# RX Family C/C++ Compiler Package

## Application Notes: Compiler Usage Guide, Option Edition

This document explains the options for C/C++ Compiler V.1 for the RX family.

Table of contents

## 1. Optimization options

The optimization options for compilers include two basic options that optimize on speed or size, and detailed options that set further details for each optimization item. 1.1 explains the basic options and related detailed options, and 1.2 explains detailed options that can increase performance in particular.

Note that the expansion code in assembly code as used in this document can be obtained by specifying "output=src" and "cpu=rx600".
When the "cpu" option is different, the expansion code in assembly language may also differ. Also, the expansion code in assembly language may change due to subsequent compiler improvements, so use this as a reference.

### 1.1 Basic options (optimizing on speed/size)

The compiler optimizations include those that reduce object size, and those that speed up execution.
To optimize on execution speed, specify the "speed" option. To optimize on size, specify the "size" option (default). The following optimizations are performed when each option is specified:

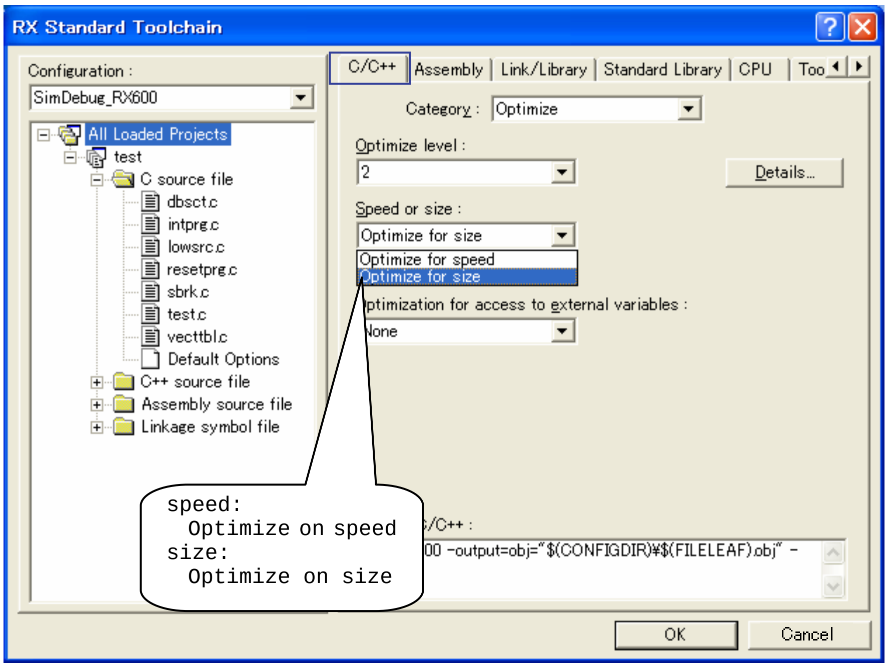
- When "speed" is specified
  Optimizations effective for size and execution speed, and optimizations that increase execution speed but may increase size
- When "size" is specified
  Optimizations effective for size and execution speed, and optimizations that decrease size but may decrease execution speed

Ideally, functions that demand execution speed and functions that demand size can be split into separate files, with size optimizations and speed optimizations selected for each file.

Format

| | |
|---|---|
| **speed** | |
| **size** | **: Default** |

[How to specify this option in High-Performance Embedded Workshop (herein as the Renesas IDE)]



RX Standard Toolchain

Configuration : SimDebug_RX600

Category : Optimize

Optimize level : 2

Speed or size : Optimize for size

ptimization for access to external variables : None

00 -output=obj="$(CONFIGDIR)¥$(FILELEAF).obj" -

**Figure1-1**

Note 1:

The execution speed on production machines may change depending on causes unrelated to code generated by the compiler, such as memory architecture and interrupts. Therefore, specifying “speed” does not necessarily result in the fastest code. Confirm the effects of the various options presented in this document on the production machines.

Note 2:

The detailed options for setting details for the items optimized by the compiler include items for which the default values change depending on how the basic options are set. Table 1-1 shows the options for which the default values change.

Table 1-1 List of defaults

| No | Function | | Size | | | Speed | | | Reference |
|---|---|---|---|---|---|---|---|---|---|
| | | optimize | max | 2 | 0 or 1 | max | 2 | 0 or 1 | |
| 1 | Automatic inline expansion | | inline=0 | noinline | noinline | inline=500 | inline=100 | noinline | 1.1.1 |
| 2 | Loop expansion optimization | | loop=1 | loop=1 | loop=1 | loop=32 | loop=2 | loop=1 | 1.1.2 |

The following explains each option.

## 1.1.1 Automatic inline expansion

An option can be used to specify whether automatic inline expansion is performed for functions.

When the “inline” option is specified, automatic inline expansion is performed. “inline=<*number*>” can be used to specify that inline expansion is to be performed until the program size grows by a given percentage. For example, if “inline=50” is specified, inline expansion is performed until the program size grows by 50% (reaching 1.5 times its original size). If ”inline=0” is specified, inline expansion is performed.

Automatic inline expansion prioritizes called functions by the smallest first.

Note that functions for which #pragma inline is specified are expanded inline regardless of the specification for automatic inline expansion. However, keep in mind that the size increase from inline expansion due to #pragma inline is included when the maximum size for automatic inline expansion is determined.

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

Keep in mind that inline expansion is not performed for functions if one of the following conditions is satisfied

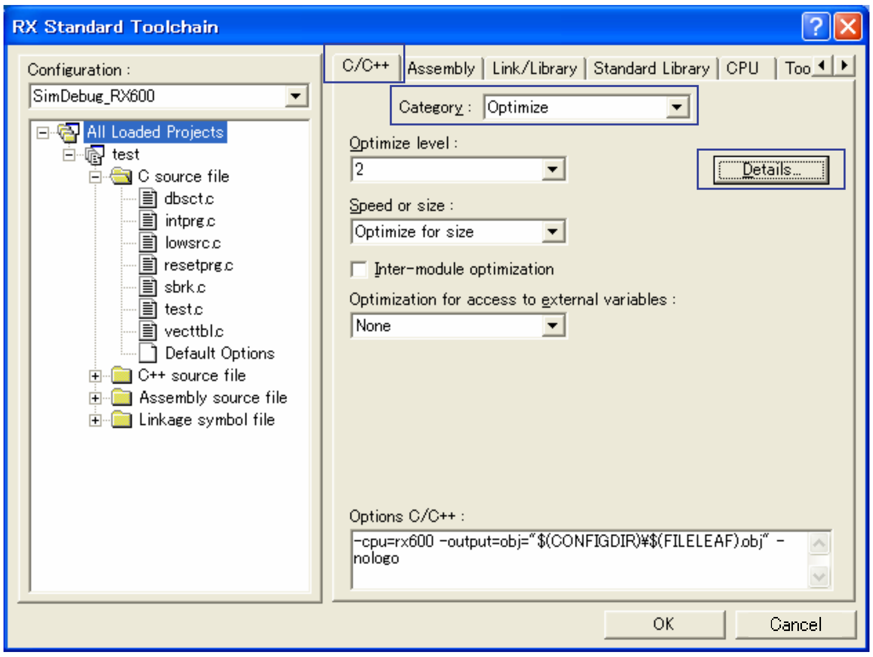
(1) The function has variable parameters.
(2) The function is called using the address of the expanded function.

