RX Development Environment Migration Guide

Migration from SuperH Family to RX Family (IDE ed.)
(High-performance Embedded Workshop to CS+)

REJ06J0102-0200
Rev.2.00
Apr 20, 2017

Introduction

This document explains how to migrate sample projects created by using SuperH, to RX.

Contents

1. Introduction

This document explains the precautions to take when migrating projects created for the SuperH-family to RX, and how to perform migration, based actual sample workspace usage.

Note that the options and other version-dependent information used in this document are based on version 9.04 of the SuperH-family C/C++ compiler, and version 2.06 of the RX-family compiler.

1.1 Dependencies on processing in C

A processing dependency is part of a program that lacks compatibility due to differences in behavior specific to certain hardware or compilers.

The C specification contains parts for which the behavior of code can be decided by each process, and parts within the SuperH-family C/C++ compiler and RX-family C/C++ compiler exist for which processing dependencies differ.

As such, even for the same C source program, the options for RX-family C/C++ compilers need to be set appropriately, to correctly handle these differences in processing dependencies.

2. Functionality Requiring Care during Migration

The SuperH-family and RX-family compilers contain parts for which the specification for processing dependencies differs under the default options. These options need to be specified explicitly to handle the differences in specification. This chapter explains the options and source program code that require special care during migration from the SuperH-family to the RX family.

2.1 Options

This chapter explains the options that require special care for RX family migration. The following table lists these options:

Table 2-1 List of options

No	Functionality	H8 option	RX option	Reference
1	Sign specification for the char type	--	signed_char	2.1.1
2	Size specification for enum	auto_enum	auto_enum	2.1.2
3	Size specification for the double type	double=float	dbl_size	2.1.3
4	Endian specification	endian	endian	2.1.4
5	Sign specification for bit field members	--	signed_bitfield	2.1.5
6	Allocation order specification for bit field members	bit_order	bit_order	2.1.6
7	Allocation specification for structures	pack	pack	*2.1.7*

2.1.1 Sign specification for the char type

With the SuperH-family compiler, char types without a specified sign are handled as signed char types, whereas the RX-family compiler handles them as unsigned char types by default.

When migrating a SuperH-family source program created assuming that char types are signed char types to the RX family, specify the "signed_char" option for the RX-family compiler.

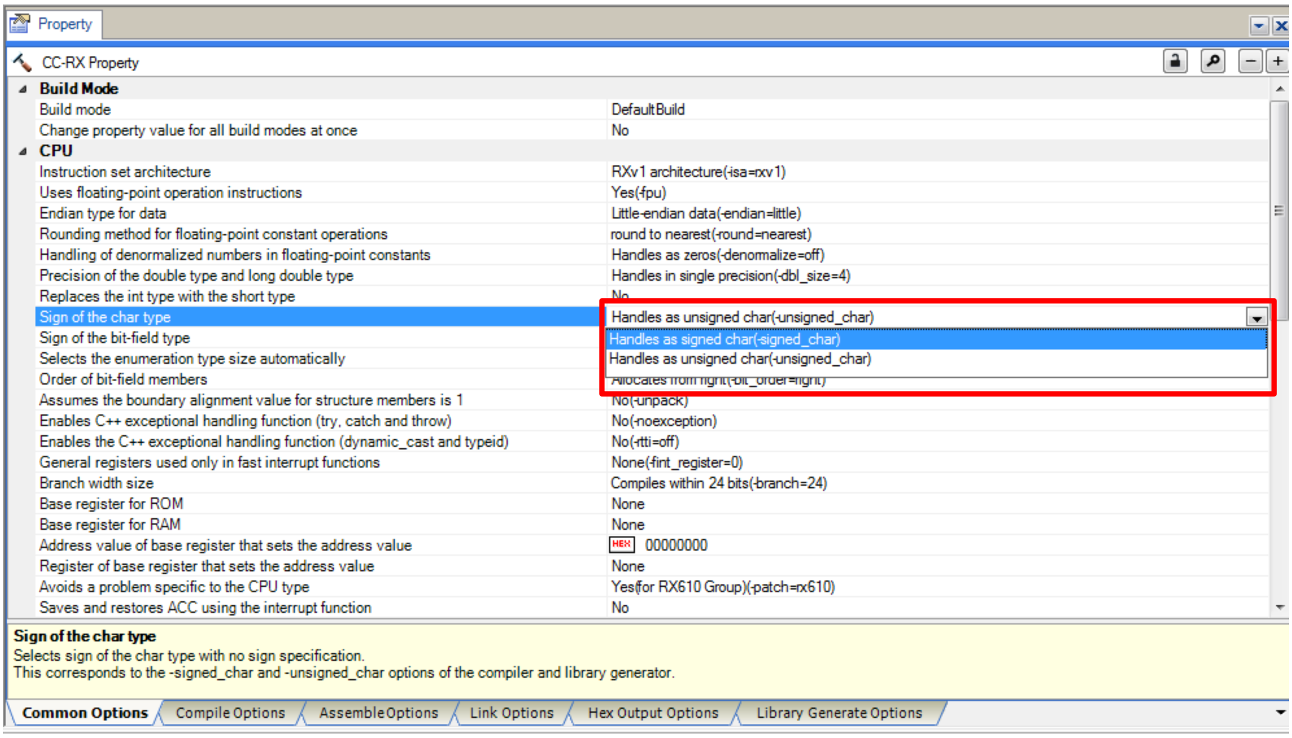
Format

signed_char

<u>unsigned_char</u> : unsigned_char by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	Little-endian data(-endian=little)
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denormalize=off)
Precision of the double type and long double type	Handles in single precision(-dbl_size=4)
Replaces the int type with the short type	No
Sign of the char type	Handles as unsigned char(-unsigned_char) Handles as signed char(-signed_char) Handles as unsigned char(-unsigned_char)
Sign of the bit-field type	
Selects the enumeration type size automatically	
Order of bit-field members	Allocates from right(-bit_order=right)
Assumes the boundary alignment value for structure members is 1	No(-unpack)
Enables C++ exceptional handling function (try, catch and throw)	No(-noexception)
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Sign of the char type
Selects sign of the char type with no sign specification.
This corresponds to the -signed_char and -unsigned_char options of the compiler and library generator.

