RX Family C/C++ Compiler Package

Application Notes: [Compiler Use Guide], #pragma Extension Guide

This document explains extended functionality (using #pragma) and convenient #pragma usage that can improve size efficiency or execution speed for C/C++ Compiler V.1 for the RX family.

Table of contents

1. Extended functionality for improving execution speed

This chapter explains extended functionality (using #pragma) that can improve size efficiency or execution speed.
Table 1-1 lists examples of such functionality.

Table 1-1 Extended functionality for improving performance

No	#pragma	Functionality	Size	Execution speed	Reference
1	#pragma inline	Specifying inline function expansion	Worsens	Improves significantly	1.1
2	#pragma inline_asm	Performing inline expansion	Worsens	Improves	1.2

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.

1.1 Specifying inline function expansion

Inline expansion is optimization processing that expands called programs at function calls, to reduce the overhead of function calls and improve speed. This can be especially effective when functions called in loops are expanded, because the call count is decreased significantly. Note that since the compiler performs optimizations for the code after inline expansion is performed, when large functions are expanded inline the program size may increase, reducing the efficacy of compiler optimizations. Inline expansion is most effective when specified for small, frequently called functions.

Format

#pragma inline [(]<*function-name*>[,···][)]

Specify #pragma inline before the definition of the function body.
External definitions are also generated for functions specified by #pragma inline. If no external definition is needed, specify static for the function declaration. When a function for which static is specified is expanded inline, the corresponding function itself is not generated, which can reduce size.
When the inline option is specified, automatic inline expansion is performed for the specified size. Even when the maximum size for automatic inline expansion is exceeded, functions for which #pragma inline is specified are expanded inline.
Keep in mind that inline expansion is not performed for functions if one of the following conditions is satisfied, even when #pragma inline is specified:
- The function is defined before the #pragma inline specification.
- The function has variable parameters.
- The function is called by address.

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

    func_p = func;

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

The function address is replaced with a function type pointer. When the function type pointer is used to call the function, the function is not expanded inline even if inline expansion is specified for the function.

Example: Inline expansion

```
Source code

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

Expanded source code

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

1.2 Performing inline expansion in assembly code functions

Inline expansion functionality embedded in the assembler is convenient for using CPU commands not supported by the C language, and for improving performance by coding in assembly language instead of the C language. When "#pragma inline_asm" is used to declare an assembly code function in a C source file, assembly code can be run in the function. This is called an inline assembly function.

Format

#pragma inline_asm [(]<*function-name*> [,⋯][)]

Keep the following in mind when using inline assembly functions:

- Use a temporary label for the function label.
  (temporary label: ‘ ?: ’)
- Do not use the RTS (return) command at the end of a definition.
- Back up and restore guaranteed registers.

If a compilation error occurs for a function specified by #pragma inline_asm, when compiler debugging information is output, since the line information in the C source is output, the line information is displayed in the C source for the Renesas IDE. Accordingly, the displayed line information cannot be used to jump to the line in the assembly program that is causing the error. When output from the compiler debugging information stops, the assembly program line is output for the line information displayed in the Renesas IDE. To debug functions specified by #pragma inline_asm, we recommend suppressing specification for compiler debugging information.

For the interface between functions, follow the generation rules for C/C++ compilers (Table 1-2 and Table 1-3).

Table 1-2 General rules for allocating arguments in C language programs

Allocation rule		
Argument passed in the register		Argument passed in the stack
Register for argument storage	Target types	
Any one item from R1 to R4	signed char, (unsigned) char, bool, (signed) short, unsigned short, (signed) int, unsigned int, (signed) long, unsigned long, float, double[#1], long double[#1], pointers, pointers to data members, and references	• Anything for which the argument type is not passed by register • Functions declared using the original type of the function, with variable arguments[#3] • Of R1 to R4, when those not yet allocated to other arguments are fewer than those to be allocated
Any two items from R1 to R4	(signed) long long, unsigned long long, double[#2], and long double[#2]	
The size for R1 to R4, divided by 4	Structures, shared structures, and class types whose size is a multiple of 4	

Note #1 When dbl_size=8 is not specified.
#2 When dbl_size=8 is specified.
#3 When the original type of the function is used to declare a function with variable arguments, the arguments without corresponding types in the declaration, and the arguments immediately before, are passed in the stack. Arguments without type are converted to the long type if they are integers of 2 bytes or less, or to the double type if they are the float type, and are all handled as arguments with boundary adjustment counts of 4.

Table 1-3 Return values and type setting locations for C language programs

Return value type	Setting location
signed char, (unsigned) char, (signed) short, unsigned short, (signed) int, unsigned int, (signed) long, unsigned long, float, double[#2], long double[#2], pointers, bool, references, pointers to data members	R1 signed char and (signed) short: sign extension (unsigned) char and unsigned short: set in result of zero extension
double[#3], long double[#3], (signed) long long, unsigned long long	R1, R2 Bottom 4 bytes in R1 and top 4 bytes in R2
Structures, shared structures, and class types within 16 bytes, and whose size is a multiple of 4	R1, R2, R3, and R4 are set in order from the start of the memory image, 4 bytes each
Structures, shared structures, and class types other than those above	Return value setting area (memory)[#1]

Notes #1 When the function return value is set in memory, it is set in an area indicating the return value address. The caller reserves a return value setting area in addition to the argument area, and once that address is set in R15, the function is called.
#2 When dbl_size=8 is not set.
#3 When dbl_size=8 is specified.

Example 1: Assembly code in an inline assembly function