For details about inline expansion, see *1.1 Specifying inline function expansion* in *C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition*.

Format

| | |
|---|---|
| **inline [= <*number*>]** | **: Default when speed is specified, with a default number of 100** |
| **noinline** | **: Default when size is specified** |

[How to specify this option in the Renesas IDE]



RX Standard Toolchain

Configuration : SimDebug_RX600

All Loaded Projects
- test
  - C source file
    - dbsct.c
    - intprg.c
    - lowsrc.c
    - resetprg.c
    - sbrk.c
    - test.c
    - vecttbl.c
    - Default Options
  - C++ source file
  - Assembly source file
  - Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU | Too

Category : Optimize

Optimize level : 2

Speed or size : Optimize for size

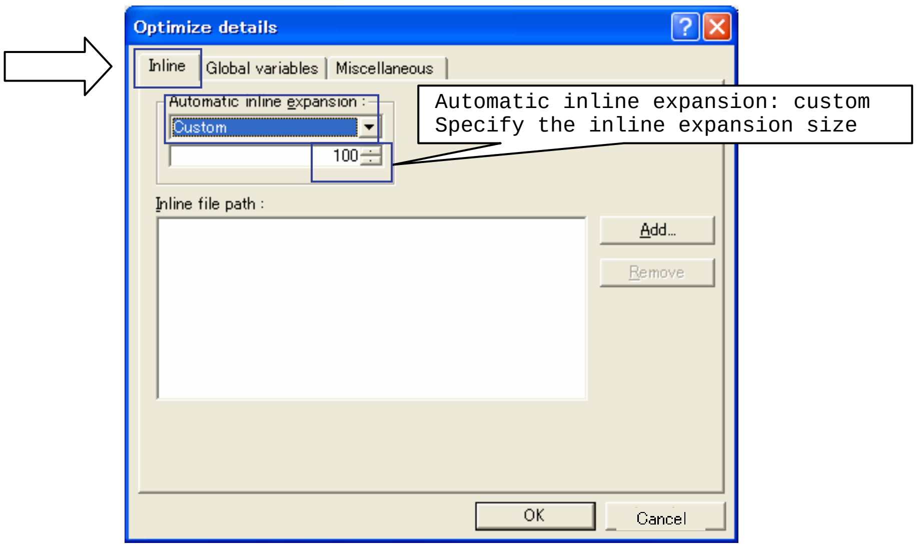
[ ] Inter-module optimization

Optimization for access to external variables : None

Options C/C++ :

```
-cpu=rx600 -output=obj="$(CONFIGDIR)¥$(FILELEAF).obj" -
nologo
```

**Figure1-2**



Optimize details

Inline | Global variables | Miscellaneous

Automatic inline expansion : Custom

100

Inline file path :

```
Automatic inline expansion: custom
Specify the inline expansion size
```

**Figure1-3**

Inline expansion requires that the definitions of functions expanded at compile time be able to be referenced. Therefore, the function without the function definition that can be referred from the file to be compiled cannot be singled out for normal inline expansion. To progress the function that there is a definition in another file, specify the inter-file inline expansion (file_inline=<*file-name*>[,...]) option. Note that operation is not guaranteed if an extern function of the same name is defined in multiple files specified inline, across the files (a single chosen function definition is used).

Note that if the same file as the source file is specified for the inline expansion files, the following warning is output by the compiler.

C1315 (W) File_inline "file name" ignored by same file as source file

[How to specify this option in the Renesas IDE]

Select the "Compiler" tab, select "Optimization" for Category, choose Details (Figure1-2), and then perform the following settings in the displayed dialog box.

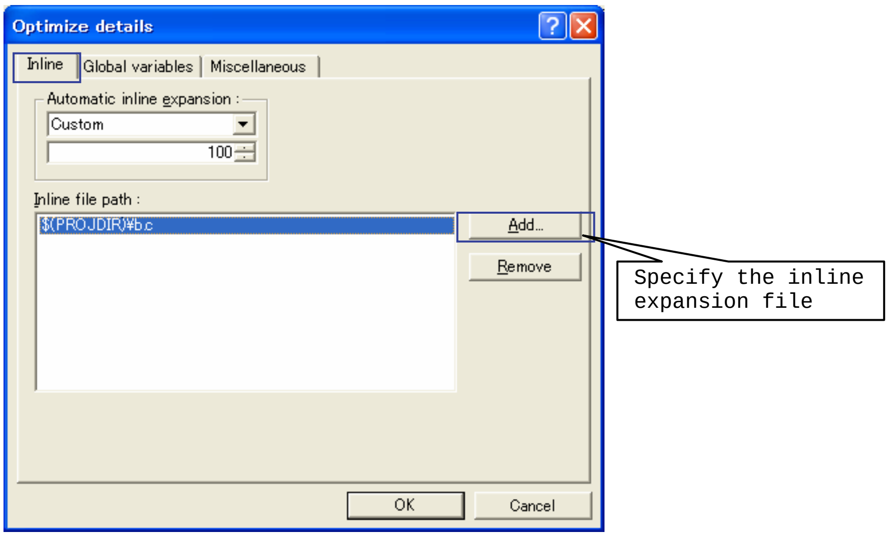


**Figure1-4**

Example:

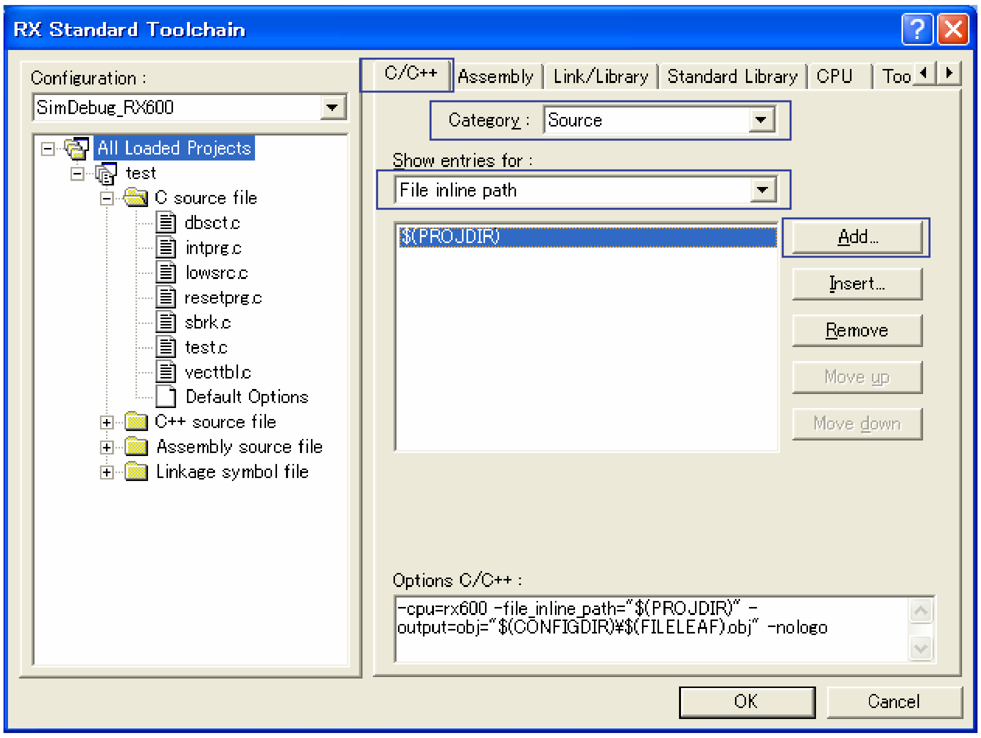
```
Source code
<a.c>
void func(void)
{
   g();
}
<b.c>
#pragma inline (g)
void g(void)
{
   h();
}

a.c expansion when file_inline=b.c is specified
void func(void)
{
   h();
}
```

When an inter-file inline expansion file outside the current folder is specified, the inter-file inline expansion folder can be specified (file_inline_path=<*path-name*>[,...]) so that the path name for inter-file inline expansion files can be omitted.

Files subject to inter-file inline expansion are first searched for in the folder specified by the "file_inline_path" option, and then in the current folder.

[How to specify this option in the Renesas IDE]



Category : Source

Show entries for : File inline path

$(PROJDIR)

Options C/C++ :

```
-cpu=rx600 -file_inline_path="$(PROJDIR)" -
output=obj="$(CONFIGDIR)¥$(FILELEAF).obj" -nologo
```

**Figure1-5**

### 1.1.2 Loop expansion optimization

An option can be used to specify whether loop expansion is performed.
When the "loop" option is specified, loop expansion is performed.
The maximum expansion count during loop expansion can be specified using "loop=<*number-from-1-to-32*>".

Format

**loop[=<number>]** **: Default when speed is specified is 2**
**: Default when size is specified is 1**

[How to specify this option in the Renesas IDE]

To specify the maximum expansion count during loop expansion, select the "Compiler" tab, select "Optimization" for Category, choose Details (Figure1-2), and then perform the following settings in the displayed dialog box.

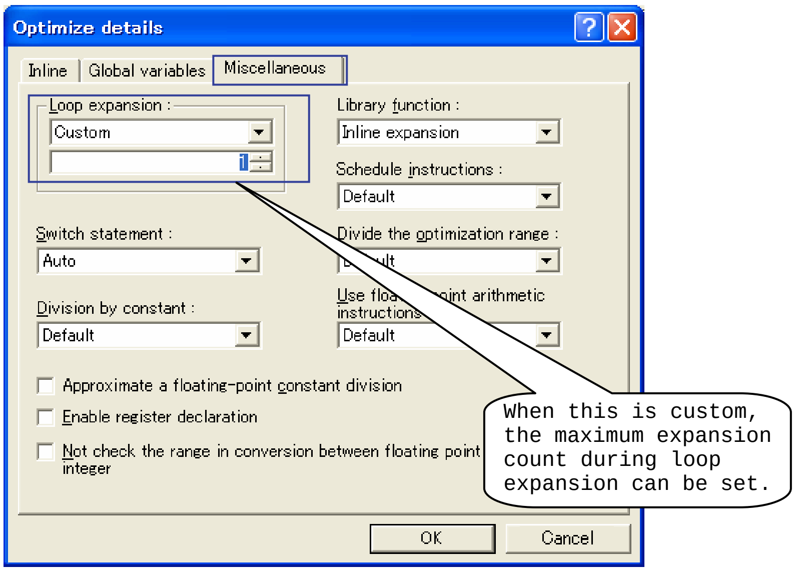

Optimize details
Inline
Global variables
Miscellaneous
Loop expansion :
Custom
1
Library function :
Inline expansion
Schedule instructions :
Default
Switch statement :
Auto
Divide the optimization range :
Default
Division by constant :
Default
Use floating-point arithmetic instructions
Default
Approximate a floating-point constant division
Enable register declaration
Not check the range in conversion between floating point integer
When this is custom, the maximum expansion count during loop expansion can be set.
OK
Cancel

**Figure1-6**

## 1.2 Detailed options to improve performance

This chapter explains the options that can improve performance in particular, of the detailed operations used to set details for each optimization item.

Table 1-2 Detailed options to improve performance

| No | Functionality | Option | Size | Execution speed | Reference |
|---|---|---|---|---|---|
| 1 | External variable access optimization | map/smap | Improves significantly | Improves significantly | 1.2.1 |
| 2 | Optimization range splitting | scope/noscope | May improve or worsen | May improve or worsen | 1.2.2 |
| 3 | Enumeration type size | auto_enum | Improves | Worsens | 1.2.3 |
| 4 | Switch statement expansion | case | May improve or worsen | May improve or worsen | 1.2.4 |

### 1.2.1 External variable access optimization

When the “map” option and “smap” option are used, access to external variables is performed relatively from the base external variable. This removes the need for load processing for external variable addresses, which improves execution speed. Also, since address value load processing can be omitted, the program size can be reduced. External variable access optimization can improve both speed and size.

The “map” optimizations are performed using the external symbol allocation information generated by the optimization linkage editor. Therefore, compilation needs to be performed twice. (In the HEW environment, it is executed automatically.)

The “smap” optimizations perform external variable optimization only for external variables defined in compiled files. Since the external symbol allocation information generated by the linkage editor is not used, optimization can be performed with only one compilation.

The “map” optimizations are more effective at optimizing than the “smap” optimization, but since they require compilation twice, they increase the build time. “smap” optimizations are limited to external variables defined in files, but are performed in a single compilation, and can be performed when files such as library files are generated without address resolution.

Table 1-3 Characteristics of map and smap

| Option | Build time | External symbol allocation information file generated by the optimization linkage editor | Optimization effect | Address resolution |
|---|---|---|---|---|
| map | Long | Required | Better than smap | Required |
| smap | Short | Not required | Worse than map | Not required |

Format

- Inter-module specification

  **map [= <*file-name*>]**

  Usage differs depending on the output option.