Figure 2-1

2.1.2 Size specification for enum

When the "auto_enum" option was specified for the SuperH-family compiler, and data for enumeration types declared as enum is the smallest type stored in which enumeration values are stored, specify the "auto_enum" option in the RX-family compiler when migrating to the RX family.

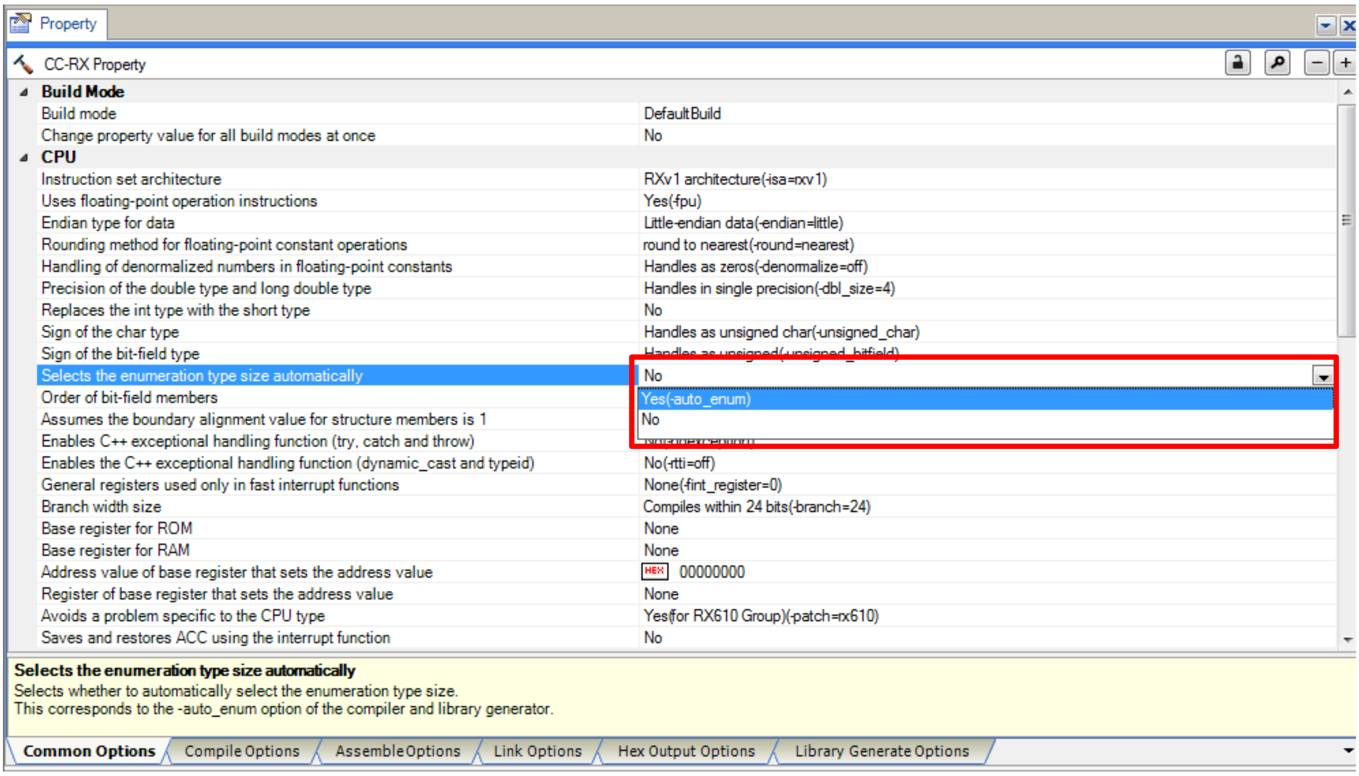
If the "auto_enum" option is not specified for the RX-family compiler, the signed long type is used as the enumeration type size.

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.



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	Little-endian data(-endian=little)
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denormalize=off)
Precision of the double type and long double type	Handles in single precision(-dbl_size=4)
Replaces the int type with the short type	No
Sign of the char type	Handles as unsigned char(-unsigned_char)
Sign of the bit-field type	Handles as unsigned(-unsigned_bitfield)
Selects the enumeration type size automatically	No Yes(-auto_enum) No
Order of bit-field members	
Assumes the boundary alignment value for structure members is 1	
Enables C++ exceptional handling function (try, catch and throw)	
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Selects the enumeration type size automatically
Selects whether to automatically select the enumeration type size.
This corresponds to the -auto_enum option of the compiler and library generator.

