RX Development Environment Migration Guide

REJ06J0078-0200
Rev.2.00
Nov. 30, 2016

Migration from M16C Family to RX Family (Compiler ed.)
(High-performance Embedded Workshop and NC30WA to CS+ and CC-RX)

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 and M16C-family to RX-family. As a result, the application part of the embedded software can be smoothly transplanted.

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

Contents

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.

Table 1-1 List of options

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

1.1 Specifying changes to enumeration type 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 correspondences between possible enumeration values and data types, see Compiler User’s Manual.

Format

auto_enum

[How to specify this option in CS+]

Perform the following settings in the [Common Options] page of CC-RX (build tool) properties.

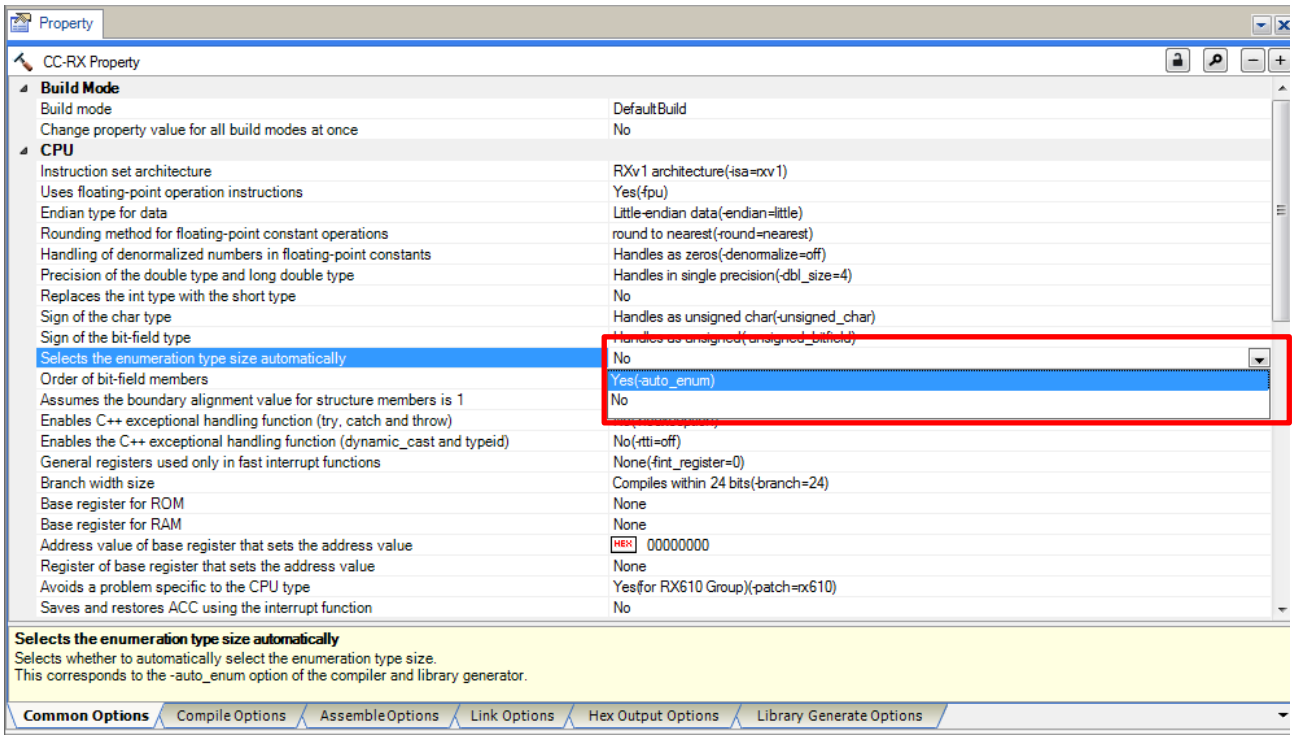


Figure 1-1

1.2 Specifying the size of double type

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} : 4by default

[How to specify this option in CS+]

Perform the following settings in the [Common Options] page of CC-RX (build tool) properties.

