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.

RENESAS

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] Option Guide

This document explains the compiler options available in version 9 of the SuperH RISC engine C/C++ compiler.

Table of contents

1. Optimization Options

The compiler optimization options include three basic options (**Optimize for speed, Optimize for size, and, Optimize for both speed and size**) and advanced options, which are used to specify optimization settings in greater detail. Section 1.1 explains the basic options and the advanced options for each. Section 1.2 explains advanced options available for improving 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 (H-1, SH-2, SH-2E, SH-3, or SH4). The code 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 Basic options (Optimize for speed, Optimize for size, Optimize for both speed and size)

The compiler performs two types of optimization: reduction of the object size and reduction of the execution time. If execution speed is the priority, specify the `speed` option. If size is the priority, specify the `size` option. If you want to balance speed and size, specify the `nospeed` option, which is the default.

The following explains these options:

`speed` option:

Performs optimization that reduces execution time but increases object size, as well as performing optimization that reduces both execution time and object size.

`size` option:

Performs optimization that reduces object size but increases execution time, as well as performing optimization that reduces both execution time object size.

`nospeed` option:

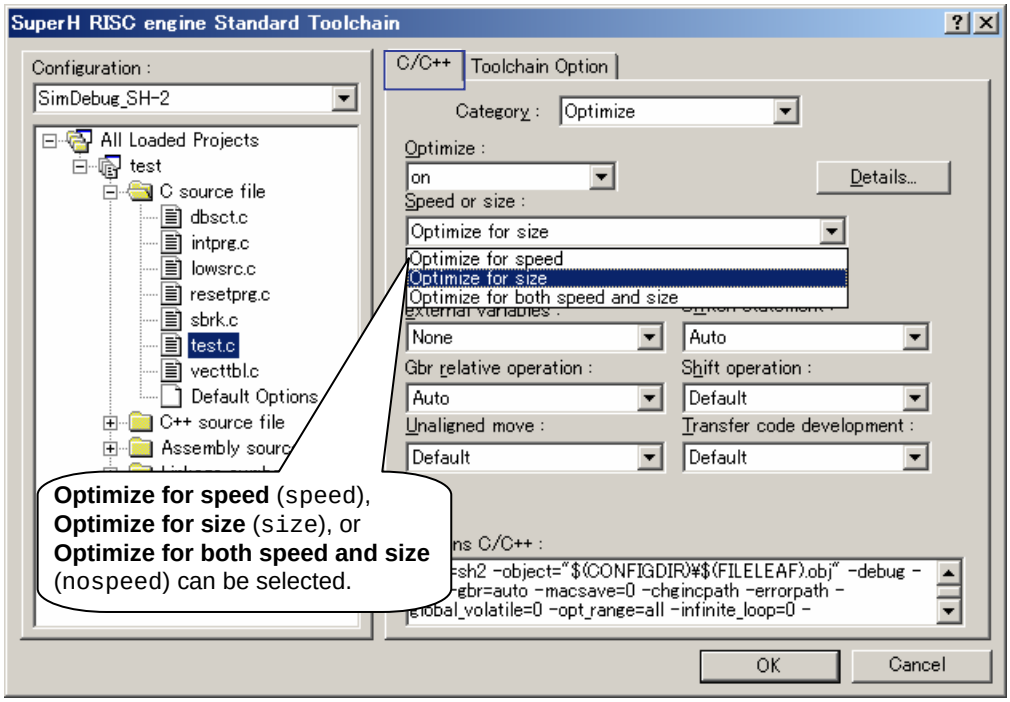
Performs optimization that reduces both execution time and object size.

In an ideal situation, the functions for which speed is the priority and the functions for which size is the priority are stored in separate files, so that the optimization type (speed first or size first) can be selected for each file.

Format:

SPeed
SIze
NOSPeed

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



Supplementary note:

The execution speed on production machines depends not only on the code generated by the compiler, but also on the memory architecture, the cache hit rate, interrupts, and other factors. Consequently, specifying the `speed` option might not always generate the fastest code. Make sure that you check the results of the options described in this document by executing them on production machines.

The defaults of some advanced compiler optimization options depend on the selected basic option. Table 1-1 lists the advanced options whose defaults depend on the selected basic option.

Table 1-1 Basic options ("nospeed", "size", and "speed") and default advanced options

No.	Functionality	nospeed (default)	size	speed	See section
1	Automatic inline expansion	noinline	noinline	Inline	1.1.1
2	Loop unroll	noloop	noloop	loop	1.1.2
3	Shift-operation expansion	Instruction expansion	Run-time routine call	Instruction expansion	1.1.3
4	Transfer-code expansion	Instruction expansion	Run-time routine call	Instruction expansion	1.1.4
5	Method of division (microcomputer other than SH-1)	Instruction expansion	Run-time routine call	Instruction expansion	1.1.5
6	Unaligned data transfer	Instruction expansion	Run-time routine call	Instruction expansion	1.1.6
7	Expansion of constant loading instructions	Literal data reference	Literal data reference	Instruction expansion	1.1.7

The following describes the advanced options listed above.

1.1.1 Automatic inline expansion

Specifies whether to automatically perform inline expansion of functions.

When the `inline` option is specified, the compiler automatically performs inline expansion. The user is able to use `inline=`<*numeric-value*>, to specify the allowed increase in the program's size due to the use of inline expansion. For example, when `inline=50` is specified, inline expansion will be applied until the program has grown to 150% of its size (gain of 50%).

The compiler performs automatic inline expansion by starting with the smallest of the called functions. Note that for the functions in which `#pragma inline` is specified, inline expansion is always performed regardless of the specification of the automatic inline expansion option. Also note that the upper limit on the size that the compiler uses for automatic inline expansion includes the increases in size resulting from inline expansion of `#pragma inline`.

When the `noinline` option is specified, automatic inline expansion is not performed.

Note that automatic inline expansion is not performed for the following functions:

- Functions that have variable parameters
- Functions that perform a call via the address of a function that will be expanded

For details about inline expansion, see *1.2 Performs inline expansion of functions* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler Use guide] Extended Specifications*.

Format:

INLine [= *numeric-value*] : The default advanced option used when "speed" is selected. The default value is 20.
NOINLine : The default advanced option used when "size" or "nospeed" is selected.

Option settings in Renesas IDE:

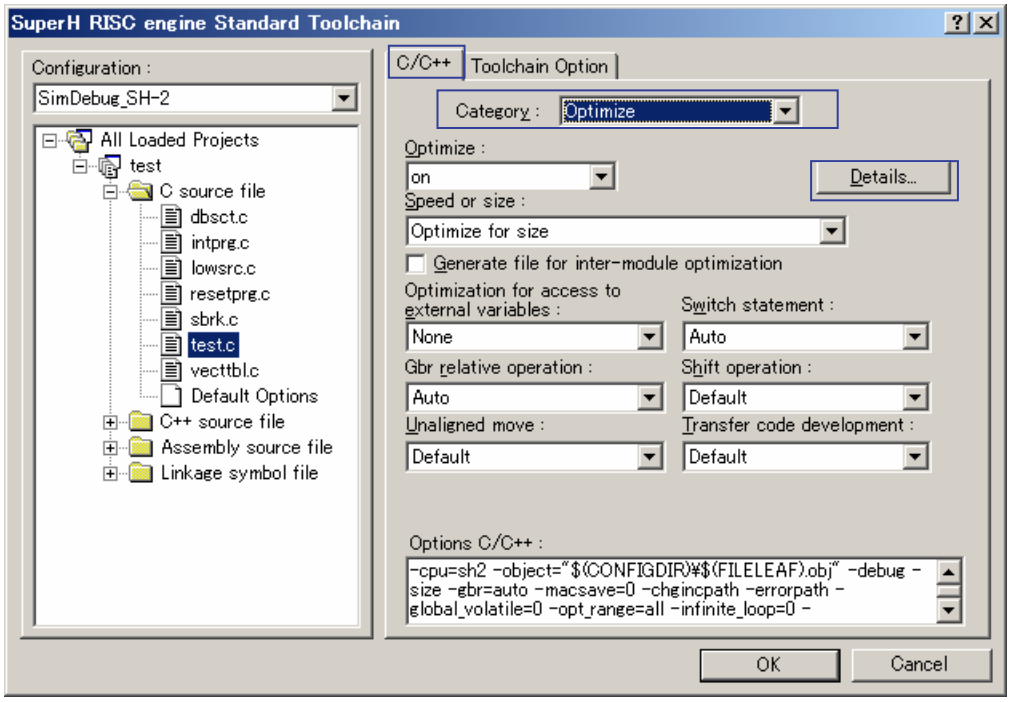


Figure 1-2

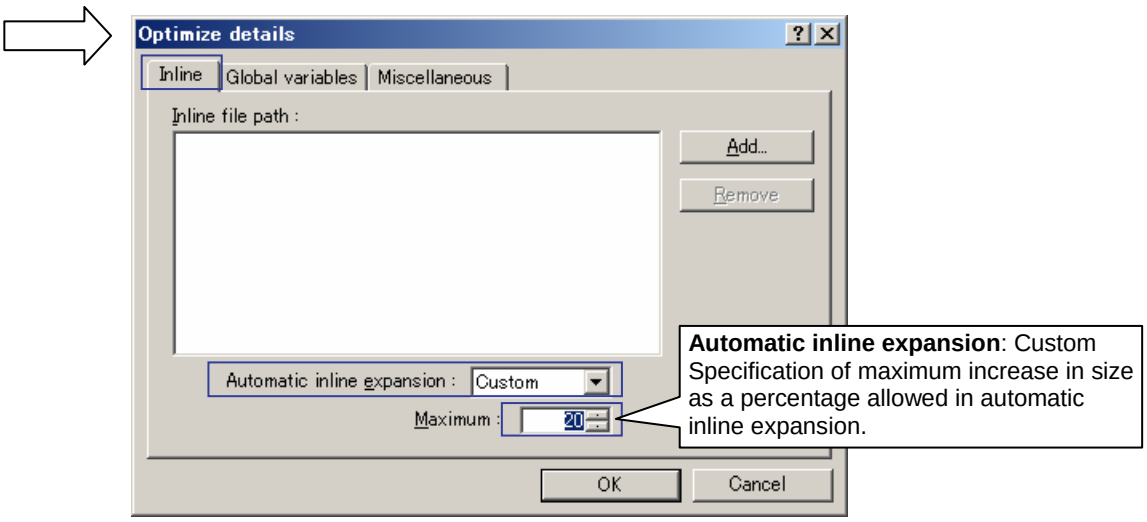


Figure 1-3

Inline expansion requires that the definitions of functions to be expanded can be referenced at compile time. Therefore, in normal inline expansion, only functions that are in the same file can be expanded. If it is necessary to expand functions located in different files, inter-file inline expansion options (`file_inline=`*file-name*`[,...]`) must be specified. Note that if `extern` functions that have the same name are defined in multiple files that are specified for inter-file inline expansion, the compiler does not guarantee the result, since one of the function definitions is selected arbitrarily.

If the source file is specified for inline expansion, the compiler excludes the file from inline expansion and outputs the following warning message:

`C1315 (W) File_inline` *file-name* `ignored by same file as source file`

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

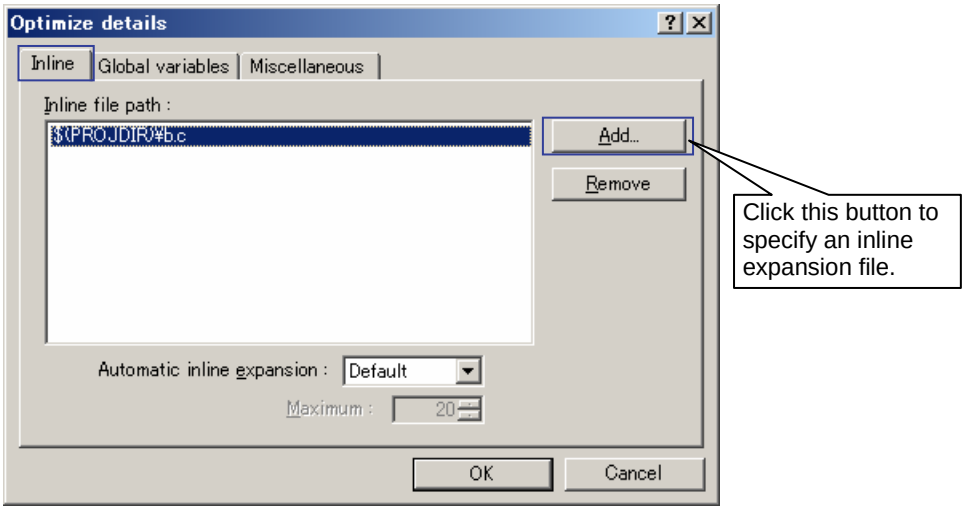


Figure 1-4

Example:

```
Source code
<a.c>
void func(void)
{
    g();
}
<b.c>
#pragma inline (g)
void g(void)
{
    h();
}

file_inline=<source image after a.c is expanded when b.c is specified>
void func(void)
{
     h();
}
```

The `file_inline_path` option is useful when you specify files that are located in folders other than the current folder for inter-file inline expansion. If you specify the names of these folders beforehand in the `file_inline_path=`*path-name*`[,...]` format, you do not need to specify the path names of the target files.