Figure 2-2

2.1.3 Specifying the size of double type

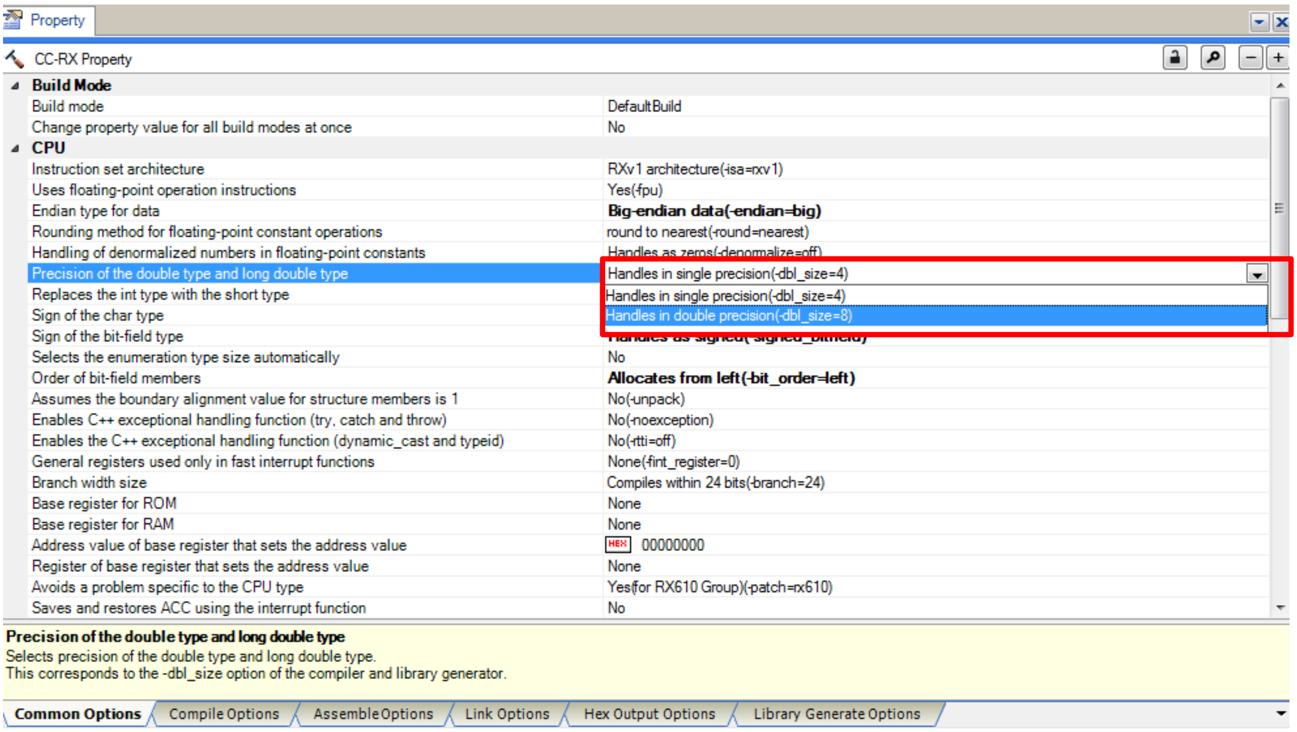
With H8-family compilers, the size of the double type is 8 bytes, whereas with RX-family compilers, the size of the double type is four bytes in default. To migrate to RX a program created in H8 based on the requirement that the size of the double type is 8 bytes, specify the “dbl_size=8” option.

Format

dbl_size ={4|8} : 4 by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	**Big-endian data(-endian=big)**
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denormalize=off)
Precision of the double type and long double type	Handles in single precision(-dbl_size=4) Handles in single precision(-dbl_size=4) Handles in double precision(-dbl_size=8)
Replaces the int type with the short type	
Sign of the char type	
Sign of the bit-field type	**Handles as signed(-signed_bitfield)**
Selects the enumeration type size automatically	No
Order of bit-field members	**Allocates from left(-bit_order=left)**
Assumes the boundary alignment value for structure members is 1	No(-unpack)
Enables C++ exceptional handling function (try, catch and throw)	No(-noexception)
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Precision of the double type and long double type
Selects precision of the double type and long double type.
This corresponds to the -dbl_size option of the compiler and library generator.

Figure 2-3

Note:

When "double=float" is specified for the SuperH-family compiler, the size of the long double type is 8 bytes, but when "dbl_size=4" is specified for the RX-family compiler, the size of the long double type is 4 bytes.

2.1.4 Endian specification

The data byte order for the SuperH-family compiler is big-endian by the default setting for the ENdian option, whereas for the RX-family compiler, it is little-endian by the default setting for the endian option.

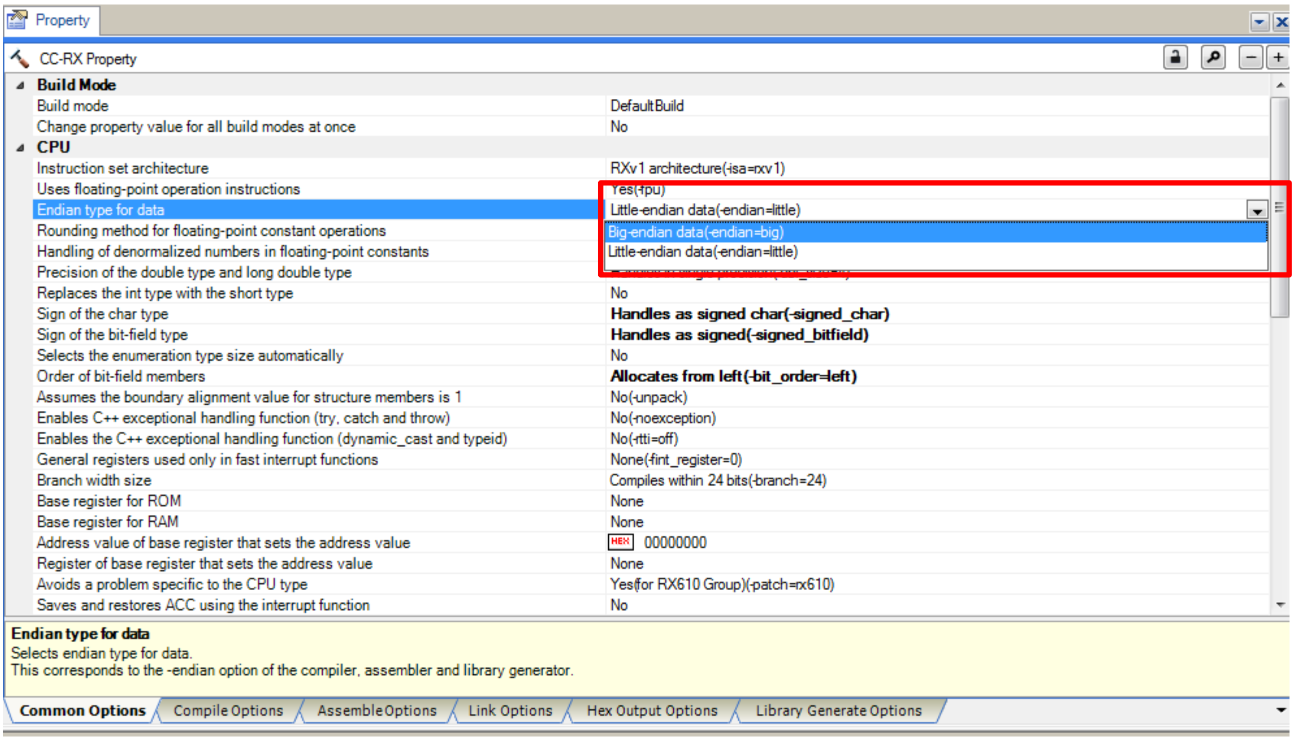
When migrating a SuperH-family source program created assuming that data byte order is big-endian to the RX family, specify the "endian=big" option for the RX-family compiler.