```
Source code

/* Inline function definition */
/* FILE: inlasm.h */
#pragma inline_asm(rev4b)
static unsigned long rev4b(unsigned long p)
/* The function is declared using static */
{
  ; Comments in definitions use assembler semicolons (;)
    REVL R1,R1
  ; RTS commands are not used at the end of definition
}
#pragma inline_asm(ovf)
static unsigned long ovf(void)
{
?:; A temporary label is used within the inline assembly function
  ; Registers that need to be secured are backed up and restored before and after function call
    PUSH.L R6
    MOV.L R1,R6
  ;    :
    CMP #1,R6
    BEQ.W ?-
    POP R6
}
```

2. Other convenient extended functionality

This chapter explains convenient extended functionality other than that for improving performance. Table 2-1 lists examples of such functionality.

Table 2-1 Convenient extended functionality

No	#pragma	Functionality	Reference
1	#pragma section	Specifying section changes	2.1
2	#pragma bit_order	Specifying bit field order	2.2
3	#pragma pack #pragma unpack #pragma packoption	Alignment control to structure member	2.3

2.1 Specifying section changes

The section names output by the compiler can be switched.

For example, sometimes allocation should be performed to separate addresses, such as when certain modules need to be allocated to external RAM, while others are allocated to internal RAM. A name is given to each split section, and then the address to be placed for each section is specified in the linkage editor.

When no section name is specified for #pragma section, the default section name is applied subsequently.

Format

