

To our customers,

---

## Old Company Name in Catalogs and Other Documents

---

On April 1<sup>st</sup>, 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 1<sup>st</sup>, 2010  
Renesas Electronics Corporation

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

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

## Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
  - “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
  - “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
  - “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

# RX Family C/C++ Compiler Package

## Application Notes: RX Migration Guide, M16C Edition

This document explains the items that need to be checked for migration from M16C, for C/C++ Compiler V1 for the RX family.

### Table of Contents

Introduction .....	2
1. Options .....	3
1.1 Specifying changes to enumeration size .....	3
1.2 Specifying the size of double type variables .....	3
1.3 Specifying the size of int type variables .....	5
2. Language specifications .....	6
2.1 Size of int type variables .....	6
2.2 Size of the double type .....	7
2.3 Integer promotion for the char type .....	8
2.4 Placement of structure members .....	9
2.5 Inline keywords .....	10
2.6 #pragma STRUCT .....	12
2.7 #pragma BITADDRESS .....	14
2.8 #pragma ROM .....	15
2.9 #pragma PARAMETER .....	16
2.10 asm .....	17
3. Optimization option setting for migration from M16C-family .....	18
Exsample: Sample source .....	19
Web site and support <website and support> .....	20

## Introduction

In this application notes, it explains the software migration method when C program made by M16C-family compiler is transplanted to RX-family compiler.

In Renesas RX-family compiler, the function to absorb the difference between the option and the language specification is supported inconsideration of the migration from M16C-family to RX-family. As a result, the application part of the embedded software can be smoothly transplanted.

In this application notes, it explains the use of the function that the difference between the option and the language specification of the compiler for the M16C-family, and the RX-family and compilers for the RX-family support it.

Please use this application notes when you transplanted to RX-family from M16C-family.

## 1. Options

Some specifications differ for the default options between M16C-family compilers and RX-family compilers. The following explains options that will likely require handling during migration from M16C to RX.

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 for reference.

Table 1-1 List of options

No	Functionality	M16C option	RX option	Reference
1	Specifying changes to enumeration size	fchar_enumerator	auto_enum	1.1
2	Specifying the size of double type variable	fdouble_32	dbl_size=4	1.2
3	Specifying the size of int type variables	--	int_to_short	1.3

### 1.1 Specifying changes to enumeration size

When the “fchar\_enumerator” option is specified for M16C-family compilers, the enumerator type is treated as an unsigned char type, not an int type.

To obtain the same results as those on RX-family compilers, specify the “auto\_enum” option. Note that the unsigned char type is only used for the enumerator type when the minimum enumerator value is 0, and the maximum value is 255. For all other cases, another type is used.

For details about changing the enumeration type size, see *1.2.3 Enumeration type size in C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Option Edition*.

### 1.2 Specifying the size of double type variables

If the “fdouble\_32” option is not specified for an M16C-family compiler, the size of the double type is treated as 8 bytes (double precision).

For the same interpretation as RX-family compilers, specify the “dbl\_size=8” option.

Format

**dbl\_size = {4|8} : 4 by default**

[How to specify this option in the Renesas IDE]

Choose Build and then RX Standard Toolchain, and perform the following settings in the displayed dialog box.