The compiler searches the folders specified in the `file_inline_path` option for the target files, and then searches the current folder.

Option settings in Renesas IDE:

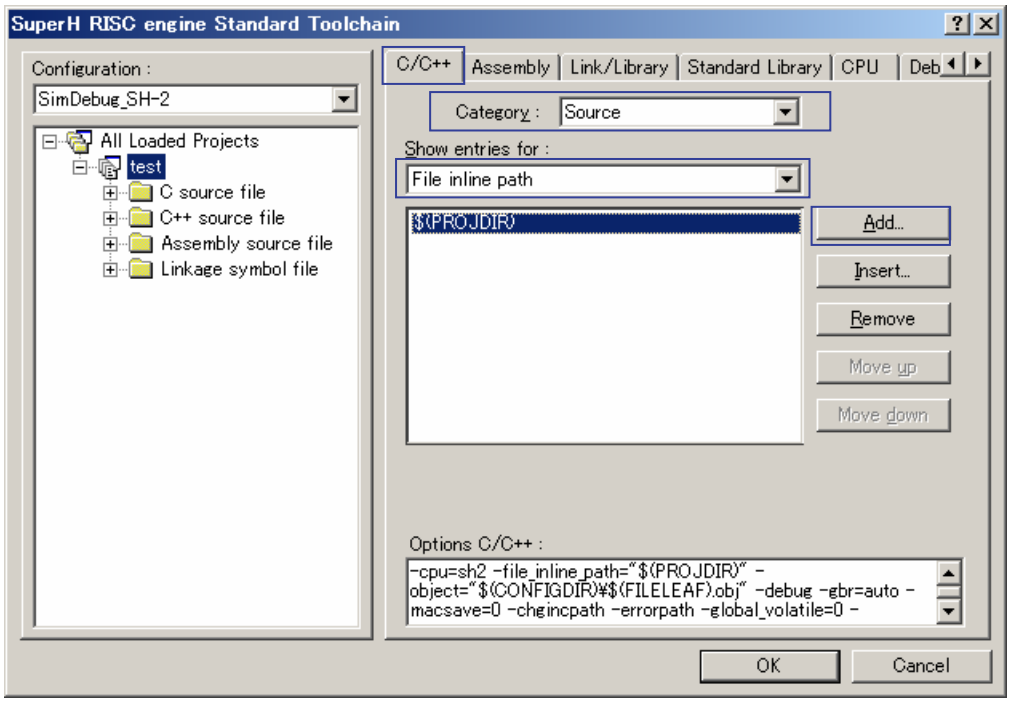


Figure 1-5

1.1.2 Loop unroll

Specifies whether to perform loop unrolling.

Specifying the `loop` option enables loop expansion optimization. For details about loop expansion optimization, see *4.1 Reducing the number of times a loop is repeated* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler use guide] Efficient programming techniquesCompiler*.

You can use the `max_unroll=`*numeric-value* (*numeric-value*: 1-32) option to specify the maximum number of loop expansions. If loop expansion optimization is enabled, the option default is 2. If loop expansion optimization is disabled, the `max_unroll` specification is ignored.

Format:

LOop : The default advanced option used when the basic option is "speed" is selected. The default value is 2.
NOLOop : The default advanced option used when the basic option is "size" or "nospeed" is specified.

Option settings in Renesas IDE:

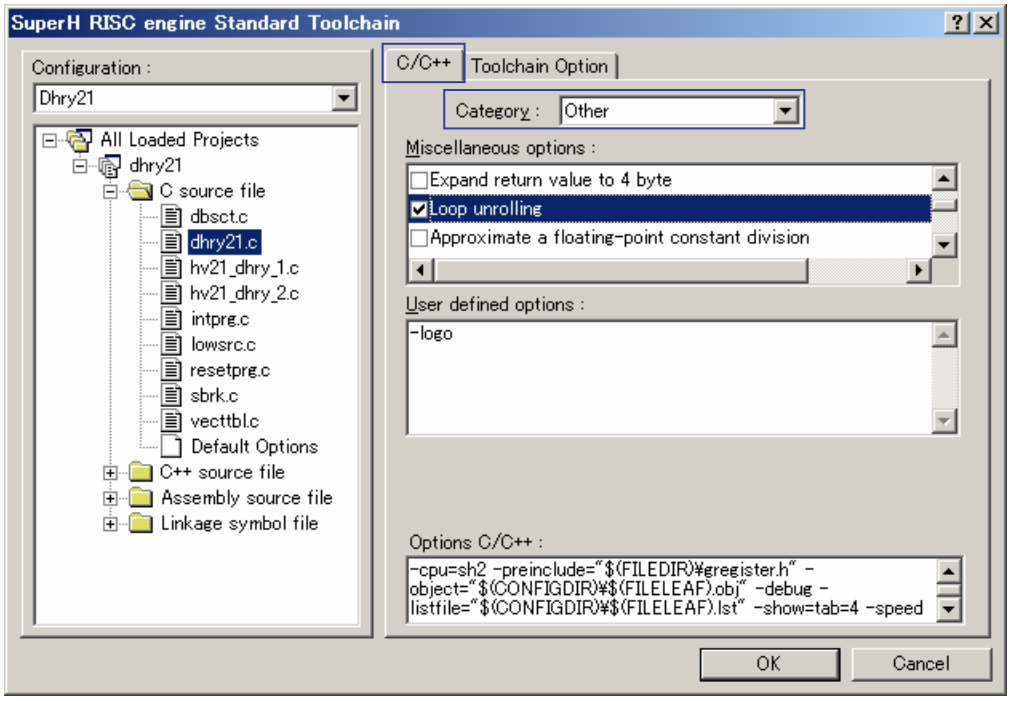


Figure 1-6

To specify the maximum number of loop expansions, click the **Compiler** tab in the SuperH RISC engine Standard Toolchain dialog box. Then select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

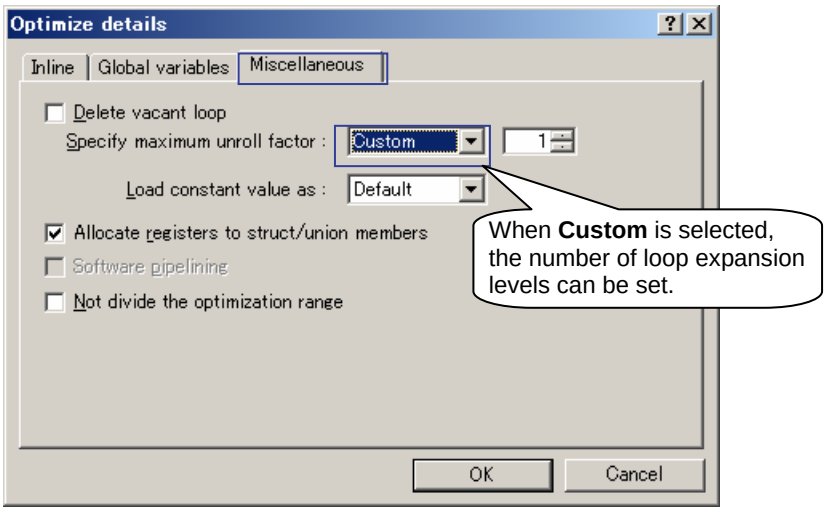


Figure 1-7

1.1.3 Shift-operation expansion

You can select whether shift operations are to be expanded into instructions or treated as run-time routine calls.

If `inline` (instruction expansion) is specified, shift operations are always expanded into instructions. If `runtime` (run-time routine call) is specified, the processing differs depending on the number of instructions into which the operation will be expanded. If the number of instructions will exceed 5, the operation is treated as a run-time routine call. If the number of instructions will not exceed 5, the operation is expanded into instructions.

Format:

SHIft = Inline : The default advanced option used when the basic option is "speed" or "nospeed".
Runtime : The default advanced option used when the basic option is "size".

Option settings in Renesas IDE:

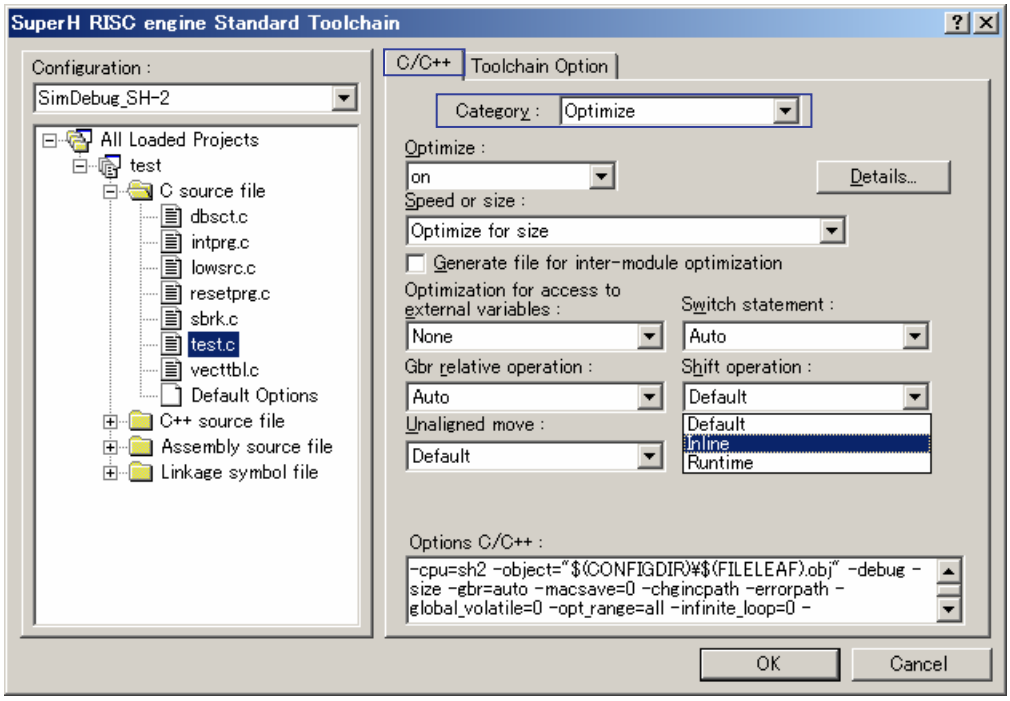


Figure 1-8

Example:

Source code:

```
int var;

void f(void)
{
    var >>= 11;
}
```

Expanded assembly code (shift=inline specified)

```
_f:
          MOV.L       L11+2,R5    ; _var
          MOV.L       @R5,R2      ; var
          SHLR8       R2
          SWAP.W      R2,R2
          EXTS.B      R2,R6
          XTRCT       R6,R2
          SHAR        R2
          SHAR        R2
          SHAR        R2
          RTS
          MOV.L       R2,@R5      ; var
L11:
          .RES.W      1
          .DATA.L     _var
```

Expanded assembly code (shift=runtime specified)