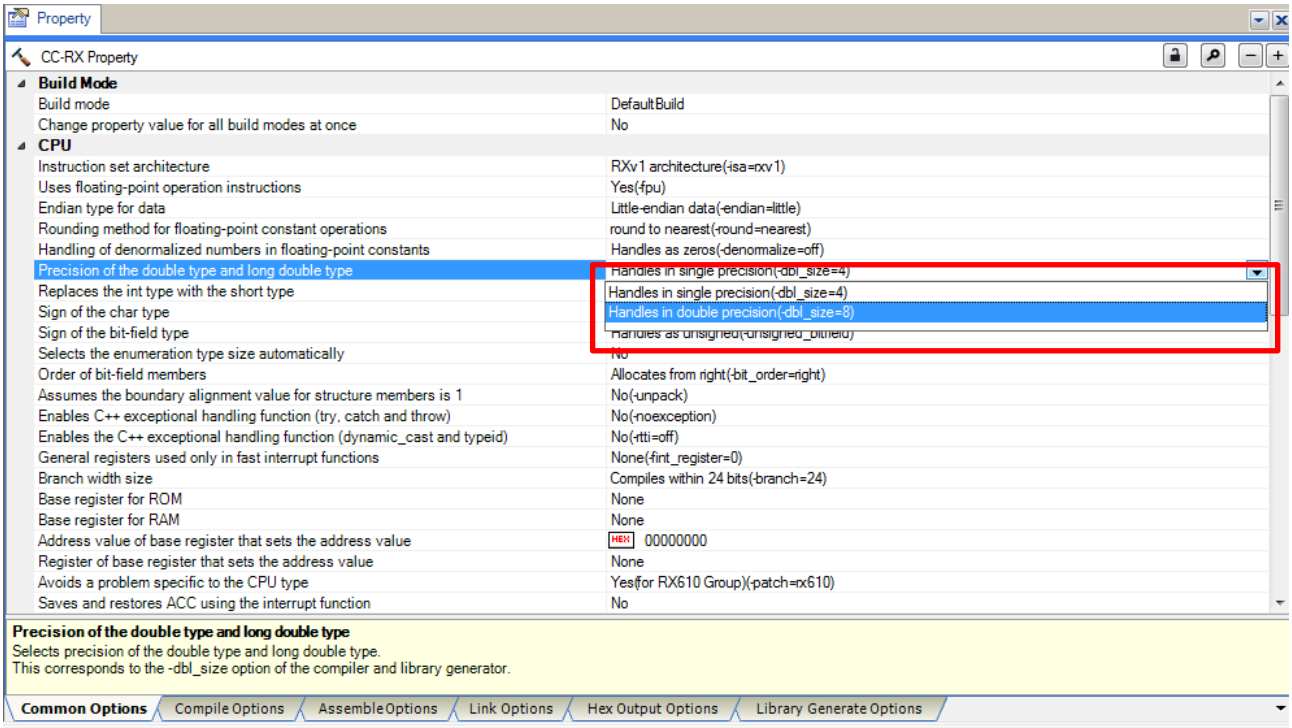


Figure 1-2

1.3 Specifying the size of int type

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 CS+]

Perform the following settings in the [Common Options] page of CC-RX (build tool) properties.