Format

endian={ big | little } : little by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	Little-endian data(-endian=little) Big-endian data(-endian=big) Little-endian data(-endian=little)
Rounding method for floating-point constant operations	
Handling of denormalized numbers in floating-point constants	
Precision of the double type and long double type	
Replaces the int type with the short type	No
Sign of the char type	**Handles as signed char(-signed_char)**
Sign of the bit-field type	**Handles as signed(-signed_bitfield)**
Selects the enumeration type size automatically	No
Order of bit-field members	**Allocates from left(-bit_order=left)**
Assumes the boundary alignment value for structure members is 1	No(-unpack)
Enables C++ exceptional handling function (try, catch and throw)	No(-noexception)
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Endian type for data
Selects endian type for data.
This corresponds to the -endian option of the compiler, assembler and library generator.

Figure 2-4

2.1.5 Sign specification for bit field members

For SuperH-family compilers, unsigned bit field members are handled as signed types, whereas RX-family compilers handle them as unsigned types by default.

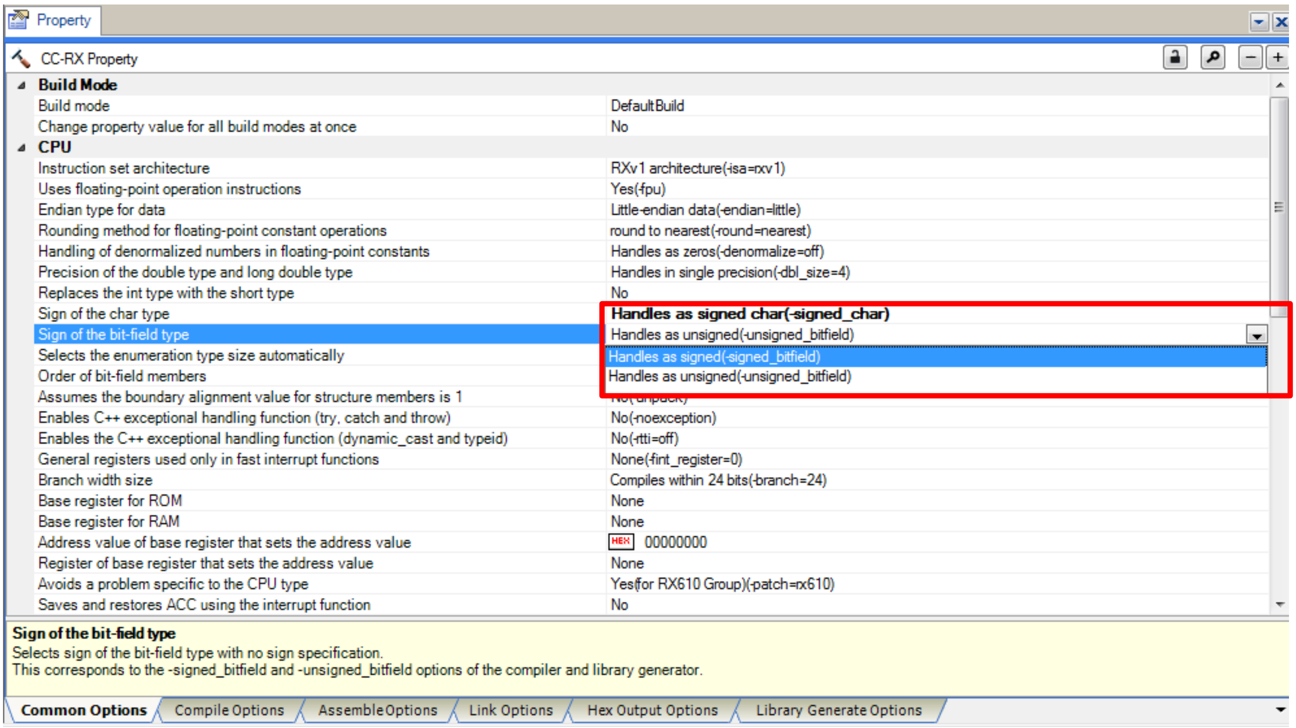
When migrating a SuperH-family source program created assuming that unsigned bit field members are signed types to the RX family, specify the "signed_bitfield" option for the RX-family compiler.