#pragma section [<*section-type*>][△<*changed-section-name*>]
<*section-type*>:{ P | C | D | B | W }

Table 2-2 and Table 2-3 show the changed section names.

Table 2-2 Section changing functionality and section names (without section type)

	Target area	Specification method	After change
1	Program area	#pragma section <XX>	P<XX>
2	Constant area		C<XX>
3	Initialized data area		D<XX>
4	Uninitialized data area		B<XX>
5	Table area of switch sentence branch		W<XX>

Table 2-3 Section change functionality and section name (with section type)

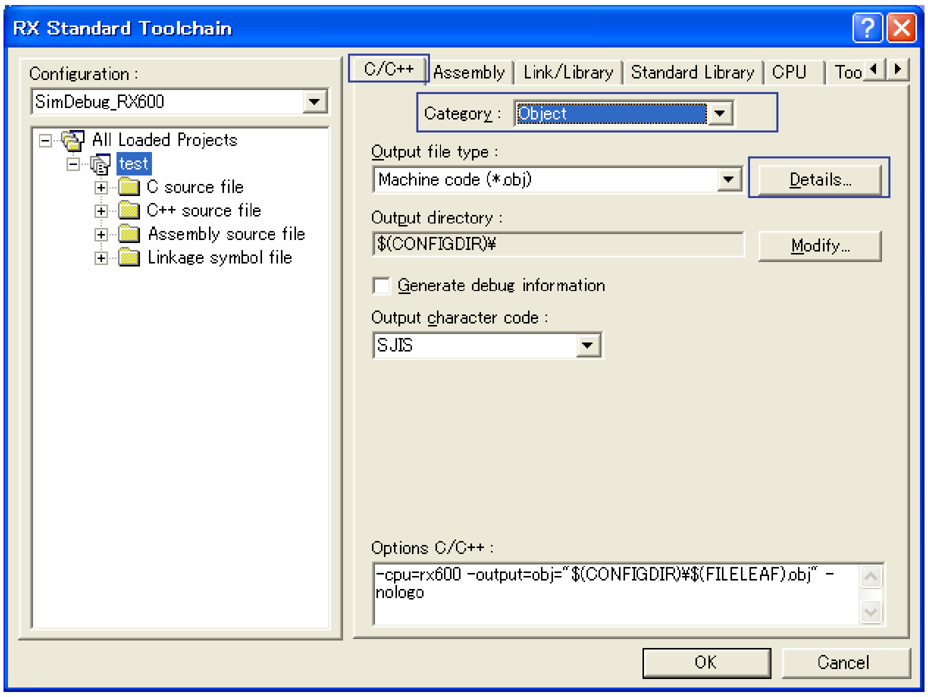
	Target area	Specification method	Section type <X>	After change
1	Program area	#pragma section <X> <XX>	P	<XX>
2	Constant area		C	<XX>
3	Initialized data area		D	<XX>
4	Uninitialized data area		B	<XX>
5	Table area of switch sentence branch		W	<XX>

The “section” option can be used to change the default section name.

Format for the “section” option

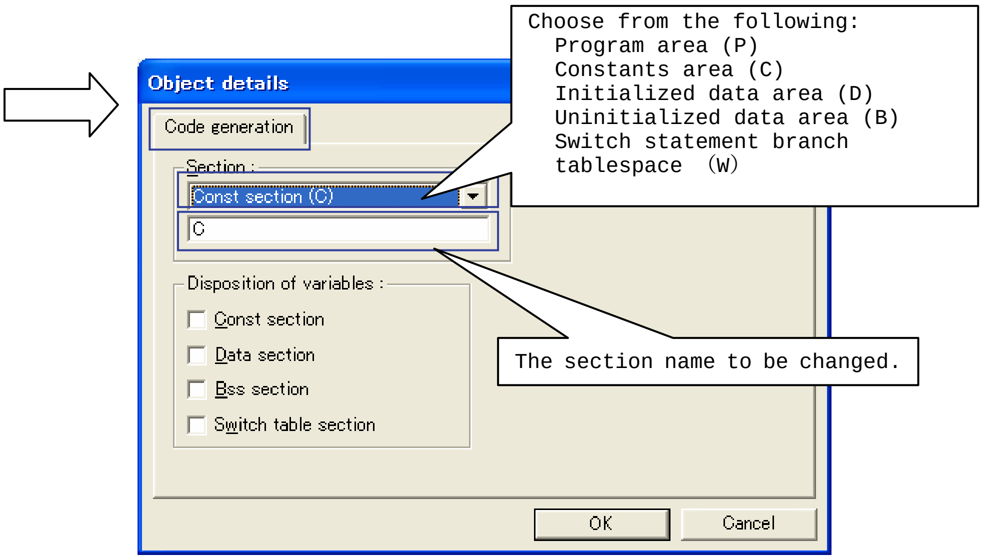
```
        section = <sub>[,…]
           <sub>:   {   P   =   <section-name>        |
                        C   =   <section-name>        |
                        D   =   <section-name>        |
                        B   =   <section-name>        |
                        W   =   <section-name>        }
```

[How to specify this option in the Renesas IDE]



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

Figure 2-1



Choose from the following:
Program area (P)
Constants area (C)
Initialized data area (D)
Uninitialized data area (B)
Switch statement branch tablespace (W)

The section name to be changed.

Figure 2-2

Example 1:

```
Source code

#pragma section abc

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

#pragma section

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

Example 2:

```
Source code

#pragma section P ABC
#pragma section B DEF

int a;         /* a is allocated to section DEF */
const int c=1; /* c is allocated to section C */
void f(void)   /* f is allocated to section ABC */
{
    a=c;
}

#pragma section

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

Example 3:

When section=P=PX,C=CX,B=BX is specified

```
Source code

#pragma section abc

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

#pragma section

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

2.2 Specifying bit field order

#pragma bit_order can be specified to change the bit field order. 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.

The bit field order can also be specified as an option for each file. When both the option and #pragma are specified at the same time, the #pragma specification takes precedence.

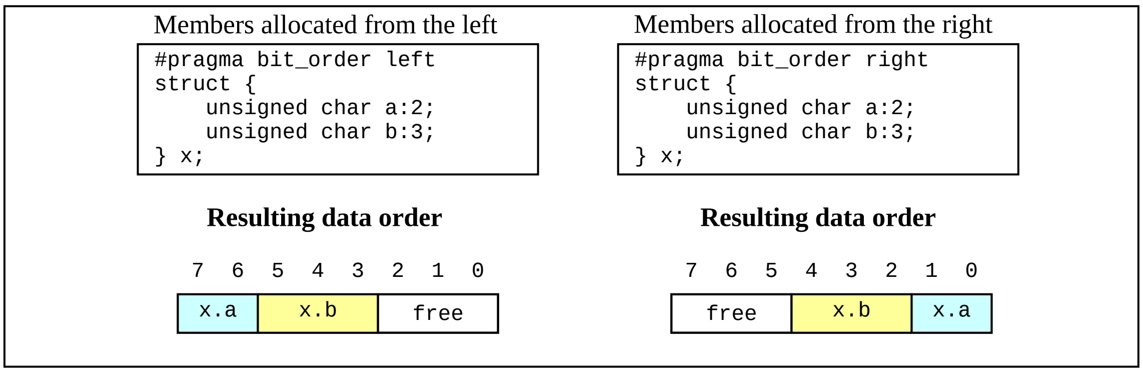
Format

#pragma bit_order [{left|right}]

If left is specified, each member is allocated from the higher-order bits, and if right is specified, from the lower-order bits. The default setting is allocation from the lower-order bits. If both left|right are omitted from the #pragma bit_order specification, the subsequent lines will use the value of the option.

Example 1:

The following shows allocation for data sort scenarios with #pragma bit_order left and #pragma bit_order right.



Members allocated from the left

```
#pragma bit_order left
struct {
    unsigned char a:2;
    unsigned char b:3;
} x;
```

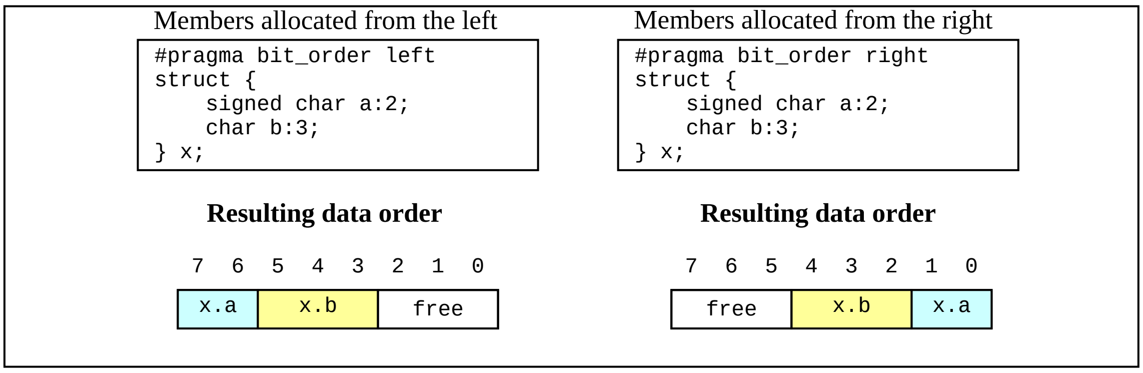
Members allocated from the right

```
#pragma bit_order right
struct {
    unsigned char a:2;
    unsigned char b:3;
} x;
```