  [For output=abs or mot]
  Specify only the “map” option.

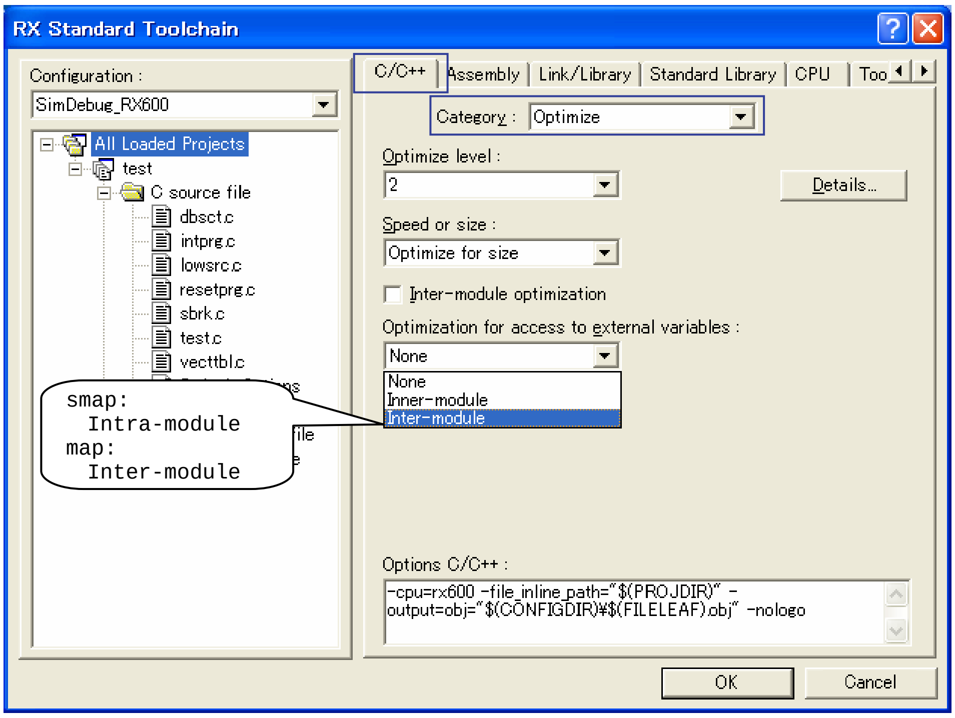
  [For output=obj or src]
  Perform compilation once without specifying the “map” option, specify “map=<*file-name*>” during linkage, and create an external symbol allocation information file. When compiling the second time, specify the external symbol allocation information file (map=<*file-name*>).
  When the definition order of external variables or static variables changes, the external symbol allocation information file will need to be regenerated.
  Operation when a second compilation is performed is not guaranteed under the following conditions:
  – When the specified option is anything other than the option specified for the first compilation and the “map” option
  – When a source file different than that specified for the first compilation is specified

- Intra-module specification

  **smap**

[How to specify this option in the Renesas IDE]

When the setting for external variable access optimization is changed from [Inter-module] to anything else, or to [Inter-module] from anything else, a warning is displayed. This is because output of external symbol allocation information files by the linkage editor is automatically enabled/disabled.



RX Standard Toolchain

Configuration : SimDebug_RX600

All Loaded Projects / test / C source file: dbsct.c, intprg.c, lowsrc.c, resetprg.c, sbrk.c, test.c, vecttbl.c

C/C++ | Assembly | Link/Library | Standard Library | CPU | Too

Category : Optimize

Optimize level : 2

Speed or size : Optimize for size

Inter-module optimization: unchecked

Optimization for access to external variables : None (options: None, Inner-module, Inter-module — Inter-module selected)

Options C/C++ :

```
-cpu=rx600 -file_inline_path="$(PROJDIR)" -
output=obj="$(CONFIGDIR)¥$(FILELEAF).obj" -nologo
```

**Figure1-7**

Example 1:

The following shows how to perform relative access of consecutively allocated variables from the same register when the variable allocation order is considered.

Source code

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

Assembly expansion code (without map/smap specified): Code size of 19 bytes

```
_f:                              ; function: f
          .STACK     _f=4
L10:
          MOV.L      #00000000H,R5
          MOV.L      #_a,R4
          MOV.L      R5,[R4]
          MOV.L      #_b,R4
          MOV.L      R5,[R4]
          RTS
          .SECTION    B,DATA,ALIGN=4
          .glb       _a
_a:                              ; static: a
          .blkl      1
          .glb       _b
_b:                              ; static: b
          .blkl      1
```

Assembly expansion code (with map/smap specified): Code size of 18 bytes

```
_f:                              ; function: f
          .STACK     _f=4
L10:
          MOV.L      #00000000H,R5
          MOV.L      #_a,R4
          MOV.L      R5,[R4]
          MOV.L      R5,04H[R4]
          RTS
          .SECTION    B,DATA,ALIGN=4
          .glb       _a
_a:                              ; static: a
          .blkl      1
          .glb       _b
_b:                              ; static: b
          .blkl      1
```

## 1.2.2 Optimization range splitting

An option can be used to specify whether the function optimization range is split into multiple ranges and compiled or compiled without splitting.

When “scope” is specified, large functions may be split into optimization ranges and then compiled.

When “noscope” is specified and no optimization range splitting is performed, since optimization can be performed across the whole function, the size and execution speed will usually both improve. However, if the registers are not sufficient, performance may actually degrade.

Since this option may cause program code efficiency to either improve or worsen, try both options when tuning performance.

Keep in mind that when compilation is performed without splitting the optimization range, compilation will take longer.

Format

| | |
|---|---|
| **scope** | **: Default when optimize=0\|1\|2 is specified** |
| **noscope** | **: Default when optimize=max is specified** |

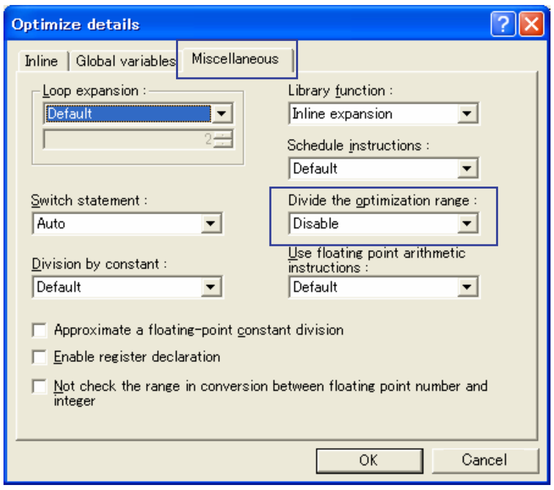
Whether the optimization range has been split can be checked using information level messages, which can be enabled by specifying the “message” option.

If the optimization range is split, the following message is displayed.