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11,R5      ; _var
          MOV.L       L11+4,R2    ; __sta_sftra11
          JSR         @R2
          MOV.L       @R5,R0      ; var
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R5      ; var
L11:
          .DATA.L     _var
          .DATA.L     __sta_sftra11
```

1.1.4 Transfer-code expansion

You can select whether the transfer code of a structure, array, or class is expanded into instructions or treated as a run-time routine call.

If `inline` is specified, transfer code is always expanded into instructions. If `runtime` is specified, the processing differs depending on the number of instructions into which the code will be expanded. If the code can be copied with two pairs of load/stores (4 instructions), the code is expanded into instructions. If the code cannot be copied with two pairs of load/stores (4 instructions), the code is treated as a run-time routine call.

Format:

BLOckcopy = Inline : The default advanced option used when the basic option is "speed" or "nospeed".
Runtime : The default advanced option used when the basic option is "size".

Option settings in Renesas IDE:

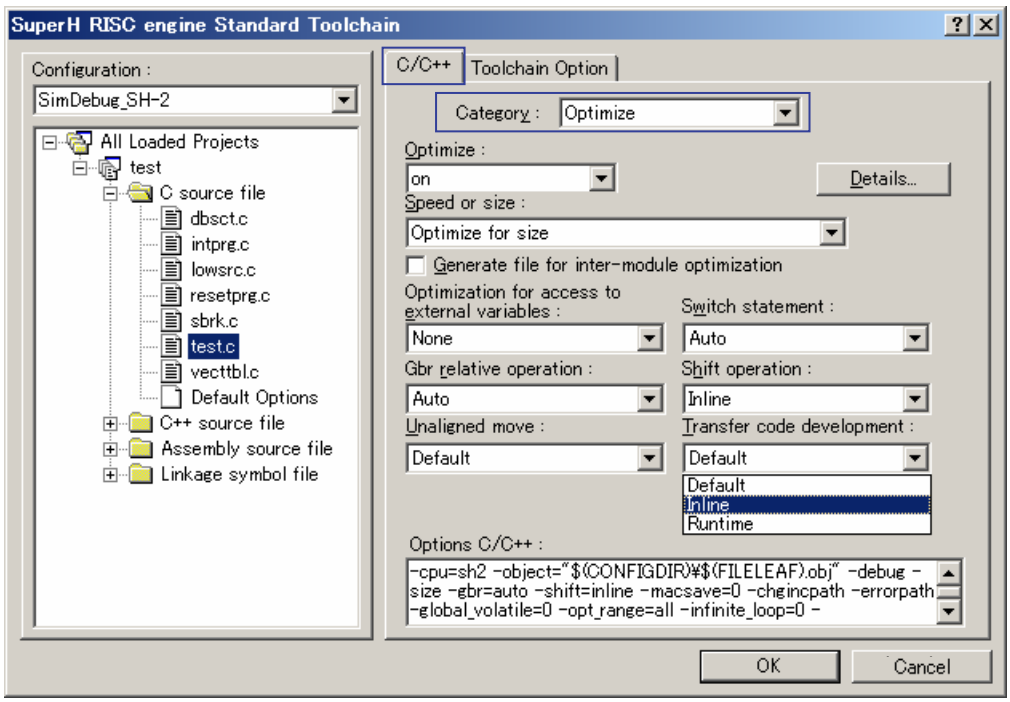


Figure 1-9

Example:

Source code:

```
struct _ST_ {
   char a[5];
} x;

extern void g(struct _ST_);

void f(void)
{
    g(x);
}
```

Expanded assembly code (blockcopy=inline specified)

```
_f:
          STS.L       PR,@-R15
          ADD         #-8,R15
          MOV.L       L11+2,R6    ; _x
          MOV.L       L11+6,R4    ; _g
          MOV.B       @(1,R6),R0 ; (part of)x
          MOV.B       @R6,R1      ; (part of)x
          MOV.B       R0,@(1,R15)
          MOV.B       @(2,R6),R0 ; (part of)x
          MOV.B       R1,@R15
          MOV.B       R0,@(2,R15)
          MOV.B       @(3,R6),R0 ; (part of)x
          MOV.B       R0,@(3,R15)
          MOV.B       @(4,R6),R0 ; (part of)x
          JSR         @R4
          MOV.B       R0,@(4,R15)
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _x
          .DATA.L     _g
```

Expanded assembly code (blockcopy=runtime specified)

```
_f:
          STS.L       PR,@-R15
          ADD         #-8,R15
          MOV.L       L11,R2      ; _x
          MOV.L       L11+4,R5    ; __slow_mvn
          MOV         R15,R1
          JSR         @R5
          MOV         #5,R0       ; H'00000005
          MOV.L       L11+8,R1    ; _g
          JSR         @R1
          NOP
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .DATA.L     _x
          .DATA.L     __slow_mvn
          .DATA.L     _g
```

1.1.5 Method of division (microcomputer other than SH-1)

You can select the method used for integer-type division and remainder calculation in the program. This option has no effect when the microcomputer is SH-1.

If `division=cpu=inline` is specified, constant division is converted to multiplication by inline expansion. Variable division is processed differently depending on the microcomputer type. If the microcomputer is SH-2A or SH2A-FPU, variable division is expanded into instructions. If the microcomputer is not SH-2A or SH2A-FPU, variable division is treated as a run-time routine call.

If `division=cpu=runtime` is specified, power-of-two constant division is expanded into instructions. Other types of constant division are processed differently depending on the microcomputer type. If the microcomputer is SH-2A or SH2A-FPU, the division operation is expanded into instructions. If the microcomputer is not SH-2A or SH2A-FPU, the division operation is treated as a run-time routine call.

Format:

`DIvision = Cpu = Inline` **: The default advanced option used when the basic option is "speed" or "nospeed".**
Runtime : The default advanced option used when the basic option is "size".

Option settings in Renesas IDE:

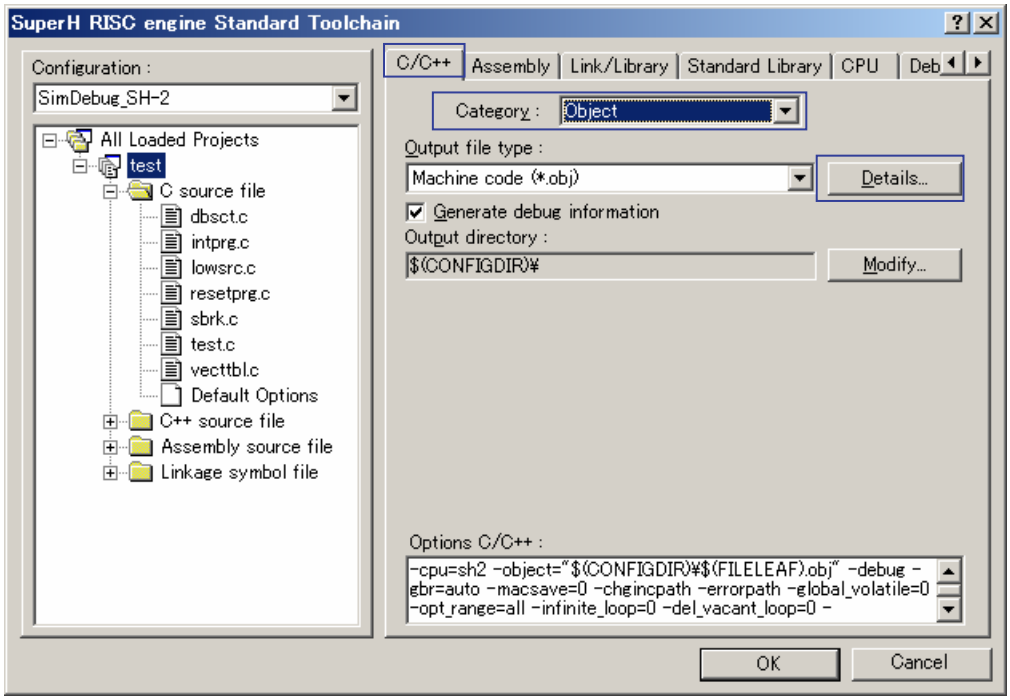


Figure 1-10

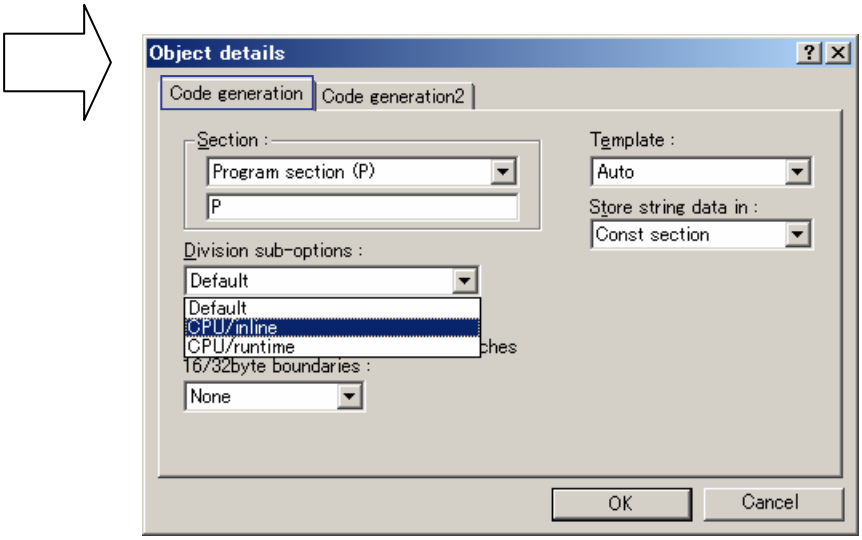


Figure 1-11

Example:

Source code:

```
int x;
void f(int y)
{
    x = y/3;
}
```

Expanded assembly code (division=cpu=inline specified)

```
_f:
        STS.L       MACL,@-R15
        STS.L       MACH,@-R15
        MOV.L       L11,R1      ; H'55555556
        MOV.L       L11+4,R5    ; _x
        DMULS.L     R4,R1
        STS         MACH,R6
        MOV         R6,R0
        ROTL        R0
        AND         #1,R0
        ADD         R0,R6
        MOV.L       R6,@R5      ; x
        LDS.L       @R15+,MACH
        RTS
        LDS.L       @R15+,MACL
L11:
        .DATA.L     H'55555556
        .DATA.L     _x
```

Expanded assembly code (division=cpu=runtime specified)

```
_f:
        STS.L       PR,@-R15
        MOV.L       L11+2,R2    ; __divls
        MOV         R4,R1
        JSR         @R2
        MOV         #3,R0       ; H'00000003
        MOV.L       L11+6,R5    ; _x
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R5      ; x
L11:
        .RES.W      1
        .DATA.L     __divls
        .DATA.L     _x
```

1.1.6 Unaligned data transfer

You can select whether instruction expansion or a run-time routine call should be applied to the data transfer of a structure, union, or class whose alignment value is 1. If `inline` (instruction expansion) is specified, the transfer is always expanded into instructions. If `runtime` (run-time routine call) is specified, the transfer is expanded into instructions unless the number of instructions after the expansion would be large. If the number of instructions would be large after the expansion, the transfer is treated as a run-time routine call.

Format:

Unaligned = Inline : The default advan ced option used when the basic option is "speed" or "nospeed".

Runtime : The default advanced option used when the basic option is "size".

Option settings in Renesas IDE:

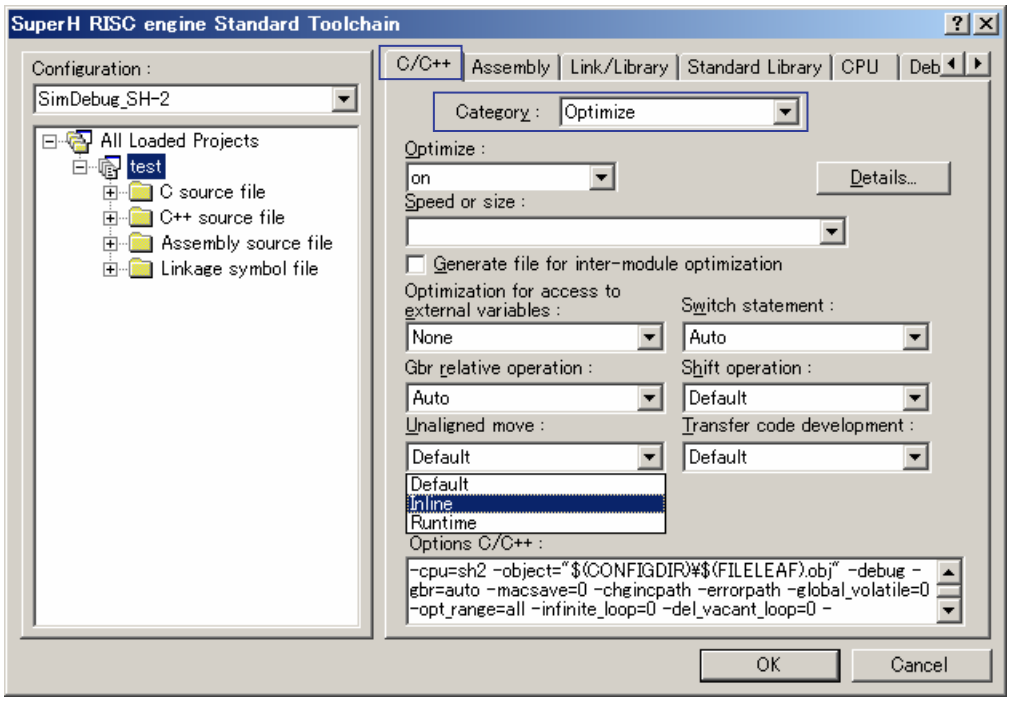


Figure 1-12

Example:

Source code:

```
#pragma pack 1
struct {
   char  a;
   short b;
   int   c;
} x,y;
#pragma unpack

void func(void)
{
    x.c = y.c;
}
```

Expanded assembly code (unaligned=inline specified)