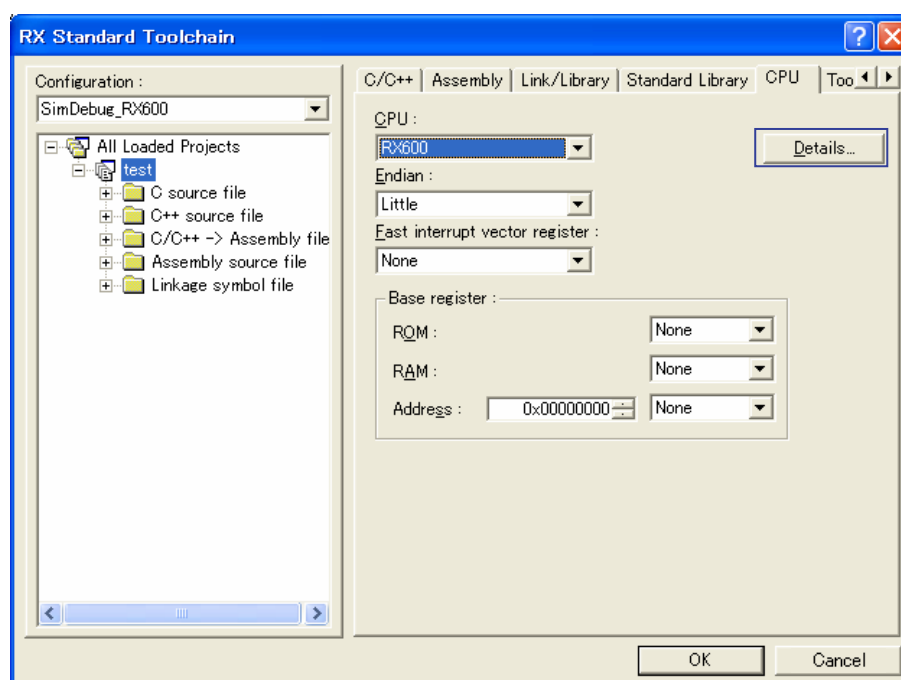


Figure 1-1

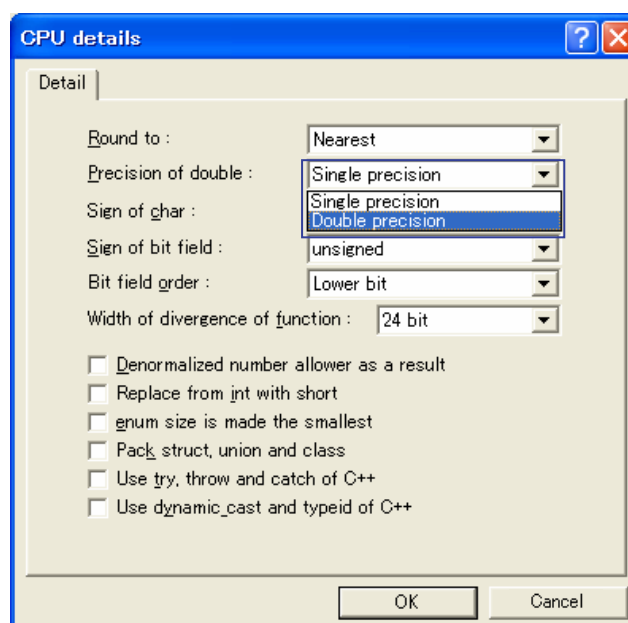


Figure 1-2

## 1.3 Specifying the size of int type variables

For M16C-family compilers, the size of the int type is treated as 2 bytes, whereas for RX-family compilers, the size of the int type is treated as 4 bytes.

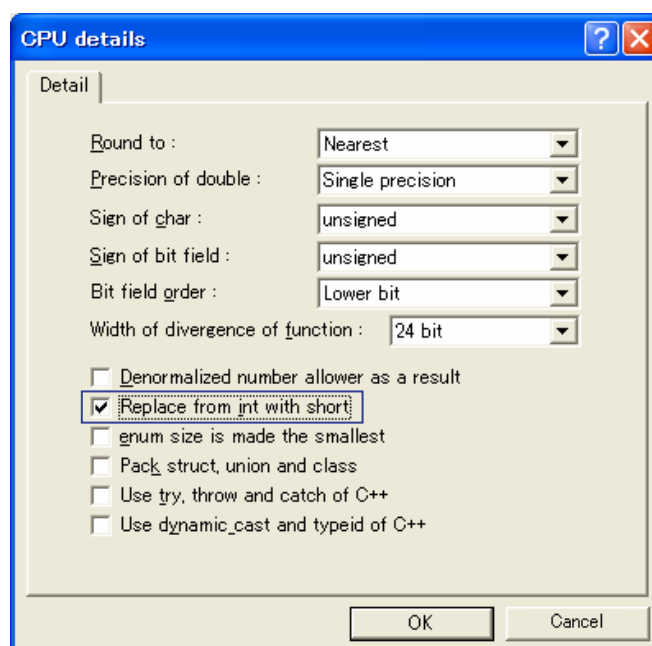
When migrating M16C programs created based on the requirement that the size of the int type is 2 bytes to RX, specify the “int\_to\_short” option.