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

[How to specify this option in the Renesas IDE]

Select the “Compiler” tab, select “Optimization” for Category, choose Details (Figure1-2), and then perform the following settings in the displayed dialog box.



Loop expansion : Default
Library function : Inline expansion
Schedule instructions : Default
Switch statement : Auto
Divide the optimization range : Disable
Division by constant : Default
Use floating point arithmetic instructions : Default
☐ Approximate a floating-point constant division
☐ Enable register declaration
☐ Not check the range in conversion between floating point number and integer

**Figure1-8**

### 1.2.3 Enumeration type size

Enumeration type data declared using enum is handled using the minimum type in which the enumeration value will fit.

If the “auto_enum” option is unspecified, the enumeration type size is processed as the signed int type. If the “auto_enum” option is specified, the type handled changed based on the values that can be assigned to the enumerator. Table 1-4 shows the values and types that can be assigned to the enumerator.

Table 1-4 Relationship of values and types that can be assigned for enumeration types

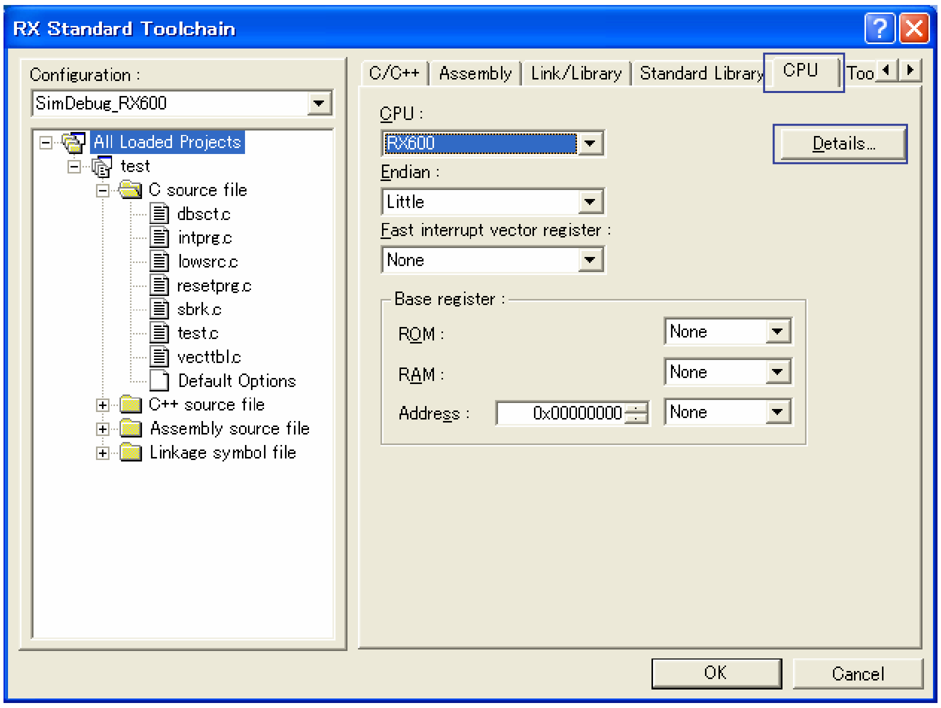
| Enumerator | | Type |
|---|---|---|
| Minimum value | Maximum value | |
| -128 | 127 | signed char |
| 0 | 255 | unsigned char |
| -32768 | 32767 | signed short |
| 0 | 65535 | unsigned short |
| None of the above | | signed int |

Because of the reduced size of data, size performance is improved. This is especially effective when used for many enum type variables or structure members.

Format

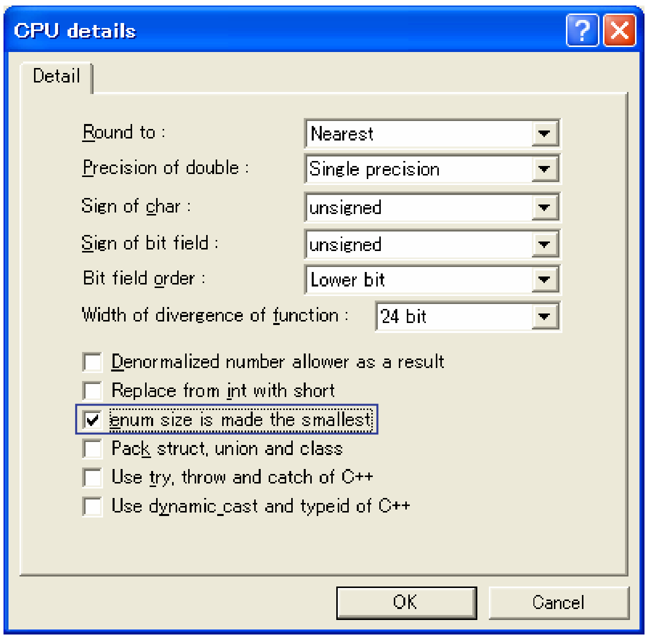
```
auto_enum
```

[How to specify this option in the Renesas IDE]



CPU : RX600
Endian : Little
Fast interrupt vector register : None
Base register :
ROM : None
RAM : None
Address : 0x00000000 None

**Figure1-9**



Round to : Nearest

Precision of double : Single precision

Sign of char : unsigned

Sign of bit field : unsigned

Bit field order : Lower bit

Width of divergence of function : 24 bit

- ☐ Denormalized number allower as a result
- ☐ Replace from int with short
- [✓ enum size is made the smallest]
- ☐ Pack struct, union and class
- ☐ Use try, throw and catch of C++
- ☐ Use dynamic_cast and typeid of C++

**Figure1-10**

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};

Assembly expansion code (when auto_enum is
unspecified)
_x:                             ; static: x
          .lword      00000000H, 00000001H,
000000FFH
```