Figure 2-3

Example 2:

Consecutive type specifiers of the same size are packed into the same area as much as possible.



Members allocated from the left

```
#pragma bit_order left
struct {
    signed char a:2;
    char b:3;
} x;
```

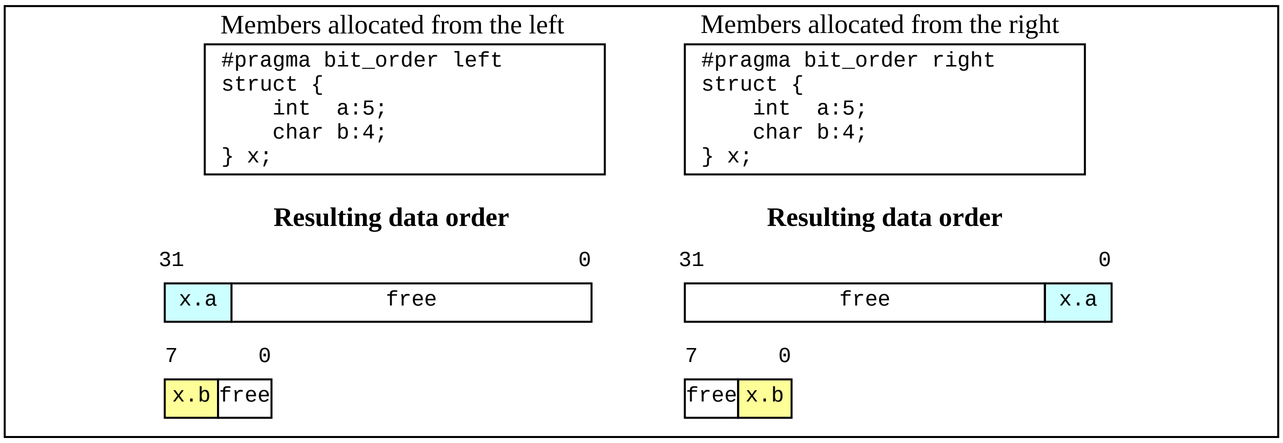
Members allocated from the right

```
#pragma bit_order right
struct {
    signed char a:2;
    char b:3;
} x;
```

Figure 2-4

Example 3:

Consecutive type specifiers of different sizes are allocated in the following areas.



Members allocated from the left

```
#pragma bit_order left
struct {
    int  a:5;
    char b:4;
} x;
```