Format

**int\_to\_short**

[How to specify this option in the Renesas IDE]

In RX Standard Toolchain (Figure 1-1), choose Details, and perform the following settings in the displayed dialog box.



**Figure 1-3**

## 2. Language specifications

This chapter explains the language specification most likely to require changes during RX migration.

Table 1-2 List of language specifications

No	Functionality	Reference
1	Size of int type	2.1
2	Size of the double type	2.2
3	Integer promotion for the char type	2.3
4	Placement of structure members	2.4
5	Inline keyword	2.5
6	#pragma STRUCT	2.6
7	#pragma BITADDRESS	2.7
8	#pragma ROM	2.8
10	#pragma PARAMETER	2.9
11	asm	2.10

### 2.1 Size of int type variables

On M16C-family compilers, the size of the int type is 2 bytes, whereas on RX-family compilers the size of the int type is 4 bytes in default. When M16C programs created based on the requirement that the size of the int type is 2 bytes are migrated to RX, they may not operate properly.

Example: Code for which operation is different due to variance in int type size

```
Source code
typedef union{
    long data;
    struct {
        int dataH;
        int dataL;
    } s;
} UN;

void main(void)
{
    UN u;
    u.data = 0x7f6f5f4f;

    if (u.s.dataH == 0x5f4f && u.s.dataL == 0x7f6f) {
        // When the size of the int type is 2 bytes (M16C)
    } else {
        // When the size of the int type is 4 bytes (RX)
    }
}
```

When migrating programs created based on the requirement that the size of the int type is 2 bytes to RX, specify the “int\_to\_short” option. For details about specifying this option, see *1.3 Specifying the size of int type variables*.



## 2.2 Size of the double type

With M16C-family compilers, the size of the double type is 8 bytes, whereas with RX-family compilers, the size of the double type is 4 bytes in default. When M16C programs created based on the requirement that the size of the double type is 8 bytes are migrated to RX, they may not operate properly.

Example: Code for which operation is different due to variance in double type size

### Source code

```
double d1 = 1E30;
double d2 = 1E20;

void main(void)
{
    d1 = d1 * d1;
    d2 = d2 * d2;

    if (d1 > d2) {
        // When the size of the double type is 8 bytes (M16C)
    } else {
        // When the size of the double type is 4 bytes (RX)
    }
}
```

When migrating programs created based on the requirement that the size of the double type is 8 bytes to RX, specify the “dbl\_size=8” option. For details about specifying this option, see *1.2 Specifying the size of double type variable*.

## 2.3 Integer promotion for the char type

With M16C-family compilers, when char type data (including unsigned char, signed char types) is evaluated, it is not promoted to the int type, whereas with RX-family compilers, char type data is always promoted to the int type when evaluated. When migrating to MX programs created based on this condition in M16C, operation may not be performed the same as M16C.

Example: Code for which operation is different due to variance in integer promotion specifications for char type calculations

```
Source code

char c1;
char c2 = 200;
char c3 = 200;

void main(void)
{
    c1 = (c2 + c3) / 2;           // Unexpected results occur on M16C, because the maximum
                                // value that can be expressed for the calculation of
                                // c2 + c3 causes an overflow

    if (c1 == 200) {
        // When the char type is promoted to the int type and evaluated (RX)
    } else {
        // When the char type is evaluated without being promoted to the int type (M16C)
    }
}
```

### Notes

Options exist for promotion to the int type when M16C-family compilers evaluate char type data (including unsigned char types and signed char types). If one of the following options is specified, no problems will occur for the integer promotion specification explained here.

- -fansi
- -fextend\_to\_int

## 2.4 Placement of structure members

M16C-family compilers place structure members in the order of appearance for member data with alignment count 1, whereas RX-family compilers place structure members in the order of appearance for member data, according to the maximum alignment count for members. When M16C programs created based on the requirement that M16C structure placement is used are migrated to RX, they may not operate properly. In this case, the “pack” option can be specified to set the alignment count for structure members to 1. Free space will no longer be able to be created for structures with alignment counts of 1.

Structure alignment counts can also be specified using #pragma pack. If both the option and #pragma are specified at the same time, the #pragma specification takes precedence.