```
_func:
          MOV.L       L11+2,R3    ; H'00000003+_y
          MOV.L       L11+6,R7    ; H'00000003+_x
          MOV.B       @R3,R4      ; y.c
          MOV.B       @(1,R3),R0  ; y.c
          MOV.B       R4,@R7      ; x.c
          MOV.B       R0,@(1,R7)  ; x.c
          MOV.B       @(2,R3),R0  ; y.c
          MOV.B       R0,@(2,R7)  ; x.c
          MOV.B       @(3,R3),R0  ; y.c
          RTS
          MOV.B       R0,@(3,R7)  ; x.c
L11:
          .RES.W      1
          .DATA.L     H'00000003+_y
          .DATA.L     H'00000003+_x
```

Expanded assembly code (unaligned=runtime specified)

```
_func:
          STS.L       PR,@-R15
          MOV.L       L11+2,R2    ; H'00000003+_y
          MOV.L       L11+6,R1    ; H'00000003+_x
          MOV.L       L11+10,R7   ; __slow_mvn
          JSR         @R7
          MOV         #4,R0       ; H'00000004
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     H'00000003+_y
          .DATA.L     H'00000003+_x
          .DATA.L     __slow_mvn
```

1.1.7 Expansion of constant loading instructions

You can select whether a constant load is expanded into instructions (`inline`) or treated as a literal load (`literal`).

In SH microcomputers, instructions can hold eight-bit constants (20-bit constants in SH-2A microcomputers). A 2-byte or 4-byte constant is handled in either of the following ways:

Literal load: Constant data (a literal) prepared in memory is loaded into a register.
Instruction expansion: Eight-bit constants are computed to obtain the 2-byte or four-byte constant.

When a literal load is used, the program size is likely to be smaller. When instruction expansion is used, the number of memory accesses is likely to be smaller. If `literal` is specified, a literal load is used only when the constant is two bytes or larger. If `inline` is specified, all 1-byte and 2-byte constants and some 4-byte constants are obtained by instruction expansion.

When the basic option is `size` or `nospeed`, instruction expansion is used if the constant satisfies the following condition, and a literal load is used if the constant does not satisfy the condition:

2-byte constant: Obtained from two or fewer instructions
4-byte constant: Obtained from three or fewer instructions

Format:

CONST_Load = Inline : The default advanced option used when the basic option is "speed".
Literal : The default advanced option used when the basic option is "size" or "nospeed". (Note, however, that when "size" or "nospeed" is selected, instruction expansion occurs depending on the condition.)

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

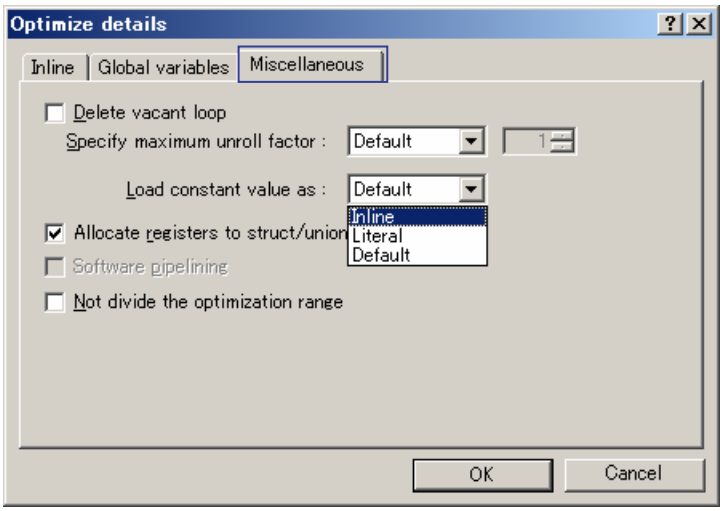


Figure 1-13

Example:

```
Source code:
int a;

void func(void)
{
    a = 0x4567;
}
```

```
Expanded assembly code (const_load=inline
specified)
_func:
          MOV         #69,R2      ; H'00000045
          SHLL8       R2
          MOV.L       L11,R6      ; _a
          ADD         #103,R2
          RTS
          MOV.L       R2,@R6      ; a
L11:
          .DATA.L     _a
```

```
Expanded assembly code (const_load=literal
specified)
_func:
          MOV.L       L11+4,R6    ; _a
          MOV.W       L11,R2      ; H'4567
          RTS
          MOV.L       R2,@R6      ; a
L11:
          .DATA.W     H'4567
          .RES.W      1
          .DATA.L     _a
```

Supplementary note:

The optimal setting of this option differs depending on the memory architecture of the target system. In a system in which memory access is fast, literal access is likely to be executed faster than instruction expansion. In a system in which memory access is slow, instruction expansion is likely to be executed faster than literal access.

1.2 Advanced options for improving performance

This section explains the advanced optimization options, use of which can improve performance.

Table 1-2 Advanced options for improving performance

No.	Functionality	Option	Effectiveness on size	Effectiveness on speed	See section
1	Specifies address range	`abs16`, `abs20`, `abs28`, `abs32`	A+	A	1.2.1
2	Disposition of variables	`stuff`, `nostuff`	A+	--	1.2.2
3	Optimized for access to external variables	`map`, `smap`	A+	A+	1.2.3
4	GBR Relative Logic Operation Generation	`logic_gbr`	A	A	1.2.4
5	Division of optimizing ranges	`scope`, `noscope`	B	B	1.2.5
6	MAC register	`macsave`	A	A	1.2.6
7	Extension of return value	`rtnext`, `nortnext`	B	B	1.2.7
8	Enumeration data size	`auto_enum`	A	C	1.2.8
9	Switch statement expansion method	`case`	B	B	1.2.9

A+: Very effective.
A: Effective.
B: Sometimes effective, sometimes lowers performance.
C: Lowers performance.
--: No effect.

1.2.1 Specifies address range

The address area declarations `abs16`, `abs20`, `abs28`, and `abs32` tell the compiler that the variable or function is in the 16-, 20-, 28-, or 32-bit address areas, respectively. The default is the 32-bit address area.

The `#pragma abs16`, `abs20`, `abs28`, or `abs32` directive can also be used to declare an address area. If both the `#pragma` directive and the `abs16`, `abs20`, `abs28`, or `abs32` option are specified, the `#pragma` directive takes precedence.

For details about address area declaration, see *1.1 Specifies address range* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler Use guide] Extended Specifications.*

Format:

ABs16 = { Program | Const | Data | Bss | Run | All }[,...]
ABS20 = { Program | Const | Data | Bss | Run | All }[,...]
ABS28 = { Program | Const | Data | Bss | Run | All }[,...]
<u>ABS32</u> = { Program | Const | Data | Bss | Run | <u>All</u> }[,...]

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Object** from the **Category** drop-down list, and click **Details** (Figure 1-10). In the displayed dialog box, shown below, specify the settings as follows.

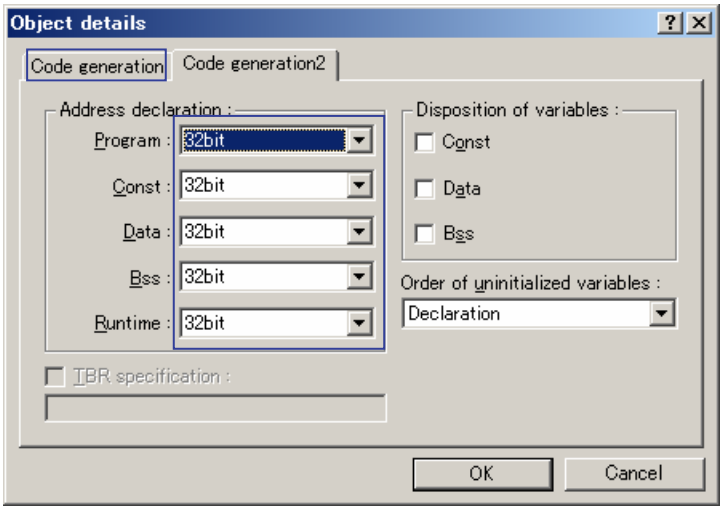


Figure 1-14

1.2.2 Disposition of variables

You can use the stuff option to align variables to any byte boundary alignment sections depending on the size of the variables. This option can eliminate the empty (padded) areas that are used for boundary adjustment, thus conserving memory.

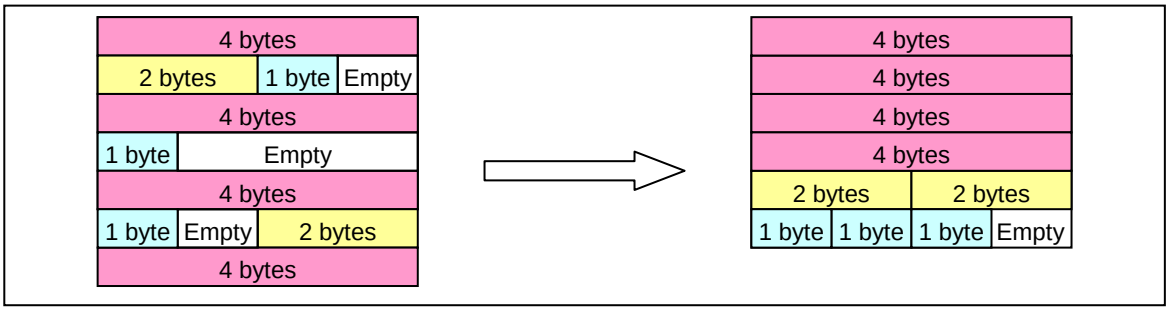


Figure 1-15

In the `stuff` option, you can also specify a section type. If a section type is specified, the variables that belonging to the section type are assigned to 4-byte, 2-byte, or 1-byte boundary alignment sections depending on the size of the variables. If a section type is not specified, all types of sections are subject to the alignment.

The data within a section is output in the order in which it is defined. Note that the `bss_order=declaration` specification is ignored.

If `nostuff` is specified, all variables are placed in the 4-byte-boundary section. The order of the data placed in each section differs depending on the section type. For the sections of type `const` or `data`, data is placed in the order in which it is defined. For the sections of type `bss`, the data is ordered according to the `bss_order` specification. `nostuff` is the default.

Table 1-3 Variable sizes and section names

	Section type	Default section name	Variable size		
			4n	*4n* + 2	2*n* + 1
Constant area	const	C	C$4	C$2	C$1
Initialized data area	data	D	D$4	D$2	D$1
Uninitialized data area	bss	B	B$4	B$2	B$1

If a default section name has been changed, the new default section name replacing `C`, `D`, or `B` is followed by `$4`, `$2`, or `$1`.

Format:

STUff [= *section-type*[,...]]
NOSTuff
***section-type*: { Bss | Data | Const }**

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Object** from the **Category** drop-down list, and click **Details** (Figure 1-10). In the displayed dialog box, shown below, specify the settings as follows.

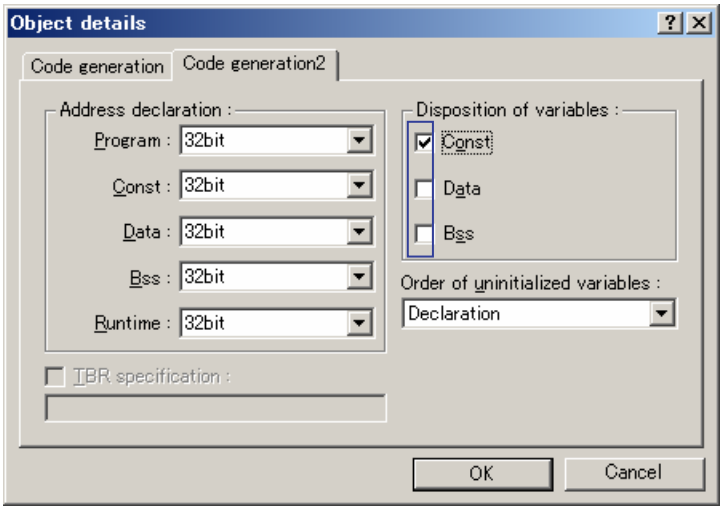


Figure 1-16

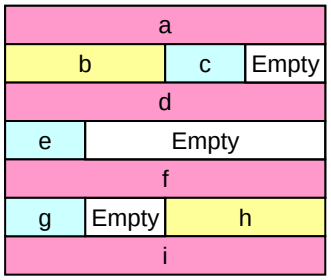
Example:

Source code:

```
int   a;
short b;
char  c;
int   d;
char  e;
int   f;
char  g;
short h;
int   i;
```

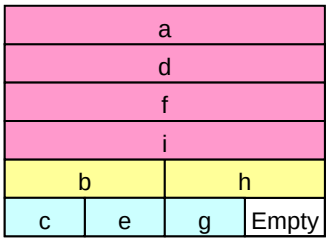
Expanded assembly code (nostuff specified)

```
          .SECTION    B,DATA,ALIGN=4
_a:                                   ; static: a
          .RES.L      1
_b:                                   ; static: b
          .RES.W      1
_c:                                   ; static: c
          .RES.B      1
          .RES.B      1
_d:                                   ; static: d
          .RES.L      1
_e:                                   ; static: e
          .RES.B      1
          .RES.B      1
          .RES.W      1
_f:                                   ; static: f
          .RES.L      1
_g:                                   ; static: g
          .RES.B      1
          .RES.B      1
_h:                                   ; static: h
          .RES.W      1
_i:                                   ; static: i
          .RES.L      1
```



Expanded assembly code (stuff specified)