Format

signed_bitfield
<u>unsigned_bitfield</u> : unsigned_bitfield by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	Little-endian data(-endian=little)
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denormalize=off)
Precision of the double type and long double type	Handles in single precision(-dbl_size=4)
Replaces the int type with the short type	No
Sign of the char type	**Handles as signed char(-signed_char)**
Sign of the bit-field type	Handles as unsigned(-unsigned_bitfield) Handles as signed(-signed_bitfield) Handles as unsigned(-unsigned_bitfield)
Selects the enumeration type size automatically	
Order of bit-field members	
Assumes the boundary alignment value for structure members is 1	No(-unpack)
Enables C++ exceptional handling function (try, catch and throw)	No(-noexception)
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Sign of the bit-field type
Selects sign of the bit-field type with no sign specification.
This corresponds to the -signed_bitfield and -unsigned_bitfield options of the compiler and library generator.

Figure 2-5

2.1.6 Allocation order specification for bit field members

For SuperH-family compilers, bit field members are allocated from the highest bit, whereas for RX-family compiler, they are allocated for the lowest bit by default.

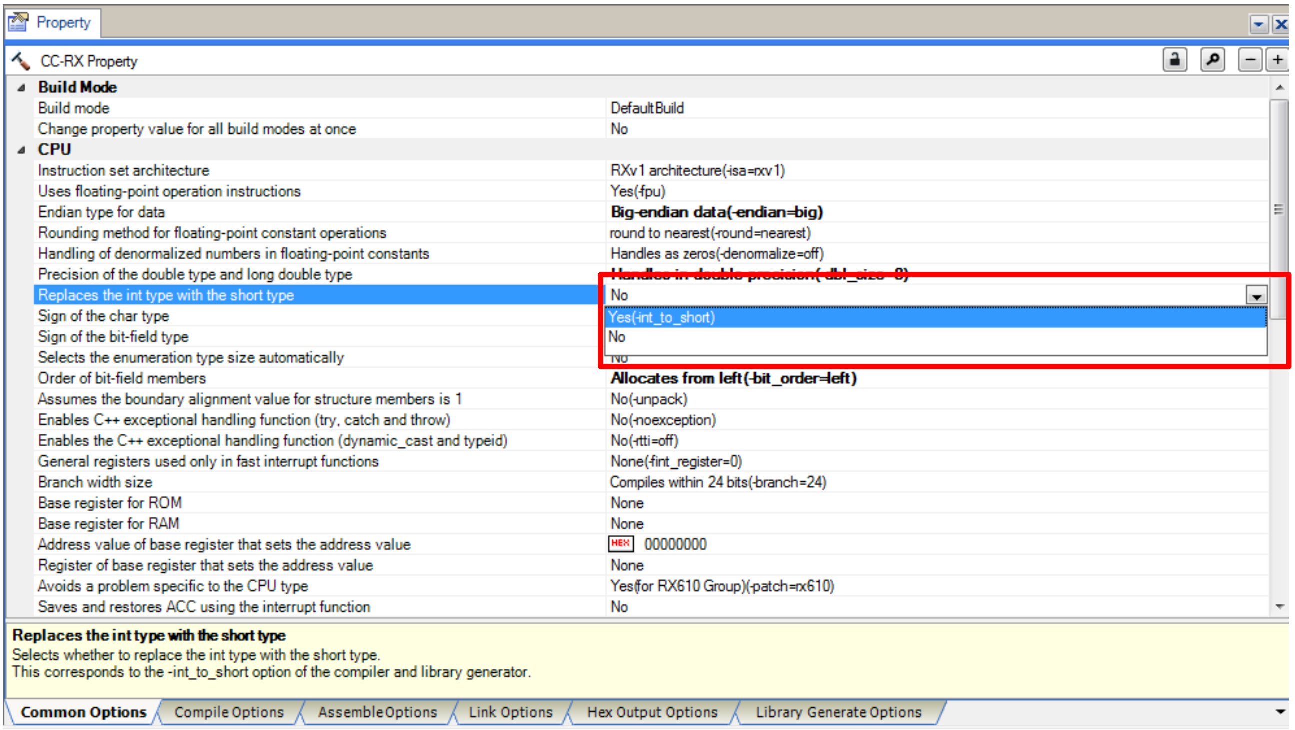
When migrating a SuperH-family source program created assuming that bit field members are allocated from the highest bit to the RX family, specify the "bit_order=left" option for the RX-family compiler.

Format

bit_order={ left | <u>right</u> } : right by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	**Big-endian data(-endian=big)**
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denomalize=off)
Precision of the double type and long double type	**Handles in double precision(-dbl_size=8)**
Replaces the int type with the short type	No Yes(-int_to_short) No
Sign of the char type	
Sign of the bit-field type	
Selects the enumeration type size automatically	No
Order of bit-field members	**Allocates from left(-bit_order=left)**
Assumes the boundary alignment value for structure members is 1	No(-unpack)
Enables C++ exceptional handling function (try, catch and throw)	No(-noexception)
Enables the C++ exceptional handling function (dynamic_cast and typeid)	No(-rtti=off)
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None
Avoids a problem specific to the CPU type	Yes(for RX610 Group)(-patch=rx610)
Saves and restores ACC using the interrupt function	No

Replaces the int type with the short type
Selects whether to replace the int type with the short type.
This corresponds to the -int_to_short option of the compiler and library generator.

Figure 2-6

2.1.7 Allocation order specification for bit field members

When the "pack=1" option is specified for the SuperH-family compiler to set the structure alignment count to 1, specify the "pack" option for the RX-family compiler when migrating to the RX family.