```
Assembly expansion code (when auto_enum is
specified)
_x:                             ; static: x
          .byte       00H, 01H,0FFH
```

### 1.2.4 Switch statement expansion method

Processing to evaluate switch statements can be performed by selecting the “if-then method”, which performs case value comparison, or the “Table method”, which references a data table created from relative values of each case value. Sometimes, such as when there are few cases or the case value maximum and minimum are far apart, the “if-then method” is used even if “Table” method is selected to the “case” optional.

If the “case” option is not specified, the compiler will automatically select one of the following expansion methods:

(1) If there are few cases or the case value maximum and minimum are far apart, the “if-then method” is used.

(2) If (1) does not apply and the “case” option is specified, and “case” option specification is used.

(3) If (1) and (2) do not apply, the “speed” option is specified, and around 10 or more case labels exist, the “Table method” is used.

During program execution, if a certain case value occurs often, execution speed is best when the corresponding case is coded first and the “if-then method” is specified. When execution does not skip to a given case value, execution speed is best when the “Table method” is specified.

Format

**case = { ifthen | table | auto}**

[How to specify this option in the Renesas IDE]

Select the “Compiler” tab, select “Optimization” for Category, choose Details (Figure1-2), and then perform the following settings in the displayed dialog box.

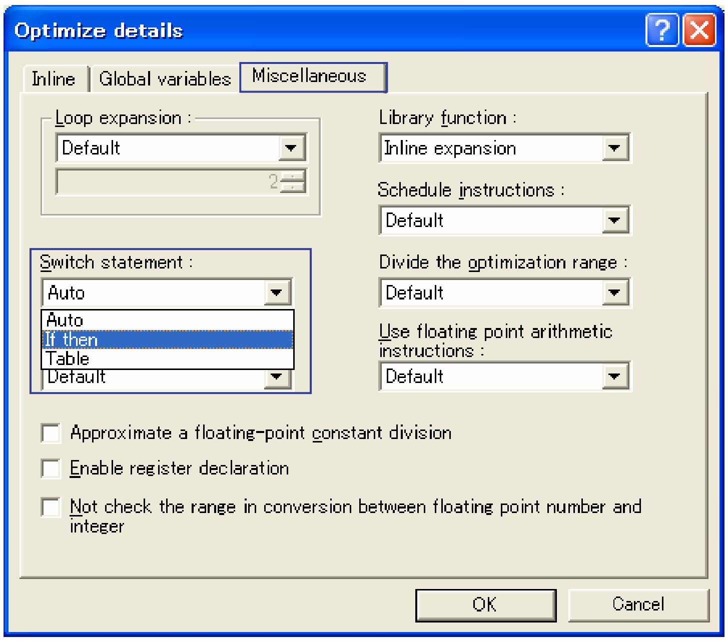

Optimize details
Inline
Global variables
Miscellaneous
Loop expansion :
Default
Library function :
Inline expansion
Schedule instructions :
Default
Switch statement :
Auto
Auto
If then
Table
Default
Divide the optimization range :
Default
Use floating point arithmetic instructions :
Default
Approximate a floating-point constant division
Enable register declaration
Not check the range in conversion between floating point number and integer
OK
Cancel

**Figure1-11**

## 2. Convenient options

This chapter explains convenient options other than those to improve performance.

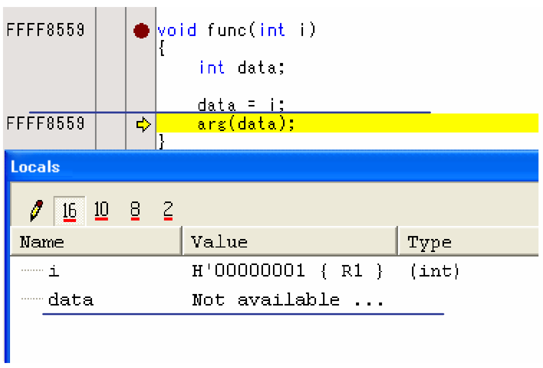
Table 2-1 List of convenient options

| No | Functionality | Option | Reference |
|---|---|---|---|
| 1 | Debugging information output mode | optimize | 2.1 |
| 2 | Pre-processor expansion | preprocessor/noline | 2.2 |
| 3 | External variable volatilization | volatile | 2.3 |
| 4 | Specifying bit field order | bit_order | 2.4 |
| 5 | Specifying structure, shared structure, and class alignment count | pack | 2.5 |

## 2.1 Debugging information output mode

When the "optimize=0" option is specified, information about local variables is always available to be referenced during debugging. Also, since optimizations related to per-statement deletion is completely suppressed, break points can be set for all each statements in the C source code. When this option is used to generate objects, object performance may suffer. We recommend using this temporarily during debugging.

**[optimize=2]**



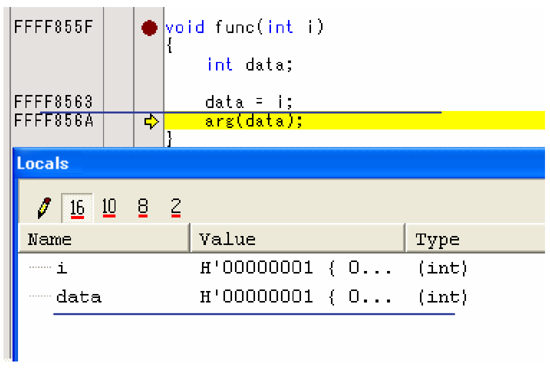
```
FFFF8559  ● void func(int i)
            {
                int data;

                data = i;
FFFF8559  ⇨     arg(data);
            }
```

Locals

```
Name          Value               Type
 i            H'00000001 { R1 }   (int)
 data         Not available ...
```

**[optimize=0]**



```
FFFF855F  ● void func(int i)
            {
                int data;

FFFF8563        data = i;
FFFF856A  ⇨     arg(data);
            }
```

Locals

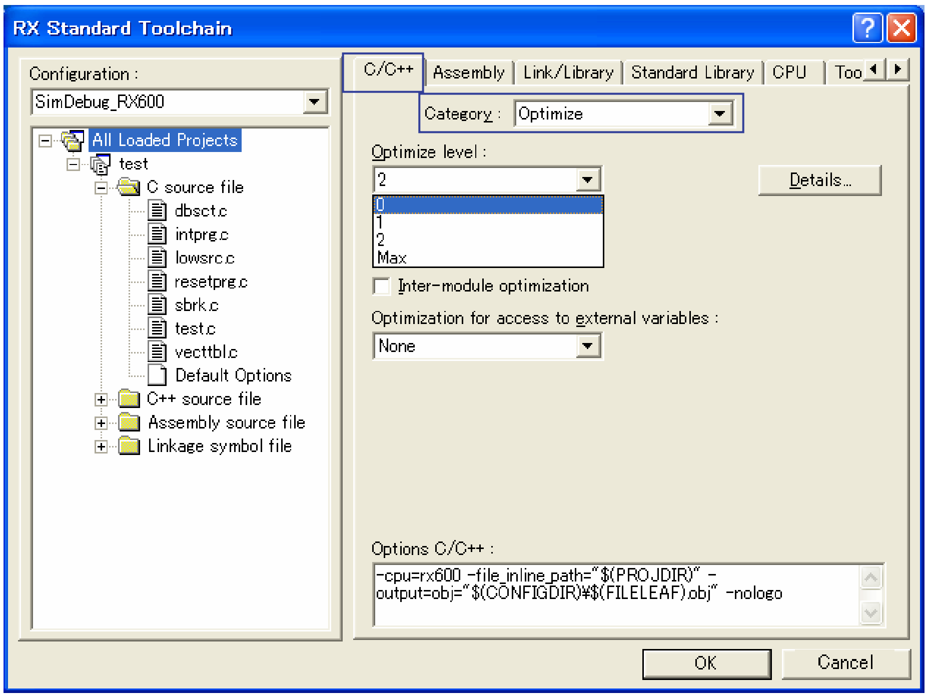
```
Name          Value               Type
 i            H'00000001 { 0...   (int)
 data         H'00000001 { 0...   (int)
```

**Figure2-1**

Format

**optimize = { 0 | 1 | 2 | max }**

[How to specify this option in the Renesas IDE]



Configuration : SimDebug_RX600

Category : Optimize

Optimize level : 2

Inter-module optimization

Optimization for access to external variables : None

Options C/C++ :