```
          .SECTION    B$4,DATA,ALIGN=4
_a:                                   ; static: a
          .RES.L      1
_d:                                   ; static: d
          .RES.L      1
_f:                                   ; static: f
          .RES.L      1
_i:                                   ; static: i
          .RES.L      1
          .SECTION    B$2,DATA,ALIGN=2
_b:                                   ; static: b
          .RES.W      1
_h:                                   ; static: h
          .RES.W      1
          .SECTION    B$1,DATA,ALIGN=1
_c:                                   ; static: c
          .RES.B      1
_e:                                   ; static: e
          .RES.B      1
_g:                                   ; static: g
          .RES.B      1
```



1.2.3 Optimized for access to external variables

The `map` and `smap` options are provided so that accesses to external variables can be performed as relative accesses from a base external variable. As a result, the loading of the addresses of external variables becomes unnecessary, improving execution speed. Since the address value literals can be omitted, program size is also reduced. If `gbr=auto` has been specified, an external variable might be accessed with a GBR relative instruction whose relative value is larger than the normal MOV instruction. External variable access optimization is effective for optimizing both execution speed and program size.

When the `map` option is used, optimization requires the external symbol allocation information generated by the Optimizing Linkage Editor. For this reason, compilation must be performed twice.

When the `smap` option is used, external variable optimization is performed for only external variables defined in the file to be compiled. Since the external symbol allocation information generated by the Optimizing Linkage Editor is not required, compilation is required only once.

Although the optimization implemented by the `map` option is more effective than the optimization implemented by the `smap` option, compilation is required twice for the `smap` option optimization. Furthermore, the optimization can be performed only when an address-resolved execution module (such as `abs` or `mot`) is generated. When the `smap` option optimization is used, compilation is required only once, and can be performed when a file whose addresses have not been resolved (such as a library file). Note that in this case, however, only the external variables defined in the file can be optimized.

Table 1-4 Advantages and disadvantages of "map" and "smap"

Option	Number of times compilation required	Build time	External symbol allocation information file generated by the Optimizing Linkage Editor	Effectiveness	Address resolution
map	Twice	Long	Required	High	Required
smap	Once	Short	Not required	Low	Not required

Format:

- Inter-module specification

  MAP = ***file-name***

  Perform compilation once without specifying the `map` option, and then, during linkage, specify `map=`*file-name* to generate an external symbol allocation information file. Next, perform a second compilation with the external symbol allocation information file (`map=`*file-name*) specified.

  Note that if the definition order of external variables or static variables is changed, you must regenerate the external symbol allocation information file.

  Also note that the result is not guaranteed if the second compilation satisfies either of the following conditions:

  - Options other than the options specified for the first compilation and the `map` option are specified.
  - The specified source file differs from the source file specified for the first compilation.

- Intra-module specification

  SMap

Option settings in Renesas IDE:

If you change the scope of external variable access optimization to **Inner-module** from another scope or from **Inter-module** to another scope, a warning message appears. The reason this message appears is that changing of this setting automatically enables or disables generation of the external symbol allocation information file from the Optimizing Linkage Editor.

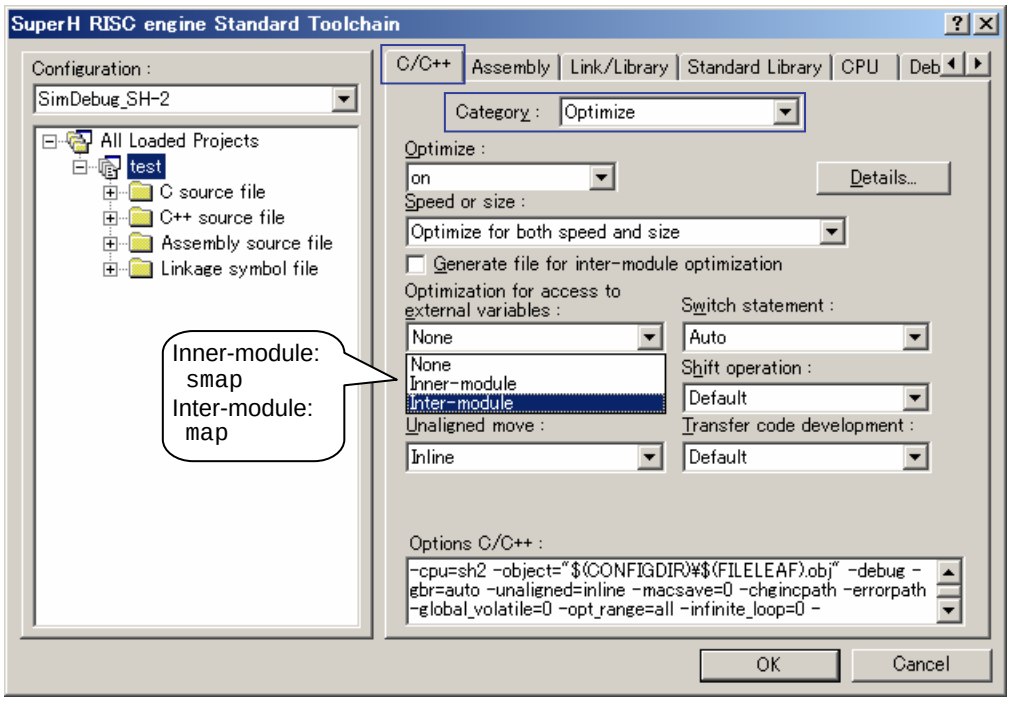


Figure 1-17

Example 1:

In this example, variables that are allocated in succession are accessed by relative access in the same register based on the variable allocation order.

```
Source code:
int a,b;
void f(void)
{
   a=0;
   b=0;
}

Expanded assembly code (map/smap not specified)
_f:
    MOV.L       L11,R1      ; _a
    MOV.L       L11+4,R4    ; _b
    MOV         #0,R2       ; H'00000000
    MOV.L       R2,@R1      ; a
    RTS
    MOV.L       R2,@R4      ; b
L11:
    .DATA.L     _a
    .DATA.L     _b
```

```
Expanded assembly code (map/smap specified)
_f:
    MOV.L       L11+2,R6    ; _a
    MOV         #0,R2       ; H'00000000
    MOV.L       R2,@R6      ; a
    RTS
    MOV.L       R2,@(4,R6) ; b
L11:
    .RES.W      1
    .DATA.L     _a
```

Example 2:

In this example, GBR is used as the base for accessing external variables when the `gbr=auto` option (default) is specified.

```
Source code:
int a[100];
void f(void)
{
    a[0]=0;
    a[50]=0;
    a[51]=0;
    a[52]=0;
}

Expanded assembly code (map/smap not specified)
_f:
          MOV.L       L11+2,R5    ; _a
          MOV         #-56,R0     ; H'FFFFFFC8
          MOV         #0,R4       ; H'00000000
          EXTU.B      R0,R0
          MOV.L       R4,@R5      ; a[]
          MOV.L       R4,@(R0,R5); a[]
          ADD         #4,R0
          MOV.L       R4,@(R0,R5); a[]
          ADD         #4,R0
          RTS
          MOV.L       R4,@(R0,R5); a[]
L11:
          .RES.W      1
          .DATA.L     _a
```

```
Expanded assembly code (map/smap specified)
_f:
          STC         GBR,@-R15
          MOV.L       L11,R0      ; _a
          LDC         R0,GBR
          MOV         #0,R0       ; H'00000000
          MOV.L       R0,@(0,GBR); a[]
          MOV.L       R0,@(200,GBR); a[]
          MOV.L       R0,@(204,GBR); a[]
          MOV.L       R0,@(208,GBR); a[]
          RTS
          LDC         @R15+,GBR
L11:
          .DATA.L     _a
```

1.2.4 GBR Relative Logic Operation Generation

If a GBR-relative logical operation code can be generated for an external variable for which `#pragma gbr_base` or `gbr_base1` is not specified, the external variable can be accessed by GBR relative access code.

A GBR-relative logical operation code can be generated for the following operations:

- Bitwise operation (AND, OR, or XOR) for an external variable of type `char` or `unsigned char`
- Reference of a bit field of an external variable

Format:

`LOGIc_gbr`

This option takes effect only when `gbr=user` is specified. Before you use this option, you must allocate the $G0 section by the linkage editor and set the first address of the section in the GBR register. You can use the `set_gbr()` intrinsic function to set the address.

Example 1:

Source code:

```
char a;

void func(void)
{
    a &= 0x0f;
}
```

Expanded assembly code (logic_gbr not specified)

```
_func:
          MOV.L       L11+2,R6    ; _a
          MOV.B       @R6,R0      ; a
          AND         #15,R0
          RTS
          MOV.B       R0,@R6      ; a
L11:
          .RES.W      1
          .DATA.L     _a
```

Expanded assembly code (logic_gbr specified)

```
_func:
          MOV.L       L11+2,R0    ; _a-(STARTOF $G0)
          RTS
          AND.B       #15,@(R0,GBR); a
L11:
          .RES.W      1
          .DATA.L     _a-(STARTOF $G0)
```

Example 2:

Source code:

```
struct {
    unsigned char a:1;
    unsigned char b:1;
} x;

void func(void)
{
   if (x.a) {
        x.b = 1;
    }
}
```

Expanded assembly code (logic_gbr not specified)

```
_func:
          MOV.L       L13,R5      ; _x
          MOV.B       @R5,R0      ; (part of)x
          TST         #128,R0
          BT          L12
          OR          #64,R0
          MOV.B       R0,@R5      ; (part of)x
L12:
          RTS
          NOP
L13:
          .DATA.L     _x
```

Expanded assembly code (logic_gbr specified)

```
_func:
          MOV.L       L13,R0      ; _x-(STARTOF $G0)
          TST.B       #128,@(R0,GBR); (part of)x
          BT          L12
          OR.B        #64,@(R0,GBR); (part of)x
L12:
          RTS
          NOP
L13:
          .DATA.L     _x-(STARTOF $G0)
```

1.2.5 Division of optimizing ranges

You can specify whether to divide the optimization range of a function at compile time.

If the `scope` option is specified, the optimization range of a large function might be divided during compilation.

If the `noscope` option is specified, the optimization range of a function is not divided. In this case, since the entire function can be optimized, normally both the program size and execution time can be reduced. However, if there are not enough registers, performance might be degraded.

Since the more effective option depends on the program, try both options while tuning the performance.

Note that compilation takes more time when the optimization scope is not divided.

Format:

`SCOpe`
`NOSCope`

An information-level message indicates whether the optimization range of a function has been divided. Information-level messages are output when the `message` option is specified.

The following is an information-level message that indicates that the optimization scope has been divided:

`C0101 (I) Optimizing range divided in function "`*function-name*`"`

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

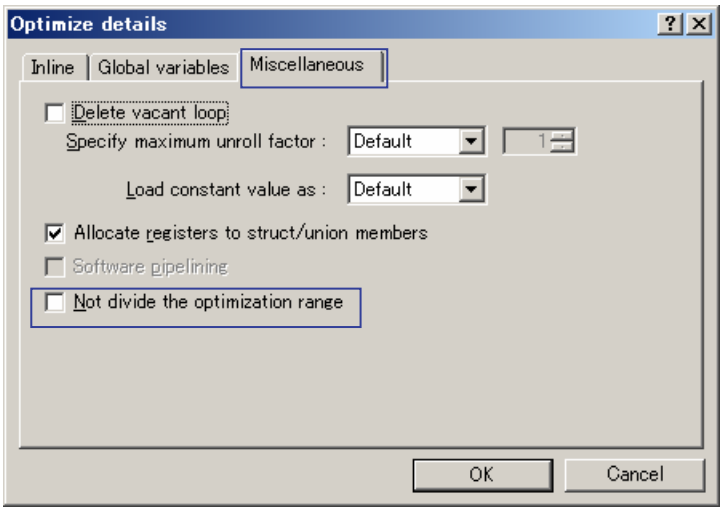


Figure 1-18

1.2.6 MAC register

You can specify whether the contents of the MACH and MACL registers are to be guaranteed at function entry and exit points.

If `macsave=1` is specified, the contents of these registers are guaranteed. That is, the contents of the registers are saved and restored at function entry and exit points. If `macsave=0` is specified, the contents of these registers are not guaranteed. In this case, the contents of the registers are saved and restored on the function caller side.

In optimization by the current version of the compiler, the `macsave=0` specification is likely to reduce both the program size and the execution time. However, for compatibility with previous versions, the default is `macsave=1`.

Since performance might improve, you should try specifying `macsave=0`.

Format:

```
Macsave = { 0 | 1 }
```

Caution:

A function compiled with `macsave=0` specified (MAC registers not guaranteed) cannot be called from a function compiled with `macsave=1` specified (MAC registers guaranteed). However, the reverse situation is possible.

Option settings in Renesas IDE:

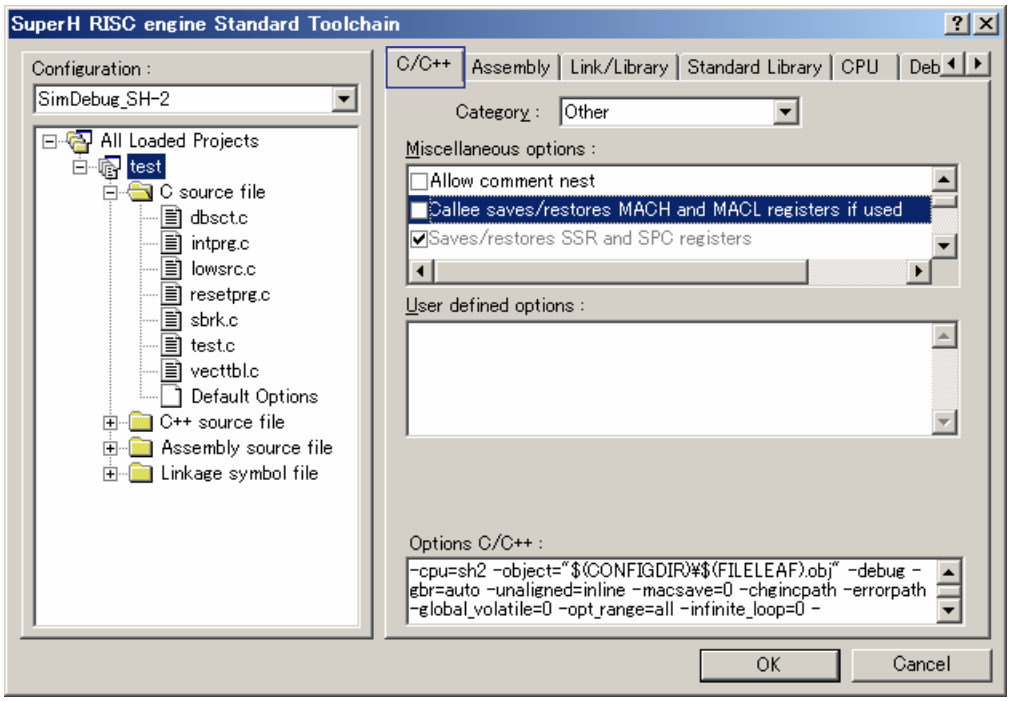


Figure 1-19

Example:

Source code:

```
int sum;
int func(short a, short b)
{
   sum += a * b;

   return sum;
}
```

Expanded assembly code (macsave=1 specified)

```
_func:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          MOV.L       L11+2,R1    ; _sum
          MOV.L       @R1,R0      ; sum
          STS         MACL,R2
          ADD         R2,R0
          MOV.L       R0,@R1      ; sum
          RTS
          LDS.L       @R15+,MACL
L11:
          .RES.W      1
          .DATA.L     _sum
```

Expanded assembly code (macsave=0 specified)

```
_func:
          MULS.W      R4,R5
          MOV.L       L11+2,R1    ; _sum
          MOV.L       @R1,R0      ; sum
          STS         MACL,R2
          ADD         R2,R0
          RTS
          MOV.L       R0,@R1      ; sum
L11:
          .RES.W      1
          .DATA.L     _sum
```

1.2.7 Extension of return value

If the return value of a function is of type `char`, `unsigned char`, `short`, or `unsigned short`, you can specify whether sign extension/zero extension is to be performed by the called function (`rtnext`) or by the caller (`nortnext`).

By default, the caller performs sign extension/zero extension.

If the function is called more than once, code is likely to be more efficient when extension is performed by the called function, because the extension needs to be coded only once. When extension is performed by the caller, optimization is likely to delete unnecessary extensions. Since the more efficient option depends on the program structure, try both options.

The option specification must be consistent throughout the project.

Format:

`RTnext`
`NORTnext`

Option settings in Renesas IDE:

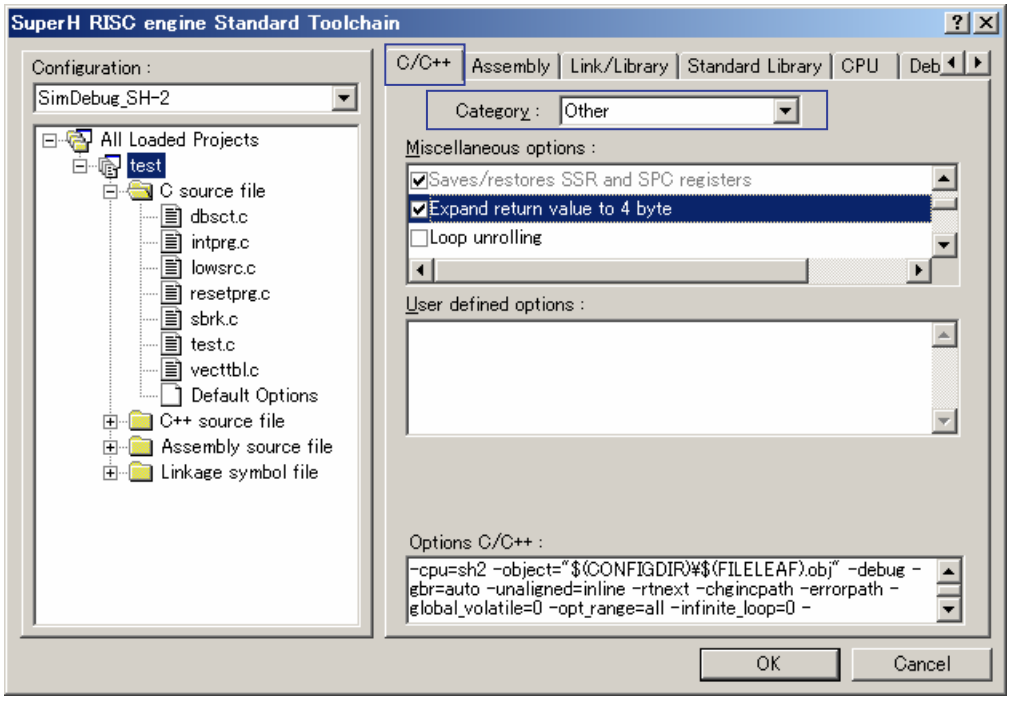


Figure 1-20

Example:

Source code:

```
short x,y;
int   i,j,k;

short f(short a,  short b)
{
    return a * b;
}

void g(void)
{
    i = f(x,y);
    j = f(x,y);
    k = f(x,y);
}
```

Expanded assembly code (nortnext specified)

```
_f:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          STS         MACL,R0
          RTS
          LDS.L       @R15+,MACL
_g:
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV.L       L12,R13     ; _y
          MOV.L       L12+4,R14   ; _x
          MOV.W       @R13,R5     ; y
          BSR         _f
          MOV.W       @R14,R4     ; x
          MOV.L       L12+8,R2    ; _i
          EXTS.W      R0,R1
          MOV.W       @R13,R5     ; y
          MOV.W       @R14,R4     ; x
          BSR         _f
          MOV.L       R1,@R2      ; i
          MOV.L       L12+12,R1   ; _j
          EXTS.W      R0,R7
          MOV.W       @R13,R5     ; y
          MOV.W       @R14,R4     ; x
          BSR         _f
          MOV.L       R7,@R1      ; j
          MOV.L       L12+16,R6   ; _k
          EXTS.W      R0,R2
          MOV.L       R2,@R6      ; k
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTS
          MOV.L       @R15+,R13
L12:
          .DATA.L     _y
          .DATA.L     _x
          .DATA.L     _i
          .DATA.L     _j
          .DATA.L     _k
```

Expanded assembly code (rtnext specified)