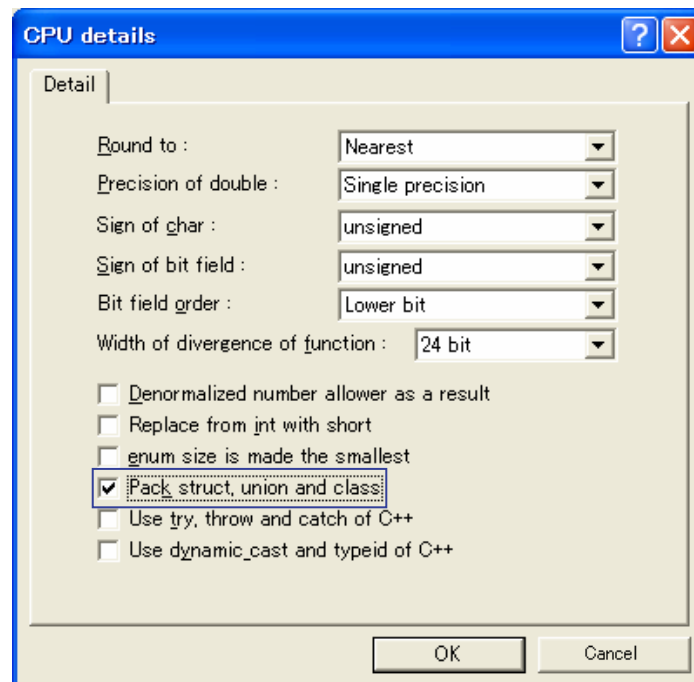
For details about this functionality, see 2.3 *Specifying structure, shared structure, and class alignment count in C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition*.

Format

**pack**  
**unpack**

[How to specify this option in the Renesas IDE]

Choose the CPU tab, then choose Details (**Figure 1-1**), and then perform the following settings in the displayed dialog box.



**Figure 1-4**

## 2.5 Inline keywords

M16C-family compilers support inline keywords, whereas RX-family compilers do not support inline keywords for ANSI-standard C89.

When a program using inline keywords for M16C is built on the RX ANSI-standard C89, inline keywords cause compilation errors for non-standard builds. This can be resolved by either of the following:

- When performing builds using ANSI-standard C89, convert inline code to `#pragma inline`.
- Perform builds using ANSI-standard C99.

The following shows an example of converting inline keywords to `#pragma inline`.

Example:

Source code (using inline keywords)	Source code (using <code>#pragma inline</code> )
<pre>inline static int func(int a, int b) {     return (a + b) / 2; } int x; main() {     x = func(10, 20); }</pre>	<pre>#pragma inline(func) static int func(int a, int b) {     return (a + b) / 2; } int x; main() {     x = func(10, 20); }</pre>

The option settings for builds on ANSI-standard c99 are as follows.

Format

`lang={c|cpp|ecpp|c99}`

[How to specify this option in the Renesas IDE]

From HEW, choose Build and then RX Standard Toolchain, and then perform the following settings in the displayed dialog box.