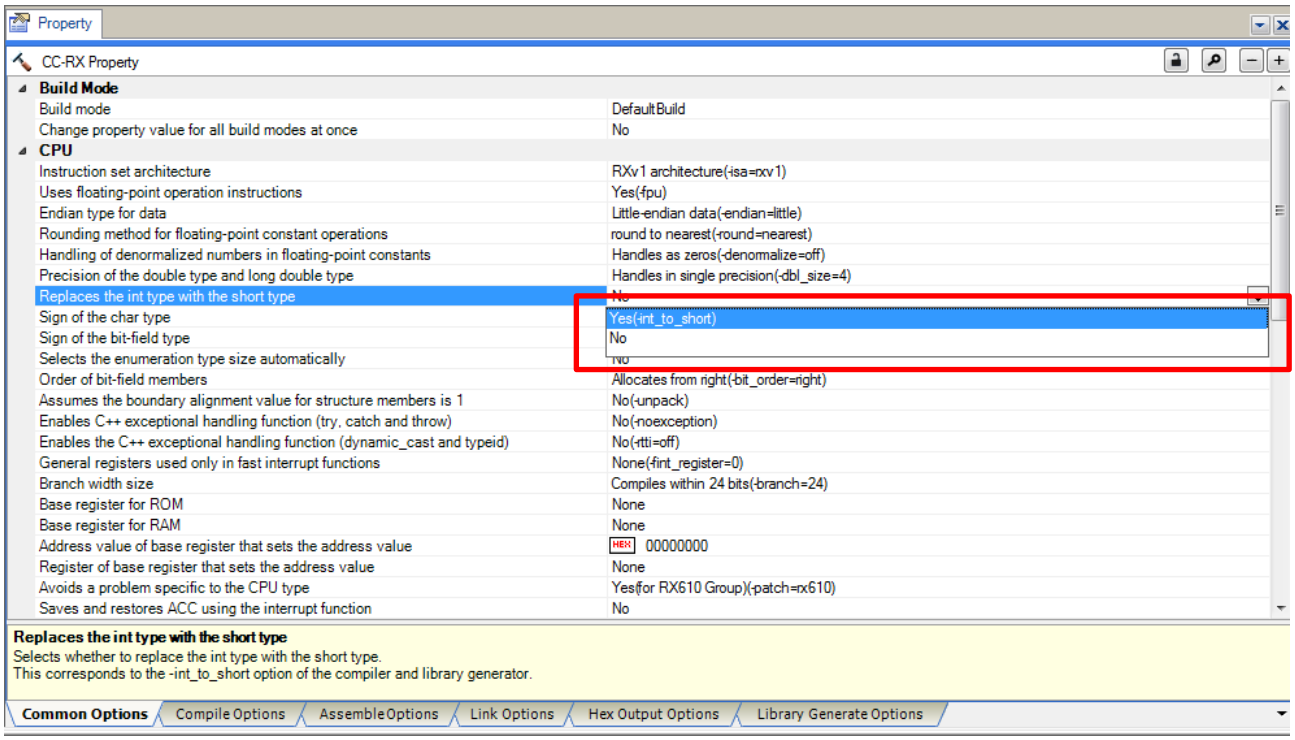


Figure 1-3

2. Language specifications

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

Table 2-1 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 keywords	2.5
6	#pragma STRUCT	2.6
7	#pragma BITADDRESS	2.7
8	#pragma ROM	2.8
9	#pragma PARAMETER	2.9
10	asm function	2.10

2.1 Size of int type

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.s.dataH = 0;
  u.s.dataL = 1;