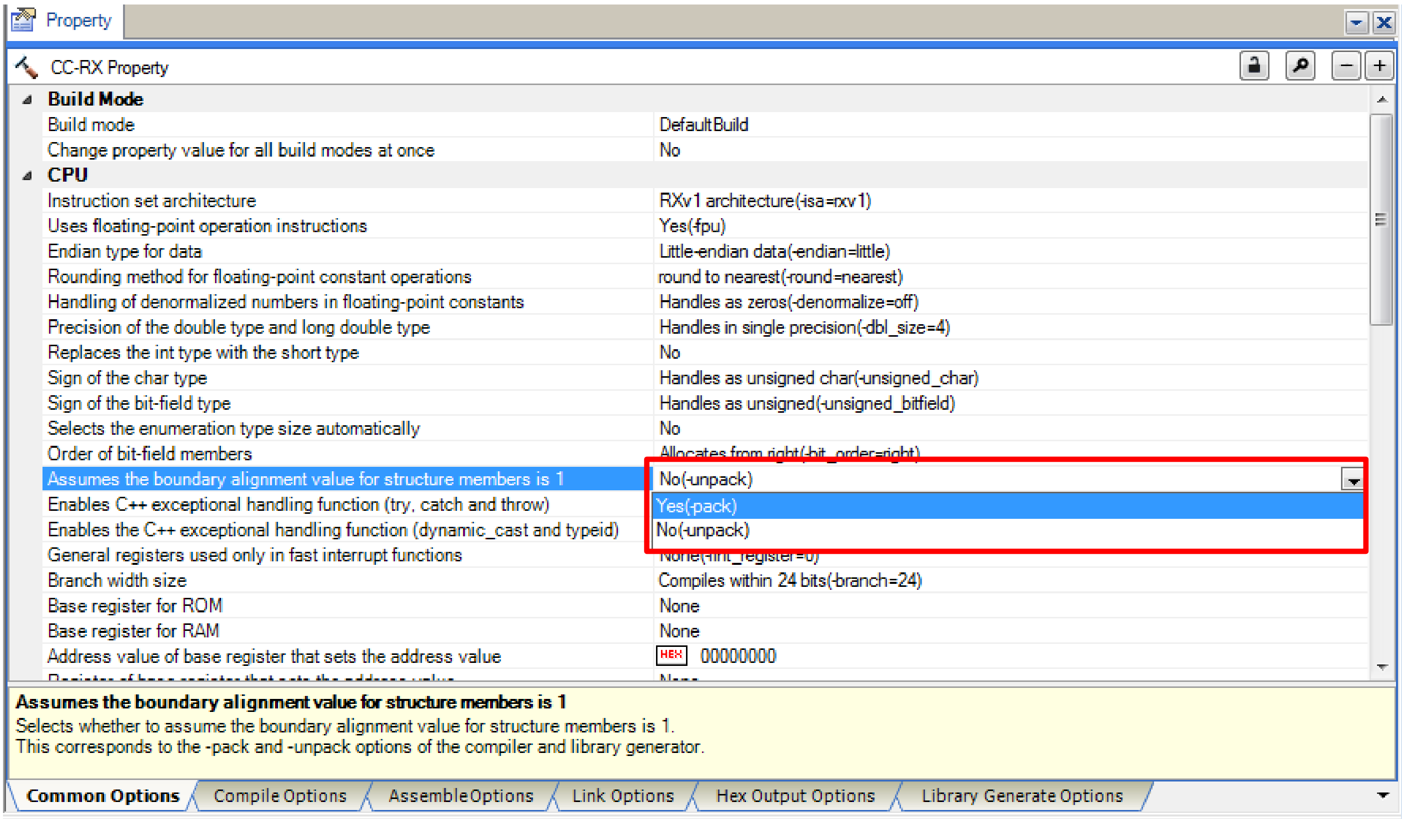
Format

pack

unpack : unpack by default

[How to specify this option in CS+]

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



Build Mode	
Build mode	DefaultBuild
Change property value for all build modes at once	No
CPU	
Instruction set architecture	RXv1 architecture(-isa=rxv1)
Uses floating-point operation instructions	Yes(-fpu)
Endian type for data	Little-endian data(-endian=little)
Rounding method for floating-point constant operations	round to nearest(-round=nearest)
Handling of denormalized numbers in floating-point constants	Handles as zeros(-denormalize=off)
Precision of the double type and long double type	Handles in single precision(-dbl_size=4)
Replaces the int type with the short type	No
Sign of the char type	Handles as unsigned char(-unsigned_char)
Sign of the bit-field type	Handles as unsigned(-unsigned_bitfield)
Selects the enumeration type size automatically	No
Order of bit-field members	Allocates from right(-bit_order=right)

Assumes the boundary alignment value for structure members is 1	No(-unpack)
	Yes(-pack)
	No(-unpack)

Enables C++ exceptional handling function (try, catch and throw)	
Enables the C++ exceptional handling function (dynamic_cast and typeid)	
General registers used only in fast interrupt functions	None(-fint_register=0)
Branch width size	Compiles within 24 bits(-branch=24)
Base register for ROM	None
Base register for RAM	None
Address value of base register that sets the address value	HEX 00000000
Register of base register that sets the address value	None

Assumes the boundary alignment value for structure members is 1
Selects whether to assume the boundary alignment value for structure members is 1.
This corresponds to the -pack and -unpack options of the compiler and library generator.

Figure 2-7

2.2 Language specification

This chapter explains the language specifications for which changes are needed during migration to RX.

Table 2-2 List of language specifications

No	Functionality	Reference
1	Signs for char types	2.2.1
2	Sizes for double types	2.2.2
3	Endianness	2.2.3
4	Allocation order for bit fields	2.2.4
5	Signs for bit fields	2.2.5

2.2.1 Signs for char types

For SuperH-family compilers, unsigned char types are handled as signed char types, whereas RX-family compilers handle them as unsigned char types.

SuperH-family source programs created assuming that char types are signed char types may not operate correctly when migrated to RX.

Example: Differing operation due to presence of a char type sign.

```
Source code


char a = -1;


void main(void)
{
  if (a < 0) {
    // The char type is signed, 'a' is evaluated as negative, and the condition is satisfied (SuperH)
  } else {
    // The char type is unsigned, 'a' is evaluated as positive, and the condition is not satisfied (RX)
  }
}
```

When migrating a source program created assuming that char types are signed char types to RX, specify the "signed_char" option.