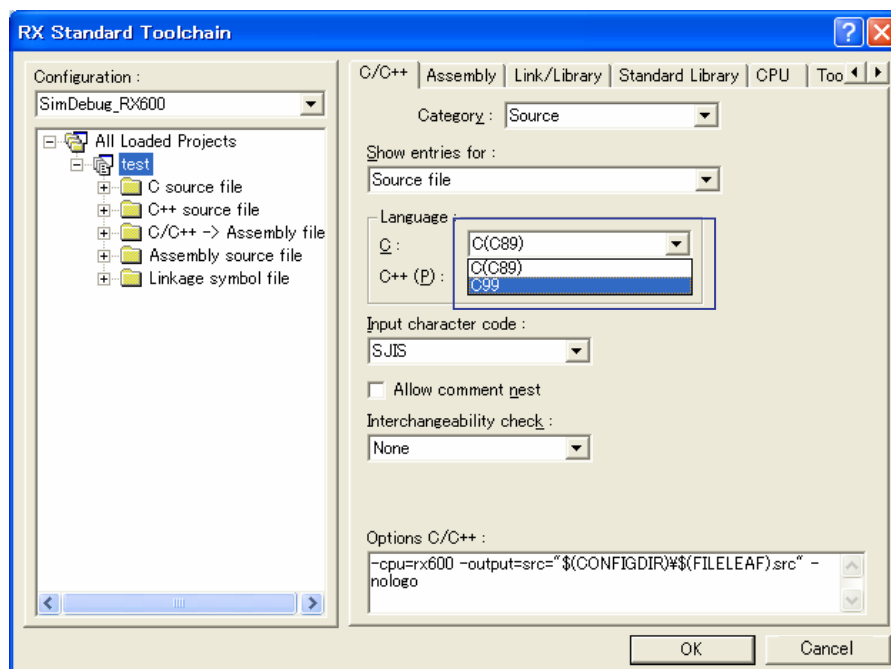


Figure 1-5

## Precautions

1. The ANSI-standard C99, C++, and #pragma inline specifications differ as follows.

Table 1-3 Inline specifications

	Default linkage	Linkage when inline expansion is not possible	Impact on declarative linkage
C99 inline keyword	Internal linkage	Internal linkage	When extern is specified: External linkage
C++ inline keyword	Internal linkage	Internal linkage	When extern is specified: Compilation error
#pragma inline	External linkage	External linkage	When static is specified: Internal linkage

2. Progressing inline key word and #pragam inline is different. #pragma inline is compulsion development.

## 2.6 #pragma STRUCT

M16C-family compilers can prohibit structure packing using #pragma STRUCT, and perform sorting for structure members. Since RX-family compilers lack the corresponding functionality, programs using #pragma STRUCT need special handling when migrated to RX.

### (1) Prohibiting structure packing

By default, M16C-family compilers place structure members with alignment 1. This means that the structure size may be an odd number of bytes. To make the structure size an even number of bytes, specify #pragma STRUCT unpack. If this specification causes the structure size to be an odd number of bytes, 1 byte of padding is inserted to bring the structure size to an even number of bytes.

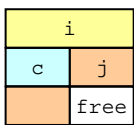
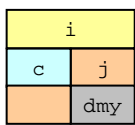
One of the following needs to be performed to achieve the same effect for RX-family compilers.

- Specify #pragma pack to set the structure alignment to 1.
- If setting the alignment to 1 causes the structure size to be an odd number of bytes, have users enter a 1-byte dummy member for adjustment.

In addition to specifying #pragma pack, specifying the “pack” option is another way to set structure alignment to 1 on RX-family compilers. For details about this functionality, see 2.4 *Placement of structure members*.

Example:

### M16C program with #pragma STRUCT unpack and program migrated to RX

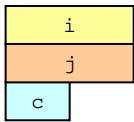
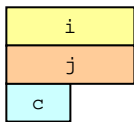
Source code with #pragma STRUCT unpack specified	RX source code
<pre>#pragma STRUCT s unpack struct s {     short i;     char c;     short j; } ss;</pre>	<pre>#pragma pack struct s {     short i;     char c;     short j;     char dmy; // To get a size with an even number               // of bytes a 1-byte dummy member               // is inserted } ss;</pre>
	

### (2) Sorting structure members

RX-family compilers do not have any functionality to sort structure members. Programs need to be changed to perform member sorting.

Example:

M16C program with #pragma STRUCT arrange and program migrated to RX

Source code with #pragma STRUCT arrange specified	RX source code
<pre>#pragma STRUCT s arrange struct s {     short i;     char c;     short j; } ss;</pre>	<pre>#pragma pack struct s {     short i;     short j;      // Placement of members 'j' and 'c'     char c;        // is changed } ss;</pre>
 <p>#pragma STRUCT places even-byte members before, and odd-byte members after.</p>	 <p>Since the alignment count is 1 due to pack, there is no free space. The structure members are placed in order of appearance.</p>

### Notes

Keep in mind that the meanings of #pragma STRUCT unpack for M16C and #pragma unpack for RX are different.

- #pragma STRUCT unpack (M16C)

Adjusts the size of the specified structure to an even number of bytes.

- #pragma unpack (RX)

Makes the alignment of the specified structure the same as the maximum alignment count for members.