```
-cpu=rx600 -file_inline_path="$(PROJDIR)" -
output=obj="$(CONFIGDIR)¥$(FILELEAF).obj" -nologo
```

**Figure2-2**

## 2.2 Pre-processor expansion

This outputs the source program after pre-processor expansion. The code after pre-processor expansion is that after #include and #define are replaced. Since header file and other information has already been expanded in this file, it can be compiled on its own.

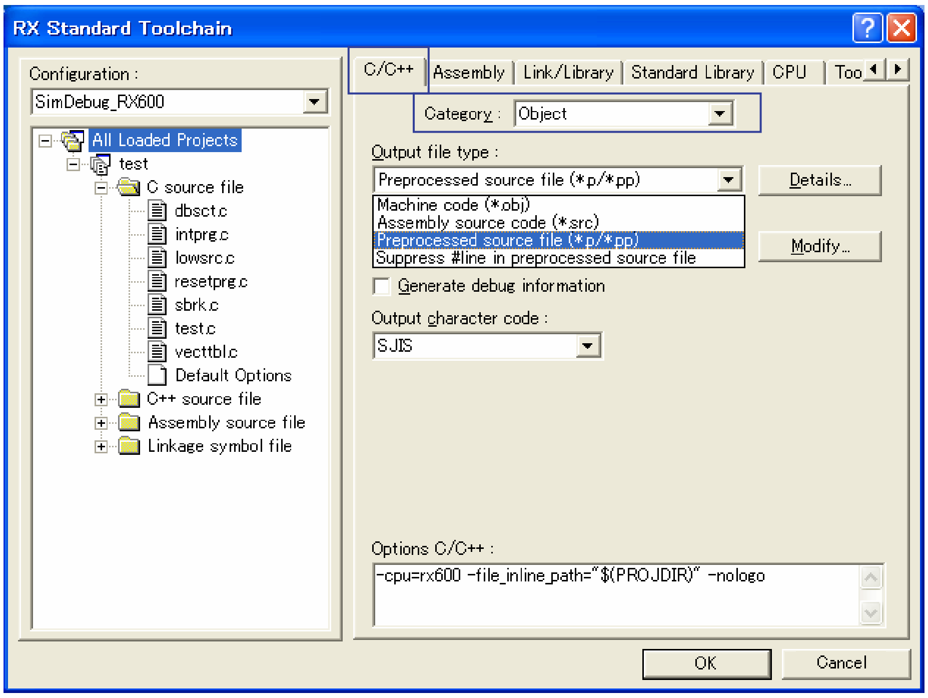
If <*file-name*> is not specified, a file with the extension p (when the input source file is a C program) or pp (when the input source program is a C++ program) is created, with the same name as the source file.

When “noline” is specified, #line output is suppressed during pre-processor expansion.

Format

```
output=prep [= <file-name>]
noline
```

[How to specify this option in the Renesas IDE]


RX Standard Toolchain

Configuration : SimDebug_RX600

All Loaded Projects
- test
  - C source file
    - dbsct.c
    - intprg.c
    - lowsrc.c
    - resetprg.c
    - sbrk.c
    - test.c
    - vecttbl.c
    - Default Options
  - C++ source file
  - Assembly source file
  - Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU | Too

Category : Object

Output file type : Preprocessed source file (*p/*pp)

- Machine code (*.obj)
- Assembly source code (*.src)
- Preprocessed source file (*p/*pp)
- Suppress #line in preprocessed source file

Details... Modify...

☐ Generate debug information

Output character code : SJIS

Options C/C++ :
-cpu=rx600 -file_inline_path="$(PROJDIR)" -nologo

OK Cancel

**Figure2-3**

Example:

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

struct {
   int   num   ;
   char* date  ;
   char* string;
} data[] = {
      MESSAGE(NUM, aaaa),
};
Pre-processor expansion
#line 1 "C:¥¥Workspace_Evaluation_RX¥¥test1¥¥test1¥¥a.c"



struct {
   int   num   ;
   char* date  ;
   char* string;
} data[] = {
      {1, "Jul 22 2009", "aaaa"},
};
```

## 2.3 External variable volatilization

When C source code is statically analyzed during compiler optimization processing, variable access order and count may be optimized as long as the result is not changed. However, when this optimization is performed on variables used in access to the I/O register or interrupt processing, operation may not be performed as intended. In this case, volatile declaration needs to be performed for such variables. When volatile declaration is performed, the access order and access count is performed as specified in the C source code.

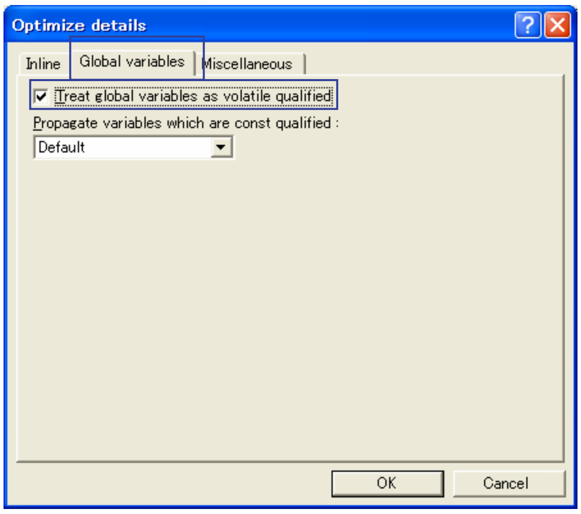
The volatile declaration should be specified for variables deemed necessary, but sometimes it is difficult to check individual variables due to the misuse of past assets. In this case, try specifying "volatile". "volatile" can be specified to treat all external variables as having the volatile declaration.

Format

```
volatile
novolatile
```

[How to specify this option in the Renesas IDE]

Select the "Compiler" tab, select "Optimization" for Category, choose Details (Figure1-2), and then perform the following settings in the displayed dialog box.