For details about specifying this option, see *2.1.1 Specifying sign for the char type*.

2.2.2 Sizes for double types

For SuperH-family compilers, the size of a double type is 8 bytes, whereas for RX-family compilers, the size of a double type is 4 bytes.

SuperH-family source programs created assuming that the size of a double type is 8 bytes may not operate correctly when migrated to RX.

Example: Differing operation due to difference in double type size

```
Source code


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

void main(void)
{
  d1 = d1 * d1; // d1 * d1 overflows when the double type size is 4 bytes
  d2 = d2 * d2; // d2 * d2 overflows when the double type size is 4 bytes
  if (d1 > d2) {
     // Size is compared correctly when the double type size is 8 bytes (SuperH)
  } else {
     // Both d1 and d2 overflow when the double type size is 4 bytes
     // so that size comparison is not satisfied (RX)
  }
}
```

When migrating a source program created assuming that the size of a double type is 8 bytes to RX, specify the "dbl_size=8" option.

For details about specifying this option, see *2.1.3 Specifying bit-field member allocation.*

2.2.3 Endianness

The data byte order for the SuperH-family compiler is big-endian by the default setting for the ENdian option, whereas for the RX-family compiler, it is little-endian by the default setting for the endian option.

When a SuperH-family source program created based on the assumption that the data byte order is big-endian is migrated to the RX family, it may not operate correctly.

Example: Differing operation due to difference in endianness

Source code

```
typedef union{
  short data1;
  struct {
     unsigned char upper;
     unsigned char lower;
 } data2;
} UN;

UN u = { 0x7f6f };

void main(void)
{
 if (u.data2.upper == 0x7f && u.data2.lower == 0x6f) {
   // When the data byte order is big-endian (SuperH)
 } else {
   // When the data byte order is little-endian (RX)
 }
}
```

When migrating a source program created assuming that the byte order for data is big-endian to RX, specify the "endian=big" option.

For details about specifying this option, see *2.1.4 Specifying endian*.

2.2.4 Allocation order for bit fields

For SuperH-family compilers, bit field members are allocated from the highest bit, whereas for RX-family compilers, they are allocated from the lowest bit.

SuperH-family source programs created assuming that bit field members are allocated from the highest bit may not operate correctly when migrated to RX.

Example: Differing operation due to differences in the allocation order for bit fields

Source code

```
union {
  unsigned char c1;
  struct {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
  } b;
} un;

void bit_order(void)
{
  un.c1 = 0xc0;
  if ((un.b.b0 == 1) && (un.b.b1 == 1) &&
     (un.b.b2 == 0) && (un.b.b3 == 0)) {
     // When bit field members are allocated from the highest bit (SuperH)
  } else {
     // When bit field members are allocated from the lowest bit (RX)
  }
}
```

SuperH allocation (left)

1	1	0	0	0	0	0	0
b0	b1	b2	b3				

The highest bits are allocated, so the set value can be read as b0, b1

RX allocation (right)

1	1	0	0	0	0	0	0
				b3	b2	b1	b0

The lowest bits are allocated, so the set value cannot be read

When migrating a source program created assuming that bit field members are allocated from the highest bit to RX, specify the "bit_order=left" option.

For details about specifying this option, see 2.1.6 Correspondence of int type size to difference.

2.2.5 Signs for bit fields

For SuperH-family compilers, unsigned bit field members are handled as signed types, whereas for RX-family compilers, they are handled as unsigned types.

SuperH-family source programs created assuming that unsigned bit field members are signed types may not operate correctly when migrated to RX.

Example: Differing operation due to presence of sign for bit field members

Source code

```
struct S {
   int a : 15;
} s = { -1 };

void main(void)
{
  if (s.a < 0) {
    // The bit field member is signed, 's.a' is evaluated as negative
    //so the condition is satisfied (SuperH)
  } else {
    // The bit field member is unsigned, 's.a' is evaluated as positive
    //so the condition is not satisfied (RX)
  }
}
```

When migrating a source program created assuming that unsigned bit field members are signed types to RX, specify the "signed_bitfield" option.

For details about specifying this option, see *2.1.5 Sign specification for bit field members*.

2.2.6 Extended language specification

(1) Support for #pragma pack

When #pragma pack is used for the SuperH-family compiler, the specification for the RX-family compiler needs to be changed.

Table 2-3 List of language specifications

SuperH	RX	Note
#pragma pack 1	#pragma pack	1 is used for the alignment count
#pragma pack 4	#pragma unpack	The default alignment is used
#pragma unpack	#pragma packoption	The pack option is used

(2) Support for evenaccess

For the SuperH-family compiler, variables declared as volatile are guaranteed to be accessed with the size of their type.

However, for the RX-family compiler, evenaccess needs to be used with the following format in order to guarantee access with the size of the type.

```
__evenaccess <type-specifier> <variable-name>

<type-specifier> __evenaccess <variable-name>
```

2.2.7 Predefined macros

Keep in mind that the predefined macros defined when options are specified differ between the SuperH-family compiler and RX-family compiler.

To make these options correspond, the changes shown in the following tables need to be made for the predefined macro names of the RX-family compiler.

Table 2-4 Predefined macros for SuperH

Option	Predefined macros
endian=big	_BIG
endian=little	_LIT
double=float	_FLT __FLT__
denormalize=on	_DON
round=nearest	_RON

Table 2-5 Predefined macros for RX

Option	Predefined macros
endian=big	__BIG
endian=little	__LIT
double=float	__DBL4
denormalize=on	__DON
round=nearest	__RON

3. Migration Sample Project

This chapter explains how to migrate the SuperH sample project whose operation can be checked in the simulator/debugger, to RX.

3.1 List of main processing files

The 'SH_Sample' SuperH sample projects can be broadly divided into those that perform pre- and post-processing such as for initialization, and those that perform main processing.