## 2.7 #pragma BITADDRESS

M16C-family compilers can allocate variables to a specified bit position for an absolute address specified by #pragma BITADDRESS. Since RX-family compilers lack the corresponding functionality, programs using #pragma BITADDRESS need special handling when migrated to RX.

The following gives an example where the absolute address bit position matches the structure bit field.

Example:

M16C program with 100th position bit number 1 for #pragma BITADDRESS set to 1 and program migrated to RX

Source code with #pragma BITADDRESS specified	RX source code
<pre>#pragma BITADDRESS io 1, 100H _Bool io;  void main(void) {     io = 1; }</pre>	<pre>struct bit_address {     unsigned char b0:1;     unsigned char b1:1;     unsigned char b2:1;     unsigned char b3:1;     unsigned char b4:1;     unsigned char b5:1;     unsigned char b6:1;     unsigned char b7:1; };  #define io (((struct bit_address*)0x100)-&gt;b1)  void main(void) {     io = 1; }</pre>

### Notes

When a program using the \_Bool keyword on an M16C-family compiler is built on a RX-family compiler with ANSI-standard C89, an error occurs because the \_Bool keyword is non-standard. RX-family compilers support the \_Bool keyword with ANSI-standard C99.



## 2.8 #pragma ROM

M16C-family compilers place variables specified by #pragma ROM in the rom section. Since RX-family compilers lack the corresponding functionality, when a program using #pragma ROM is migrated to RX, the const modifier needs to be used to place variables in the rom section, as shown in the following example.

Example:

M16C program with variable 'i' placed in the rom section using #pragma ROM, and program migrated to RX

<u>Source code with #pragma ROM specified</u>	<u>RX source code</u>
#pragma ROM i unsigned short i;	const unsigned short i; // const keyword added
<u>Assembler expansion code</u>	<u>Assembler expansion code</u>
.SECTION rom_FE,ROMDATA,align .glb _i _i: .byte 00H .byte 00H	.SECTION C_2,ROMDATA,ALIGN=2 .glb _i _i: ; static: i .byte 00H .byte 00H

## 2.9 #pragma PARAMETER

M16C-family compilers can use #pragma PARAMETER to store arguments in a register and declare passed assembler functions. Since RX-family compilers lack the corresponding functionality, programs using #pragma PARAMETER need special handling when migrated to RX.

RX-family compilers have no way to specify that arguments be stored in arbitrary registers. The argument interface for function calls needs to follow C/C++ generation rules. Change the argument interface for assembler functions specified by #pragma PARAMETER during RX migration to adhere to C/C++ generation rules.

For details about argument interfaces, see *1.2 Performing inline expansion in assembly code functions in C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition*.

Example:

M16C program using #pragma PARAMETER, and program combining the assembler function and argument interface in RX

Source code with #pragma PARAMETER specified	Source code using RX assembly code functions
<u>C source code</u>  <pre>int asm_func(unsigned int, unsigned int); #pragma PARAMETER asm_func(R0, R1)  void main(void) {     int i, j;      i = 0x7FFD;     j = 0x007F;      asm_func( i, j ); // Assembler function call                       // 'i' is stored in R1, 'j'                       // is stored in R0 }</pre> <u>Assembler source code</u>  <pre>_asm_func:     Code requiring 'i' to be in R1, 'j' to be in     R0 for passage     ...     RTS</pre>	<u>C source code</u>  <pre>int asm_func(unsigned int, unsigned int);  void main(void) {     int i, j;      i = 0x7FFD;     j = 0x007F;      asm_func( i, j ); // Assembler function call                       // As per the C/C++ argument                       // interface,                       // 'i' is stored in R1, 'j' is                       // stored in R2 }</pre> <u>Assembler source code</u>  <pre>_asm_func:     Changing code requiring 'i' to be in R1, 'j' to     be in R2 for passage     ...     RTS</pre>

## 2.10 asm

M16C-family compilers can use the asm function to code assembly language routines (asm functions) within C source programs. Since RX-family compilers lack the corresponding functionality, programs using asm functions need special handling when migrated to RX.

Assembly code functions exist to code assembly language in a C source program in RX. The contents coded in the asm function can sometimes be handled by being coded in the assembly code function.

For details about assembly code functions, see *1.2 Performing inline expansion in assembly code functions* in *C/C++ Compiler Package for the RX Family Application Notes: Compiler Usage Guide, Extended Functionality Edition*.