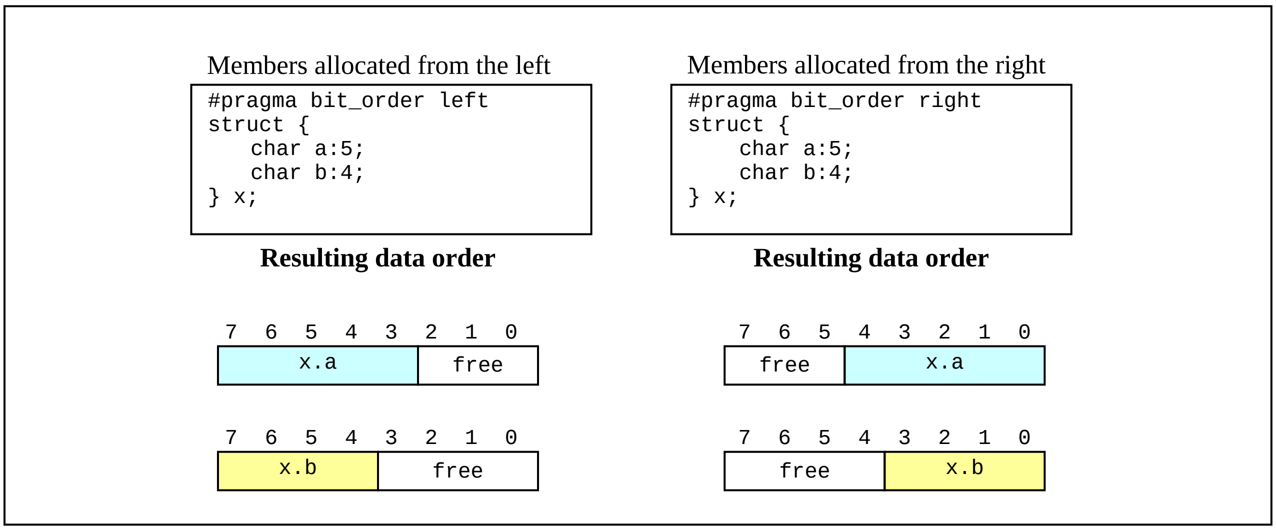
Members allocated from the right

```
#pragma bit_order right
struct {
    int  a:5;
    char b:4;
} x;
```

Figure 2-5

Example 4:

If the remaining bits in the area into which consecutive type specifiers of the same size are to be packed are smaller than the bit field size, the remaining area is unused, and the areas are allocated as follows.



Members allocated from the left

```
#pragma bit_order left
struct {
   char a:5;
   char b:4;
} x;
```

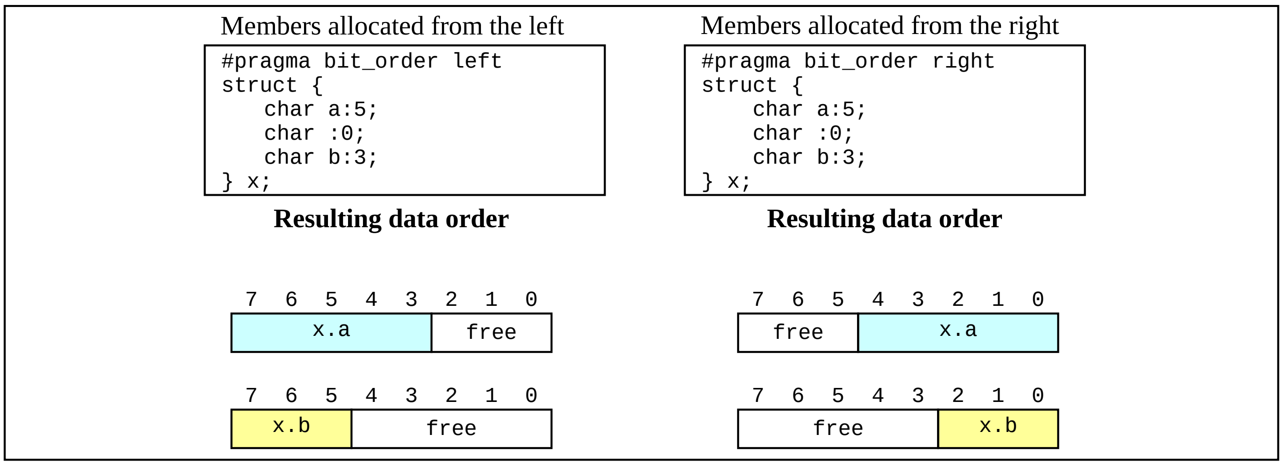
Members allocated from the right

```
#pragma bit_order right
struct {
    char a:5;
    char b:4;
} x;
```

Figure 2-6

Example 5:

When a member of a bit field with bit width 0 is specified, the subsequent members are forcibly allocated to the following areas.