[✓] Treat global variables as volatile qualified

Propagate variables which are const qualified : Default

**Figure2-4**

Example:

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

Optimization (when novolatile is specified)
```
int   var;
void func(void)
{
   var = 0;
}
```

Assembly expansion code (when novolatile is specified)
```
_func:                            ; function: func
          .STACK     _func=4
L10:
          MOV.L      #_var,R4
          MOV.L      #00000000H,[R4]
          RTS
          .SECTION    B,DATA,ALIGN=4
          .glb       _var
_var:                             ; static: var
          .blkl      1
```

Optimization (when volatile is specified)
```
int   var;
void func(void)
{
   var = 1;
   var = 0;
}
```

Assembly expansion code (when volatile is specified)
```
_func:                            ; function: func
          .STACK     _func=4
L10:
          MOV.L      #_var,R4
          MOV.L      #00000001H,[R4]
          MOV.L      #00000000H,[R4]
          RTS
          .SECTION    B,DATA,ALIGN=4
          .glb       _var
_var:                             ; static: var
          .blkl      1
```

## 2.4 Specifying bit field order

The sort order for bit fields can be changed. The ordering rules for bit fields may differ depending on the microcomputer. This functionality can be used to increase the portability for programs running on other microcomputers.
If “bit_order=left” is specified, members are allocated from the higher-order bits.
If “bit_order=right” is specified, members are allocated from the lower-order bits.

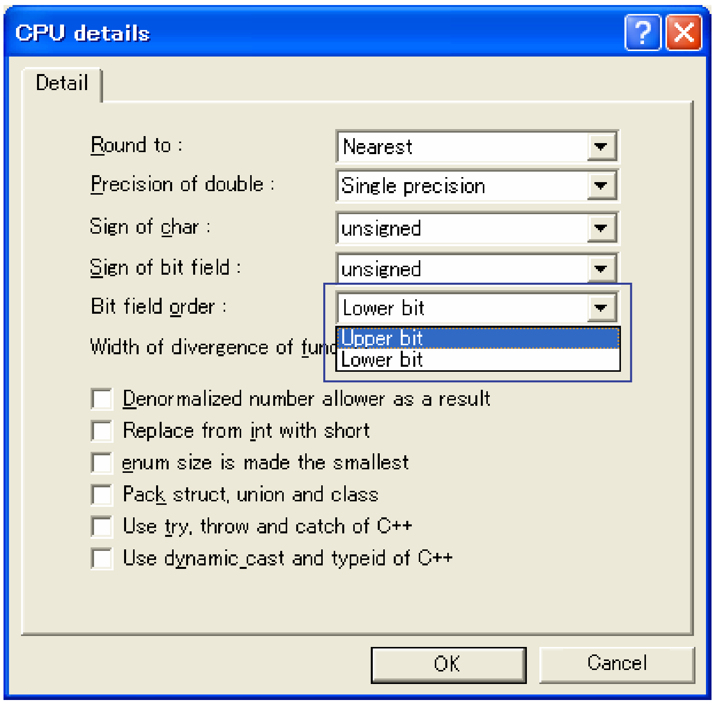
The bit field order can be specified even when #pragma bit_order is specified. When both the option and #pragma are specified at the same time, the #pragma specification takes precedence.
For details about this functionality, see *2.2 Specifying bit field order* in *C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition.*

Format

```
bit_order = { left | right }
```

[How to specify this option in the Renesas IDE]

Select the “CPU” tab, choose Details (Figure1-9), and then perform the following settings in the displayed dialog box.


CPU details

Detail

Round to : Nearest
Precision of double : Single precision
Sign of char : unsigned
Sign of bit field : unsigned
Bit field order : Lower bit
Upper bit
Lower bit
Width of divergence of func

- [ ] Denormalized number allower as a result
- [ ] Replace from int with short
- [ ] enum size is made the smallest
- [ ] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Use dynamic_cast and typeid of C++

OK Cancel

**Figure2-5**

## 2.5 Specifying structure, shared structure, and class alignment count

Sometimes, the creation of free space needs to be avoided in structures (including shared structures or classes) used for communication programs. In such cases, "pack" can be specified as an option to set the alignment count of structure members to 1. This prevents free space from being created in structures with an alignment count of 1.

#pragma pack can also be used to set the structure alignment count to 1. When both this option and #pragma are specified at the same time, the #pragma specification takes precedence.

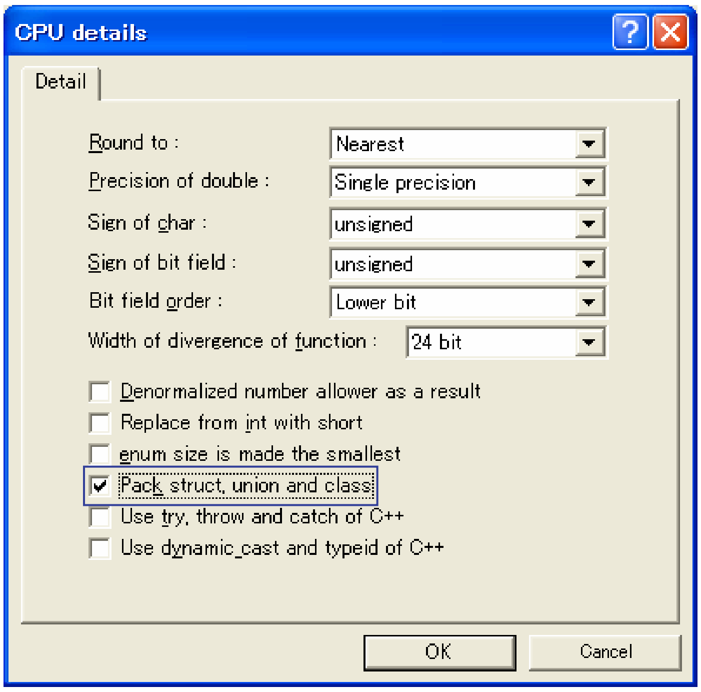
For details about this functionality, see *2.3 Specifying structure, shared structure, and class alignment count C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition.*

Format

```
pack
unpack
```

[How to specify this option in the Renesas IDE]

Select the " CPU " tab, choose Details (Figure1-9), and then perform the following settings in the displayed dialog box.


CPU details

Detail

| | |
|---|---|
| Round to : | Nearest |
| Precision of double : | Single precision |
| Sign of char : | unsigned |
| Sign of bit field : | unsigned |
| Bit field order : | Lower bit |
| Width of divergence of function : | 24 bit |

- [ ] Denormalized number allower as a result
- [ ] Replace from int with short
- [ ] enum size is made the smallest
- [✓] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Use dynamic_cast and typeid of C++

**Figure2-6**

## Web site and support <website and support>

Web site for Renesas Technology

http://japan.renesas.com/

Contact information

http://japan.renesas.com/inquiry

csc@renesas.com

## Revision history<revision history,rh>

| Rev. | Date issued | Contents changed | |
|---|---|---|---|
| | | Page | Details |
| 1.00 | 2009.10.1 | -- | Initial edition |
| | | | |
| | | | |
| | | | |
| | | | |