Example:

M16C program using the asm function, and program using the assembly code function in RX

(The code contents are not equivalent, but this can be used as an example of migration.)

Source code with the M16C asm function specified	Source code using the RX assembly code function
<u>C source code</u> <pre>void func(void) {     asm("FSET I"); }</pre>	<u>C source code</u> <pre>#pragma inline_asm interrupt_flag static void interrupt_flag(void) {     MOV.L    #00010000H,R5     MVTC    R5,PSW }  void func(void) {     interrupt_flag(); }</pre>
<u>Assembler source expansion code</u> <pre>_func:     FSET I     rts</pre>	<u>Assembler source expansion code</u> <pre>_func:     MOV.L    #00010000H,R5     MVTC    R5,PSW     RTS</pre>

### Precautions

- M16C can code variables in the asm function, but RX cannot.
- M16C can use a dummy asm function as a step to partially suppress optimization, but RX cannot.

### 3、Optimization option setting for migration from M16C-family

There is a difference in an optional setting method for optimization in the compiler of M16C-family and RX-family.  
Please refer to the following optimization option setting when embedded software transplant from M16C-family to RX-family and the performance is evaluated.

Optimization option setting of each compiler and comparison of ROM size

(The sample program for the measurement is described to the next page.)

M16C	Optimize OFF	Optimize Size			Optimize Speed		
	opt=0	opt=O3 OR	-	opt=OR_max	opt=O3 OS	-	opt=OS_max
main()	0xA2	0x80	-	0xA0	0xA1	-	0x27A
sort()	0xB3	0x79	-	0x79	0x7C	-	0x7B

RX	Optimize OFF	Optimize Size			Optimize Speed		
	opt=0	opt=1	opt=2	opt=max	opt=1 speed	opt=2 speed	opt=max speed
main()	0xD1	0x95	0x6A	0x6A	0x95	0xEB	0x11A
sort()	0xFD	0x69	0x65	0x65	0x6A	0x5F	0x5F

Optimization option setting for migration from M16C-family

- (1) Please specify it for RX-family above opt=1 or opt=1 speed when you specify opt=0 for M16C-family.  
In opt=0 for RX-family, the meaning is different from opt=0 for M16C-family. Optimization is not executed at all.  
Therefore, the performance of RX-family looks bad.  
Exsample:  
 • If O0 for M16C is used, rom size is 0xA2 (main()) -> If optimize=0 for RX is used, rom size is 0xD1 (main()) ...NG  
 • If O0 for M16C is used, rom size is 0xA2 (main()) -> If optimize=1 for RX is used, rom size is 0x95 (main()) ...OK

Please refer to the compiler user's manual for details of the optimization level of the RX compiler.

# Exsample: Sample source

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void main(void);
void sort(long *a);
void change(long *a);

void main(void)
{
    long a[10];
    long j;
    int i;

    printf("### Data Input ###\n");

    for( i=0; i<10; i++ ){
        j = rand();
        if(j < 0){
            j = -j;
        }
        a[i] = j;
        printf("a[%d]=%ld\n",i,a[i]);
    }
    sort(a);
    printf("*** Sorting results ***\n");
    for( i=0; i<10; i++ ){
        printf("a[%d]=%ld\n",i,a[i]);
    }
    change(a);
}
```

```
void sort(long *a)
{
    long t;
    int i, j, k, gap;

    gap = 5;
    while( gap > 0 ){
        for( k=0; k<gap; k++){
            for( i=k+gap; i<10; i=i+gap ){
                for(j=i-gap; j>=k; j=j-gap){
                    if(a[j]>a[j+gap]){
                        t = a[j];
                        a[j] = a[j+gap];
                        a[j+gap] = t;
                    }else{
                        break;
                    }
                }
            }
        }
        gap = gap/2;
    }
}
```

```
void change(long *a)
{
    long tmp[10];
    int i;
    for(i=0; i<10; i++){
        tmp[i] = a[i];
    }
    for(i=0; i<10; i++){
        a[i] = tmp[9 - i];
    }
}
```

## 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](mailto:csc@renesas.com)

## Revision history<revision history,rh>

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