```
_f:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          STS         MACL,R2
          EXTS.W      R2,R0
          RTS
          LDS.L       @R15+,MACL
_g:
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV.L       L12,R13     ; _y
          MOV.L       L12+4,R14   ; _x
          MOV.W       @R13,R5     ; y
          BSR         _f
          MOV.W       @R14,R4     ; x
          MOV.L       L12+8,R1    ; _i
          MOV.W       @R13,R5     ; y
          MOV.W       @R14,R4     ; x
          BSR         _f
          MOV.L       R0,@R1      ; i
          MOV.L       L12+12,R2   ; _j
          MOV.W       @R13,R5     ; y
          MOV.W       @R14,R4     ; x
          BSR         _f
          MOV.L       R0,@R2      ; j
          MOV.L       L12+16,R7   ; _k
          MOV.L       R0,@R7      ; k
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTS
          MOV.L       @R15+,R13
L12:
          .DATA.L     _y
          .DATA.L     _x
          .DATA.L     _i
          .DATA.L     _j
          .DATA.L     _k
```

1.2.8 Enumeration data size

You can use the `auto_enum` option to handle enumeration data declared by an enum declaration as the smallest data type that can contain enumerated values.

If the `auto_enum` option is not specified, enumeration data is handled as type `int`. If the `auto_enum` option is specified, the data type changes depending on the range of possible enumerator values. Table 1-5 shows the relationship between the possible enumerator values and data types.

Table 1-5 Possible enumerator values and data types

Enumerator		Data Type
Minimum Value	Maximum Value	
-128	127	signed char
0	255	unsigned char
-32768	32767	signed short
0	65535	unsigned short
Other than the above		int

Use of this option can reduce the size of handled data. The option is especially effective for reducing size when there are many variables and structure members of type `enum`. However, since the number of extensions might increase when this option is specified, specifying this option might reduce the execution speed.

Format:

`AUto_enum`

Option settings in Renesas IDE:

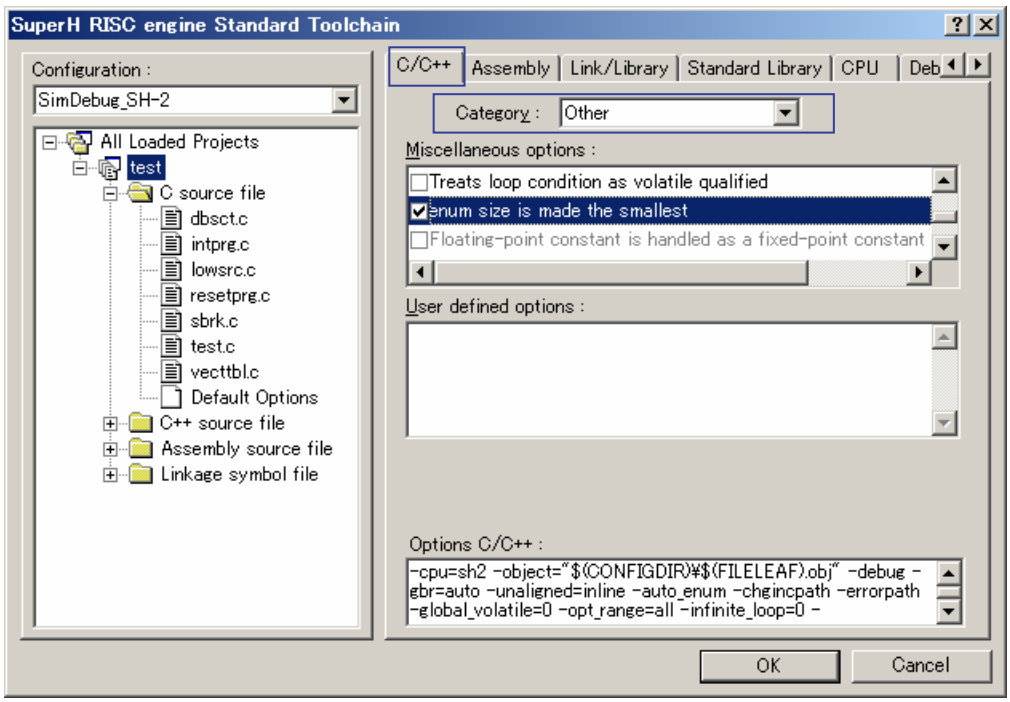


Figure 1-21

Example:

Source code: enum En {A_000 =0,A_001,A_002,A_003,A_END=255}; enum En x[3] = {A_000, A_001, A_END}; Expanded assembly code (auto_enum not specified) _x: ; static: x .DATA.L H'00000000,H'00000001,H'000000FF	Expanded assembly code (auto_enum specified) _x: ; static: x .DATA.B H'00,H'01,H'FF

1.2.9 Switch statement expansion method

You can use the `case` option to select whether to use the if-then method or the table method for evaluation of a `switch` statement. If the if-then method is selected, the target value is compared with each `case` value. If the table method is selected, the data table created with the relative value of each `case` value is referenced for comparison. If there are only a few `case` clauses or the difference between the maximum and minimum `case` values is large, the if-then method might be used regardless of the specification of the `case` option.

When the `case` option is not specified, the compiler automatically selects one or the other of the methods as follows:

(1) If there are only a few `case` labels or the difference between the maximum and minimum `case` values is large, the compiler selects the if-then method.
(2) When (1) does not apply, if the `case` option is specified, the compiler follows the option specification.
(3) When neither (1) nor (2) applies, if the basic option is `speed` and the number of `case` labels is about 10 or more, the compiler selects the table method.

When a specific `case` value matches frequently during program execution, program execution likely to be faster if the relevant `case` value is written first and the if-then method is specified.

For details, see *5. Branching* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler use guide] Efficient programming techniques*.

Format:

CAse = { Ifthen | Table }

Option settings in Renesas IDE:

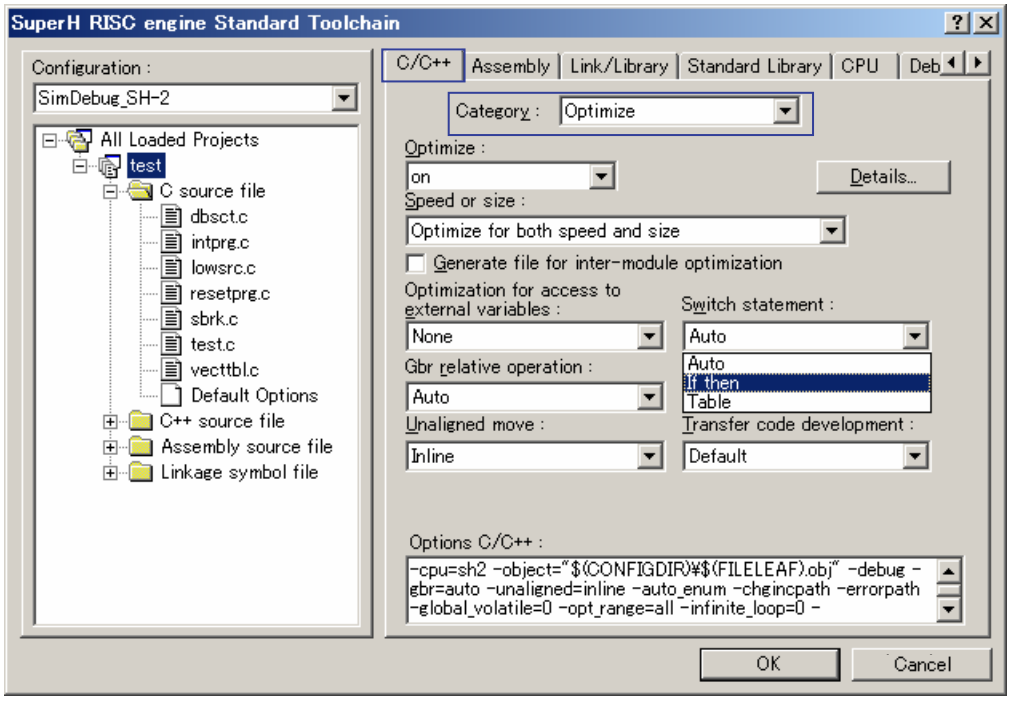


Figure 1-22

2. Useful Options

This chapter explains options that provide benefits that are not related to the improvement of performance.

Table 2-1 List of useful options

No.	Functionality	Option	See section
1	Debugging Information Output Mode	optimize	2.1
2	Pre-processor expansion	preprocessor/noline	2.2
3	External variables handled as volatile	global_volatile	2.3
4	Vacant loop elimination	del_vacant_loop	2.4
5	Elimination of expression preceding infinite loop	infinite_loop	2.5
6	Switches the order of bit assignment	bit_order	2.6
7	Specifies the boundary alignment value for structures, unions, and classes	pack	2.7

2.1 Debugging Information Output Mode

When the `optimize=debug_only` option is specified, you can always view local variable information during debugging. In addition, optimization related to statement-based deletion is suppressed completely. This allows you to set a break point for each statement in the C source code. Note that performance of an object generated with this option specified might be less than the performance of the object generated with `optimize=0` (no optimization) specified. Before you use this option, you should first test it during debugging.

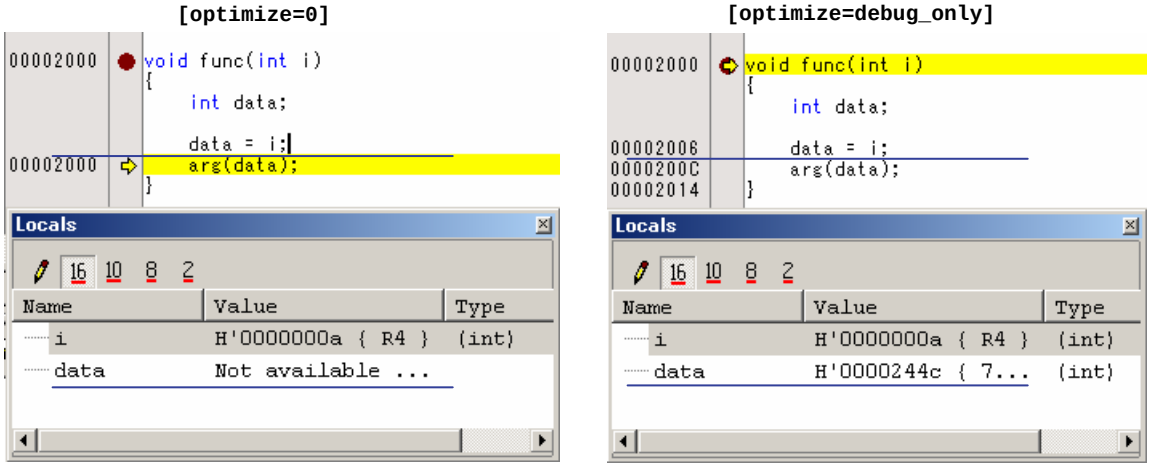


Figure 2-1

Format:

OPtimize = { 0 | 1 | Debug_only }

Option settings in Renesas IDE:

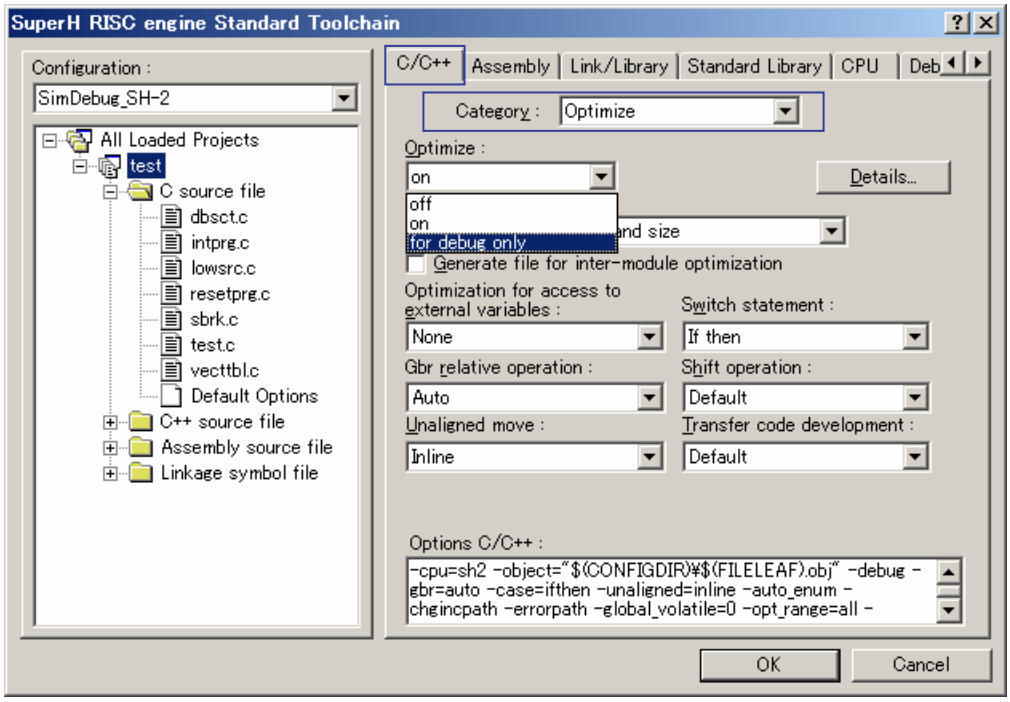


Figure 2-2

2.2 Pre-processor expansion

Outputs a source program processed by the preprocessor. The resultant code in the file replaces the `#include` and `#define` directives in the original code with the corresponding code. Because information such as header files has already been expanded, this file can be compiled without the use of any other files.

If no <file name> is specified, an output file with the same file name as the source file and with a standard extension is created. The standard extension after C compilation is p (if the input source program is written in C), and that after C++ compilation is pp (if the input source program is written in C++).

When `preprocessor` is specified, no object file is output from the compiler.

When `noline` is specified, disables #line output at preprocessor expansion.

Format:

`PREProcessor [=` ***file-name*** **`]`**
`NOLINe`

Option settings in Renesas IDE:

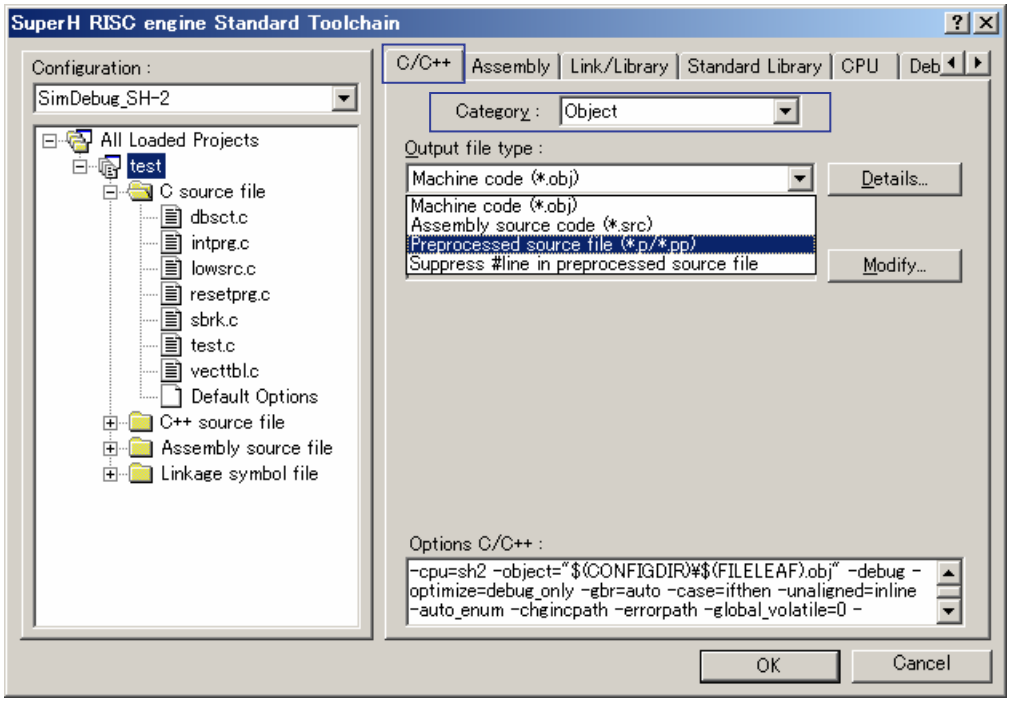


Figure 2-3

Example:

```
Source code:
#define NUM 1
#define MESSAGE(num, name) {num, __DATE__, #name}

struct {
    int   num   ;
    char* date  ;
    char* string;
} data[] = {
        MESSAGE(NUM, aaaa),
};
Preprocessor expansion:
#line 1 "test.c"

struct {
    int   num   ;
    char* date  ;
    char* string;
} data[] = {
        {1, "Jun 13 2007", "aaaa"},
};
```

2.3 External variables handled as volatile

The compiler statically parses C source code and might optimize the access order of a variable and the number of times a variable is accessed if, by doing so, the meaning of the source code does not change. However, if this type of optimization is performed for variables that are used for I/O register access or interrupt processing, the program might not operate as intended. To avoid program misoperation, you must therefore declare these variables as volatile. If a variable has been declared as volatile, optimization will not change the access width of the variable, or access order of the variable, or the number of times the variable is accessed.

Although you need to carefully determine whether a variable should be declared as volatile, checking all variables might be difficult if, for example, a legacy system is reused. For cases such as these, try `global_volatile=1`, which directs the compiler to treat all external variables as volatile.

Format:

```
GLOBAL_Volatile = { 0 | 1 }
```

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

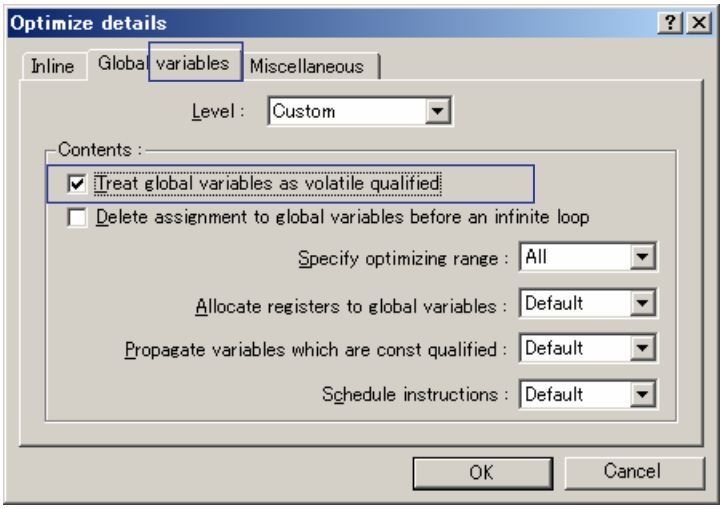


Figure 2-4

Example:

Source code:

```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

Source after optimization (global_volatile=0 specified)

```
int   var;
void func(void)
{
   var = 0;
}
```

Expanded assembly code (global_volatile=0 specified)

```
_func:
          MOV.L       L11,R6      ; _var
          MOV         #0,R2       ; H'00000000
          RTS
          MOV.L       R2,@R6      ; var
L11:
          .DATA.L     _var
```

Source after optimization (global_volatile=1 specified)

```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

Expanded assembly code (global_volatile=1 specified)

```
_func:
          MOV.L       L11,R6      ; _var
          MOV         #1,R1       ; H'00000001
          MOV         #0,R4       ; H'00000000
          MOV.L       R1,@R6      ; var
          RTS
          MOV.L       R4,@R6      ; var
L11:
          .DATA.L     _var
```

2.4 Vacant loop elimination

You can select whether to delete empty loops (loops which contain no processing).

If `del_vacant_loop=0` is specified, the compiler does not delete empty loops. If `del_vacant_loop=1` is specified, the compiler deletes empty loops. The default is `del_vacant_loop=0`.

Note that if you specify `del_vacant_loop=1`, the compiler also deletes necessary empty loops that have been intentionally coded this way. For example, an empty loop might have been coded for timing purposes.

Format:

```
DEL_vacant_loop = { 0 | 1 }
```

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

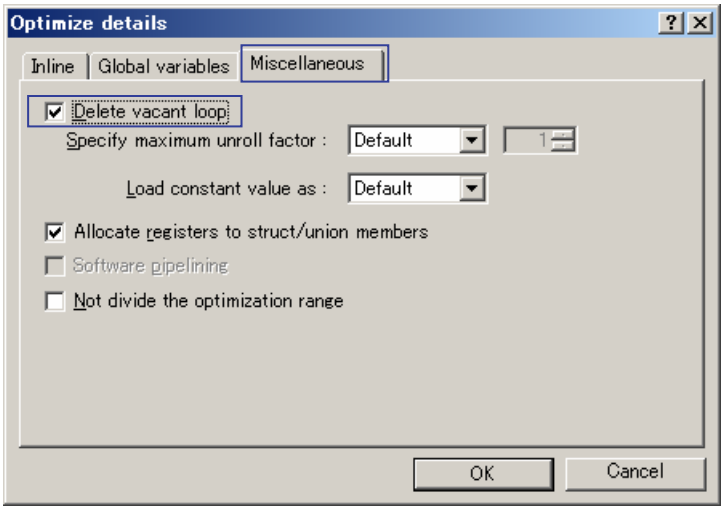


Figure 2-5

2.5 Elimination of expression preceding infinite loop

When an expression that assigns a value to a non-volatile external variable precedes an infinite loop in which the external variable is not referenced, you can delete the expression.

When `infinite_loop=0` is specified, an assignment expression for external variables, which is located immediately before an infinite loop is not eliminated.

When `infinite_loop=1` is specified, an assignment expression that is located immediately before an infinite loop and is for external variables that are not referenced from the infinite loop is eliminated.

The default for this option is `infinite_loop=0`.

Format:

```
INFinite_loop = { 0 | 1 }
```

Option settings in Renesas IDE:

In the SuperH RISC engine Standard Toolchain dialog box, on the **C/C++** tab, select **Optimize** from the **Category** drop-down list, and click **Details** (Figure 1-2). In the displayed dialog box, shown below, specify the settings as follows.

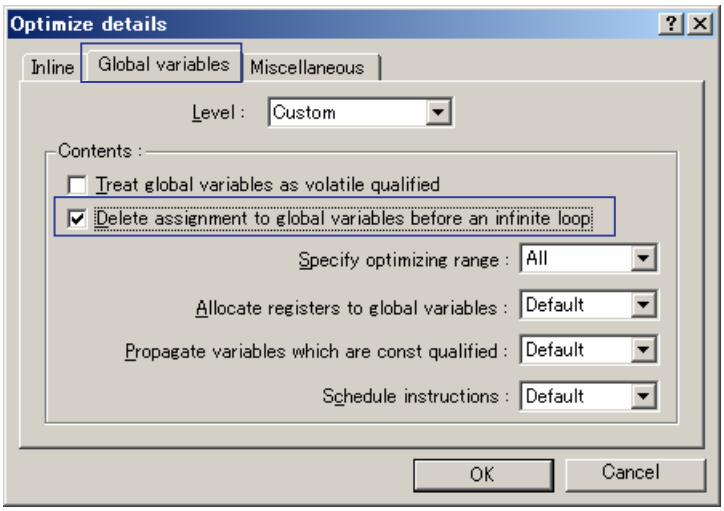


Figure 2-6

Example:

Source code:

```
int a;
void f(void)
{
    a = 1;
    while(1) {
    }
}
```

Source after optimization (infinite_loop=0 specified)

```
int a;
void f(void)
{
    a = 1;
    while(1) {
    }
}
```

Expanded assembly code (infinite_loop=0 specified)

```
_f:
          MOV.L       L13+2,R6    ; _a
          MOV         #1,R2       ; H'00000001
          MOV.L       R2,@R6      ; a
L11:
          BRA         L11
          NOP
L13:
          .RES.W      1
          .DATA.L     _a
```

Source after optimization (infinite_loop=1 specified)

```
int a;
void f(void)
{
    while(1) {
    }
}
```

Expanded assembly code (infinite_loop=1 specified)

```
_f:
L10:
          BRA         L10
          NOP
```

2.6 Switches the order of bit assignment

Specifies the order of bit field members. Since the bit field member allocation rule might differ depending on the microcomputer, you can use this functionality to improve portability of programs between different microcomputers.

When `bit_order=left` is specified, members are allocated from the upper bit.

When `bit_order=right` is specified, members are allocated from the lower bit.

You can also use the `#pragma bit_order` directive to specify the bit field order. If you specify both the `bit_order` option and the `#pragm a bit_order` directive, the `#pragma bit_order` directive takes precedence.

For details about the functionality of this option, see *2.2 Switches the order of bit fields* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler Use guide] Extended Specifications*.

Format:

```
BIt_order = { Left | Right }
```

Option settings in Renesas IDE:

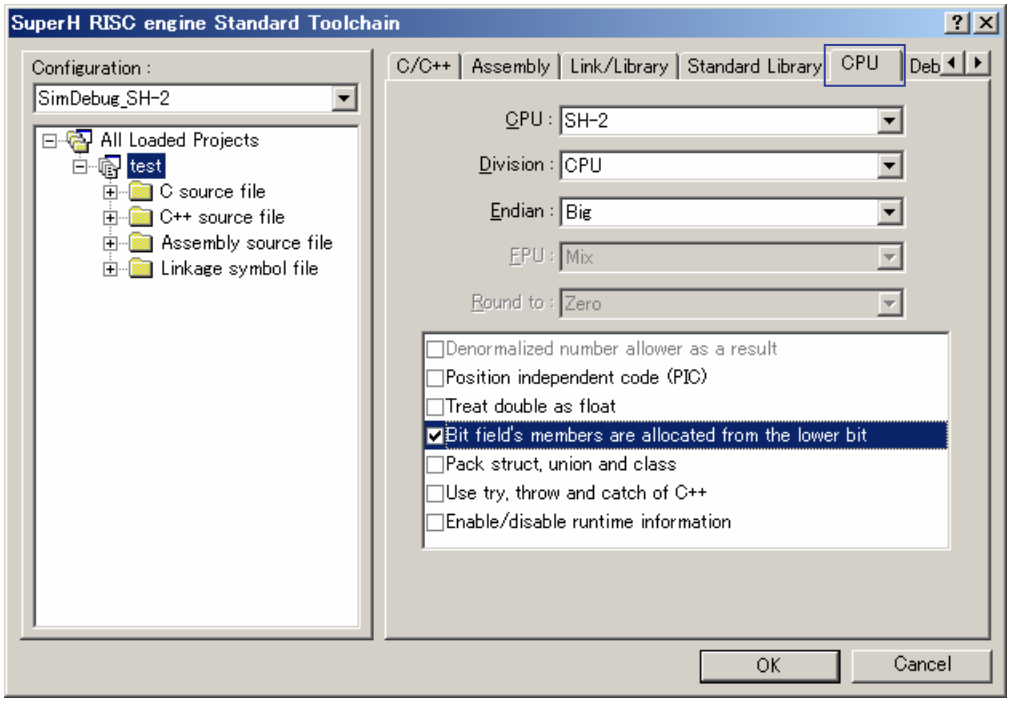


Figure 2-7

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

In some types of programs, such as communication programs, you might not want structures to have padding bits. This is also true for unions and classes. In these cases, you can specify the `pack=1` option to align structure members on a 1-bit boundary. Structures aligned on a 1-bit boundary do not include a padding area.

You can also use the `#pragma pack` directive to specify the alignment for structures. If you specify both the `pack` option and the `#pragma pack` directive, the `#pragma pack` directive takes precedence.

For details about the functionality of this option, see *2.3 Specifies the boundary alignment value for structures, unions, and classes* in the manual *SuperH RISC engine C/C++ Compiler Package APPLICATION NOTE: [Compiler Use guide] Extended Specifications*.

Format:

```
PACK = { 1 | 4 }
```

Option settings in Renesas IDE:

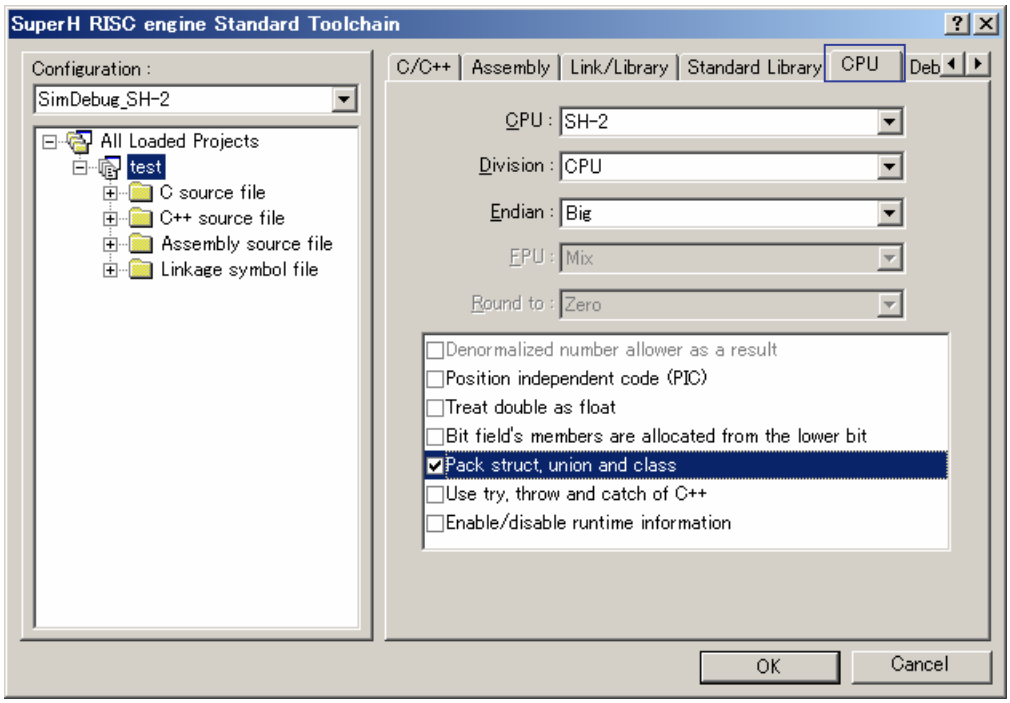


Figure 2-8

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

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.