  if (u.data == 0) {
    // When the size of the int type is 4 bytes(RX)
  } else {
    // When the size of the int type is 2 bytes(M16C)
  }
}
```

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.

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.

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 value 1, whereas RX-family compilers place structure members in the order of appearance for member data, according to the maximum alignment value 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 value for structure members to 1. Free space will no longer be able to be created for structures with alignment value of 1.

In addition, structure alignment value can also be specified using #pragma pack too. If both the option and #pragma are specified at the same time, the #pragma specification takes precedence.

For details about this functionality, see Compiler User’s Manual.

Format

pack : unpack by default

unpack

[How to specify this option in CS+]

Perform the following settings in the [Common Options] page of CC-RX (build tool) properties.

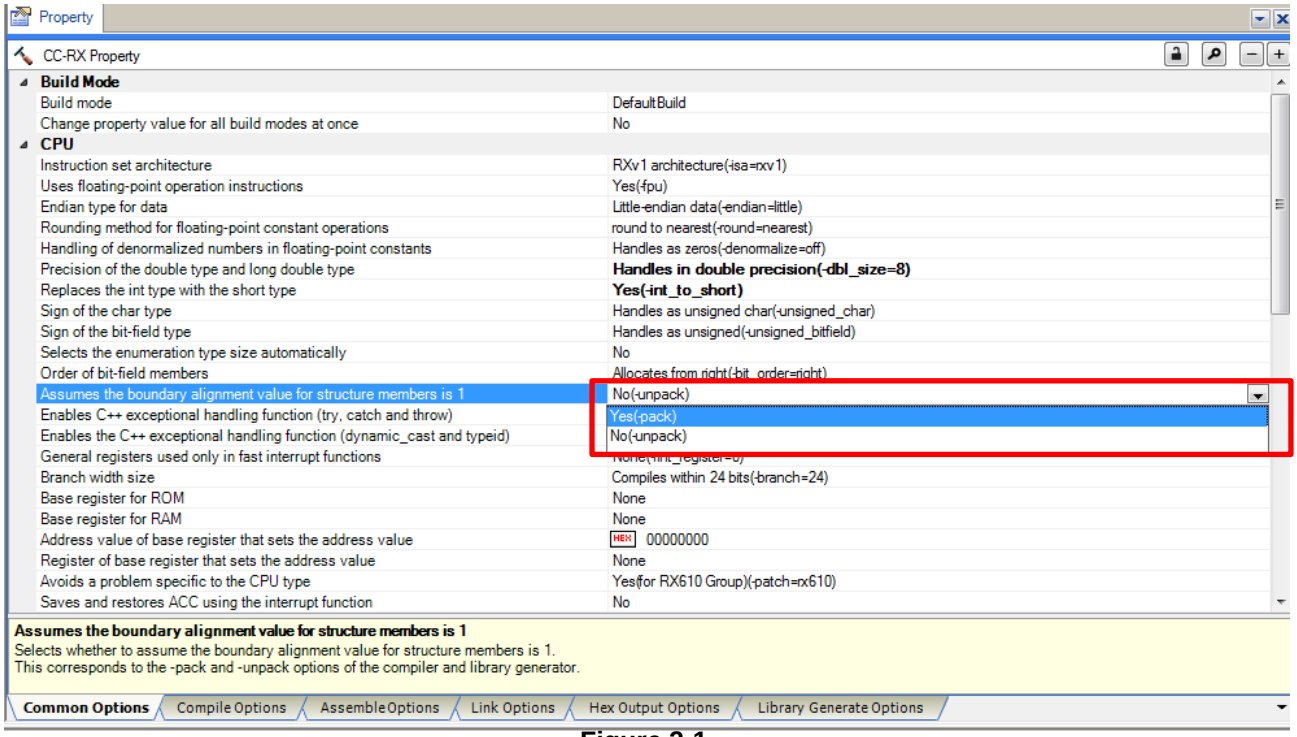


Figure 2-1

2.5 inline keywords

M16C-family compilers support inline keywords, whereas RX-family compilers do not support inline keywords for ANSI-standard C89. Thus, a program using inline keywords cause compilation errors. This can be resolved by either of the following:

- Convert inline keywords to #pragma inline.
- Perform builds using ANSI-standard C99.

Example:

M16C program with inline keyword and program migrated to #pragma inline of RX.

Source code (using inline keywords)	Source code (using #pragma inline)

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

When building on ANSI-standard C99, specify the “lang=c99” option.

Format

lang={c|cpp|ecpp|c99}

[How to specify this option in CS+]

Perform the following settings in the [Compile Options] page of CC-RX (build tool) properties.

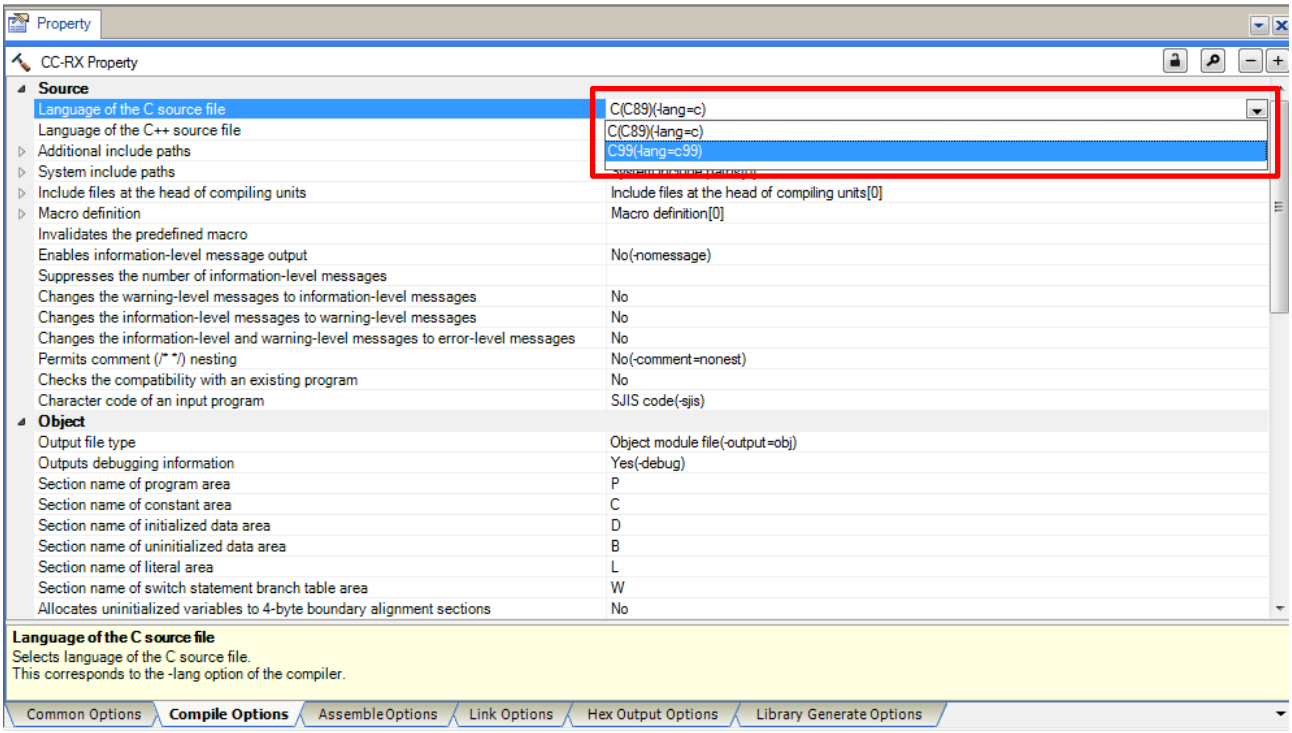


Figure 2-2

Precautions

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

Table 2-2 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. Expansion of #pragma inline is different from inline keyword. #pragma inline is forced directive.

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

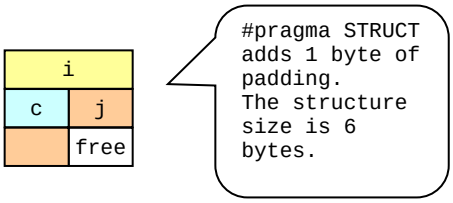
In addition to specifying #pragma pack, specifying the “pack” option is another way to set structure alignment value to 1 on RX-family compilers. For details about this functionality, see 2.4 Placement of structure members.

Example:

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

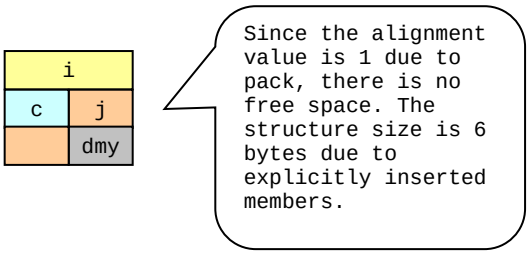
Source code with #pragma STRUCT unpack specified