Members allocated from the left

```
#pragma bit_order left
struct {
    char a:5;
    char :0;
    char b:3;
} x;
```

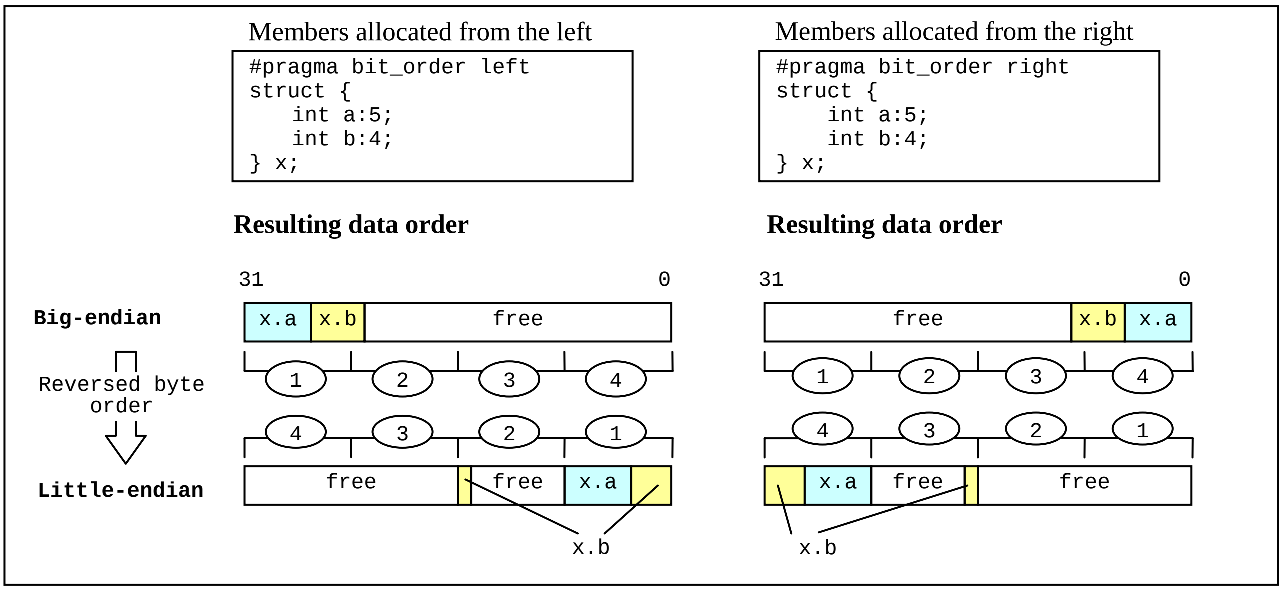
Members allocated from the right

```
#pragma bit_order right
struct {
    char a:5;
    char :0;
    char b:3;
} x;
```

Figure 2-7

Example 6:

The default endian is little-endian, but can be changed using an option. When little-endian (endian=little) is specified, the sort order for each area is the reverse byte order for that of big-endian.



Members allocated from the left

```
#pragma bit_order left
struct {
    int a:5;
    int b:4;
} x;
```

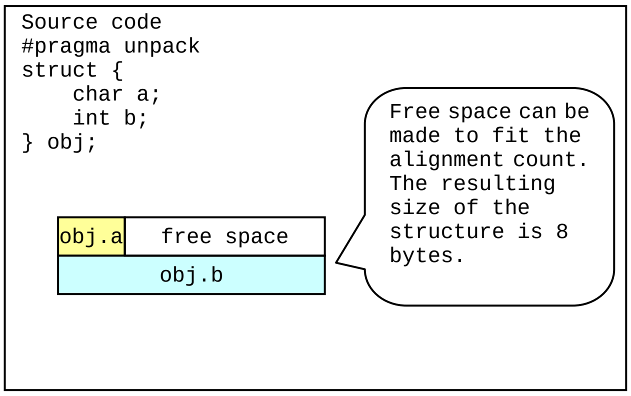
Members allocated from the right

```
#pragma bit_order right
struct {
    int a:5;
    int b:4;
} x;
```

Figure 2-8

2.3 Alignment control to structure member

When a structure (including a shared structure or class) contains a mix of 1-byte, 2-byte, and 4-byte members, free space may exist between each member, due to their alignment count.



```
Source code
#pragma unpack
struct {
    char a;
    int b;
} obj;
```

obj.a
free space
obj.b
Free space can be made to fit the alignment count. The resulting size of the structure is 8 bytes.

