; (part of)obj.b
          SHLL8       R2
          MOV.B       @R3+,R5    ; (part of)obj.b
          EXTU.B      R7,R7
          OR          R2,R7
          SHLL8       R7
          EXTU.B      R5,R4
          MOV.B       @R3,R3     ; (part of)obj.b
          OR          R7,R4
          SHLL8       R4
          EXTU.B      R3,R0
          RTS
          OR          R4,R0
L11:
          .RES.W      1
          .DATA.L     H'00000001+_obj
          .SECTION    B,DATA,ALIGN=4
_obj:                            ; static: obj
          .RES.B      5
```

Members are accessed by byte access.



No padding areas occur because a one-byte boundary is used. The structure becomes 5 bytes.

Note:

Care is required for the structure for I/O registers in the `iodefine.h` file created in Renesas IDE. If you use the `#pragma pack` directive or its equivalent option to specify a 1-byte boundary, the structure members will not indicate correct addresses. Specify `#pragma pack 4` at the beginning of the `iodefine.h` file and `#pragma unpack` at the end of the file.

```
/* iodefine.h */
#pragma pack 4

    ...

#pragma unpack
```

Website and Support <website and support,ws>

Renesas Technology Website
http://japan.renesas.com/

Inquiries
http://japan.renesas.com/inquiry
csc@renesas.com

Revision Record <revision history,rh>

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Sep.01.07	—	First edition issued

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, 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 products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas 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 applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.