```
#pragma STRUCT s unpack
struct s {
    short  i;
    char  c;
    short  j;
} ss;
```



RX source code

```
#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;
```



(2) Sorting structure members

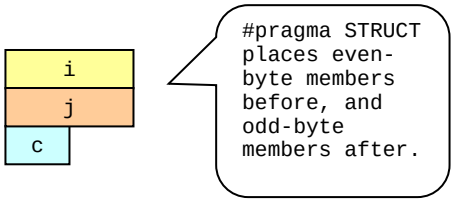
RX-family compilers do not have any functionality to sort structure members. Programs need to be changed to perform member sorting.

Example:

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

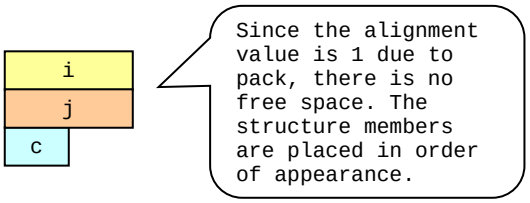
Source code with #pragma STRUCT arrange specified

```
#pragma STRUCT s arrange
struct s {
    short  i;
    char  c;
    short  j;
} ss;
```



RX source code

```
#pragma pack
struct s {
    short  i;
    short  j;  // Placement of members
    char c;    //  'j' and 'c' is changed
} ss;
```



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 value of the specified structure the same as the maximum alignment value 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

```
#pragma BITADDRESS io 1, 100H
_Bool io;

void main(void)
{
  io = 1;
}
```

RX source code

```
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)->b1)

void main(void)
{
  io = 1;
}
```

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.

Example:

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

Source code with #pragma ROM specified

```
#pragma ROM i
unsigned short i;
```

Assembler expansion code

```
  .SECTION rom_FE,ROMDATA,align
  .glb _i
_i:
  .byte 00H
  .byte 00H
```

RX source code

```
const unsigned short i;  // const keyword added
```

Assembler expansion code

```
  .glb _i
  .SECTION C_2,ROMDATA,ALIGN=2
_i:
  .word 0000H
```

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 compiler generation rules. Change the argument interface for assembler functions specified by #pragma PARAMETER during RX migration to adhere to compiler generation rules.

For details about argument interfaces, see Compiler User's Manual.

Example:

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

Source code with #pragma PARAMETER specified

C source code

```
int asm_func(unsigned int, unsigned int);
#pragma PARAMETER asm_func(R0, R1)

void main(void)
{
  int i, j;

  i = 0x7FFD;
  j = 0x007F;

  // Assembler function call
  // 'i' is stored in R1, 'j' is stored in R0
  asm_func( i, j );
}
```

Assembler source code

```
_asm_func:
  Code requiring 'i' to be in R1,
  ' j' to be in R0 for passage
    …
  RTS
```

Source code using RX assembly code functions

C source code

```
int asm_func(unsigned int, unsigned int);
void main(void)
{
  int i, j;

  i = 0x7FFD;
  j = 0x007F;

  // Assembler function call
  // As per the compiler argument interface,
  // 'i' is stored in R1, 'j' is stored in R2
  asm_func( i, j );
}
```

Assembler source code

```
_asm_func:
  Changing code requiring 'i' to be in R1,
  ' j' to be in R2 for passage
    …
  RTS
```

2.10 asm function

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

RX-family compilers have assembly code functions to code assembly language in C source programs. 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 Compiler User's Manual.

Example:

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

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

Source code using the M16C asm function

C source code

```
void func(void)
{
  asm("FSET I");
}
```

Assembler source expansion code

```
_func:
  FSET I
  rts
```

Source code using the RX assembly code function

C source code

```
#pragma inline_asm interrupt_flag
static void interrupt_flag(void)
{
  MOV.L #00010000H,R5
  MVTC R5,PSW
}

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

Assembler source expansion code

```
_func:
  MOV.L #00010000H,R5
  MVTC R5,PSW
  RTS
```

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-famiy 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	Object Size Precedence		Execution Speed Precedence	
	O0	O3 OR	OR_MAX	O3 OS	OS_MAX
main()	0xE6	0x99	0x98	0xB4	0x358
sort()	0xF6	0x92	0x91	0x94	0x94

* By M16C-family compiler V.6.00 Release 00

RX	Optimize OFF	Object Size Precedence			Execution Speed Precedence		
	optimize=0	optimize=1	optimize=2	optimize =max	optimize=1 speed	optimize=2 speed	optimize =max speed
main()	0xAC	0x80	0x76	0x76	0x80	0x76	0x76
sort()	0x92	0x47	0x51	0x53	0x4A	0x5D	0x5F

* By RX-family compiler V2.05.00

Please refer to the compiler user's manual for details of the optimization level of the RX-family 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];
    }
}
```

Website and Support

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

Inquiries
http://www.renesas.com/contact/

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Oct. 1, 2009	--	Initial edition
2.00	Nov. 30, 2016	--	Revised the destination to CS+ and CC-RX

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141