Figure 2-9

Sometimes, the creation of free space needs to be avoided in structures used for communication programs. In such cases, #pragma pack can be specified to set the alignment count of structure members to 1.

The empty area between members is not made in the structure of which one is the number of alignments of members.
However, when the member of the structure of which one is the number of alignments is accessed, there is a possibility that the speed performance decreases compared with before the number of alignments is changed.
This prevents free space from being created in structures with an alignment count of 1. However, the speed performance of member access for structures with an alignment count of 1 may suffer.

The "pack" option can be used to set all a structural arrangement counts to 1. When both the "pack" option and #pragma are specified at the same time, the #pragma specification takes precedence.

Format

#pragma pack
#pragma unpack
#pragma packoption

The alignment counts for these specifications are as follows.

Table 2-4 Alignment counts for structures, shared structures, class members

Specification	#pragma pack	#pragma unpack	#pragma packoption, or unspecified
[signed]char	1	1	1
[unsigned]short	1	2	Follows the pack option
[unsigned]int, [unsigned]long, [unsigned]long long, floating point number type, pointer type	1	4	Follows the pack option
Structures, shared structures, and classes with an alignment count of 1	1	1	1
Structures, shared structures, and classes with an alignment count of 2	1	2	Follows the pack option
Structures, shared structures, and classes with an alignment count of 4	1	4	Follows the pack option

Example:

The structure allocation is performed as follows.

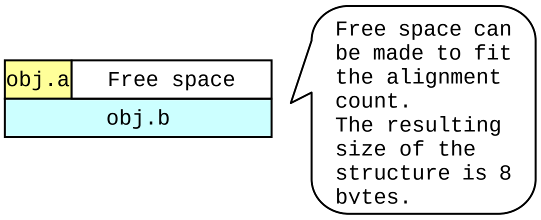
Source code with #pragma unpack specified

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

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

Assembler expansion code

```
_func:                       ; function: func
            .STACK      _func=4
L10:
            MOV.L       #00000001H,R5
            MOV.L       #_obj,R4
            MOV.L       R5,04H[R4]
            MOV.B       R5,[R4]
            RTS
            .SECTION    B,DATA,ALIGN=4
            .glb        _obj
_obj:                        ; static: obj
            .blkl       2
```


obj.a
Free space
obj.b
Free space can be made to fit the alignment count. The resulting size of the structure is 8 bytes.

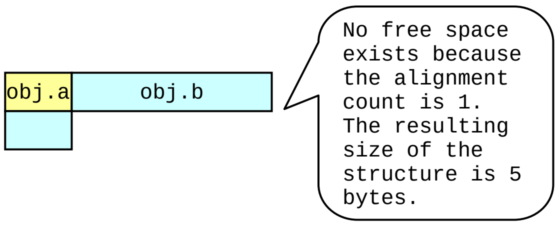
Source code with #pragma pack specified

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

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

Assembler expansion code

```
_func:                       ; function: func
            .STACK      _func=4
L10:
            MOV.L       #_obj,R3
            MOV.L       #00000001H,R5
            ADD         #01H,R3,R4
            MOV.L       R5,[R4]
            MOV.B       R5,[R3]
            RTS
            .SECTION    B_1,DATA
            .glb        _obj
_obj:                         ; static: obj
            .blkb       5
```


obj.a
obj.b
No free space exists because the alignment count is 1. The resulting size of the structure is 5 bytes.

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