The following table lists the files that comprise main processing.

Table 3-1 List of main processing files

No	Functionality	File name	Reference
1	Signs for char types	SH_sign_char.c	3.2.5(1)
2	Sign for bit field members	SH_sign_bit_field.c	3.2.5(2)
3	Allocation for bit field members	SH_bit_order.c	3.2.5(3)
4	Endianness	SH_endian.c	3.2.5(4)
5	Size for the double type	SH_double_size.c	3.2.5(5)
6	main function	SH_Sample.c	—

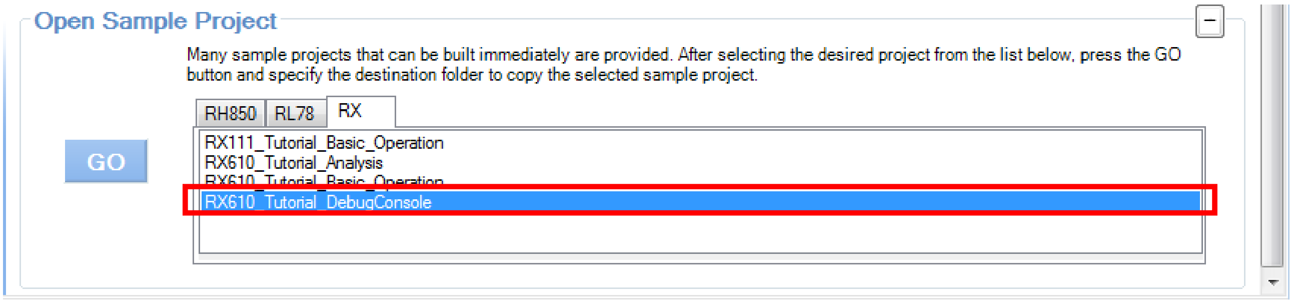
3.2 Migrating the SuperH sample project to RX

3.2.1 Creating an RX project

Create a new RX project workspace to which migrate the SuperH sample projects.

(1) Import Sample Project

Select “RX” tab in [Open Sample Project] and select [RX610_Tutorial_DebugConsole].



Open Sample Project

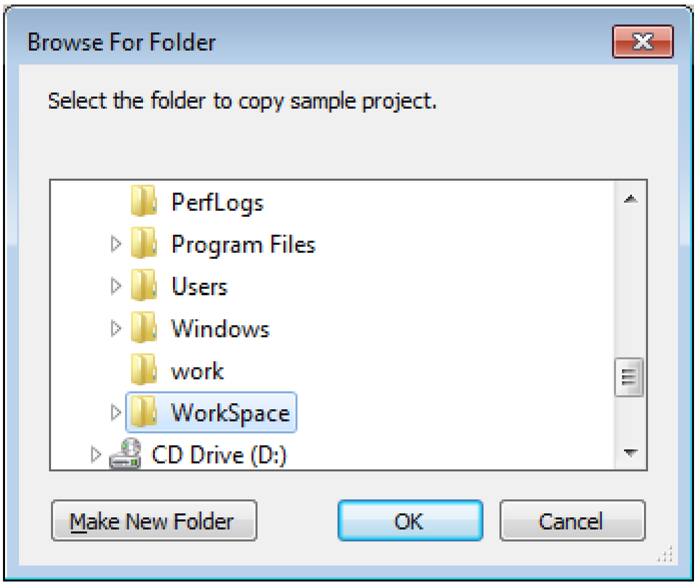
Many sample projects that can be built immediately are provided. After selecting the desired project from the list below, press the GO button and specify the destination folder to copy the selected sample project.

RX111_Tutorial_Basic_Operation
RX610_Tutorial_Analysis
RX610_Tutorial_Basic_Operation
RX610_Tutorial_DebugConsole

Figure 3-1

(2) Select where to copy sample project.

Select folder to copy sample project.



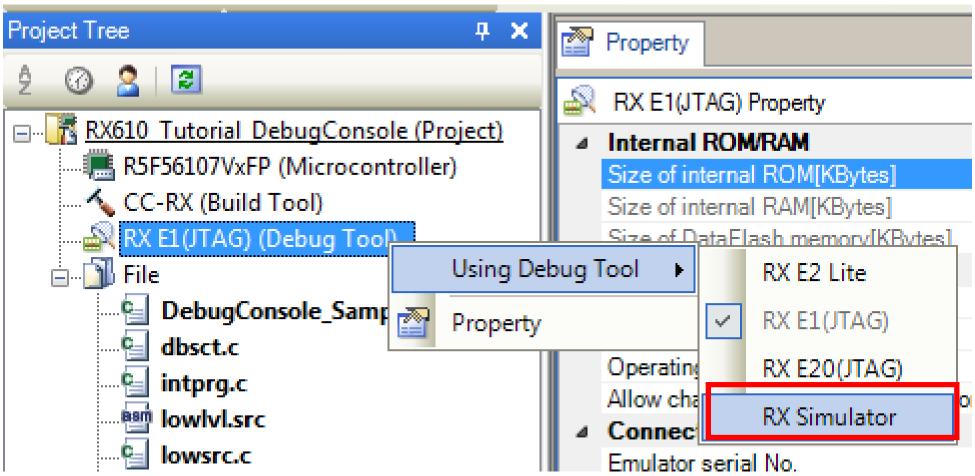
Browse For Folder

Select the folder to copy sample project.

Figure 3-2

(3) Select debug tool

Select [Using Debug Tool], and then “RX Simulator”.



Project Tree

- RX610_Tutorial_DebugConsole (Project)
  - R5F56107VxFP (Microcontroller)
  - CC-RX (Build Tool)
  - RX E1(JTAG) (Debug Tool)
  - File
    - DebugConsole_Samp
    - dbsct.c
    - intprg.c
    - lowlvl.src
    - lowsrc.c

Property

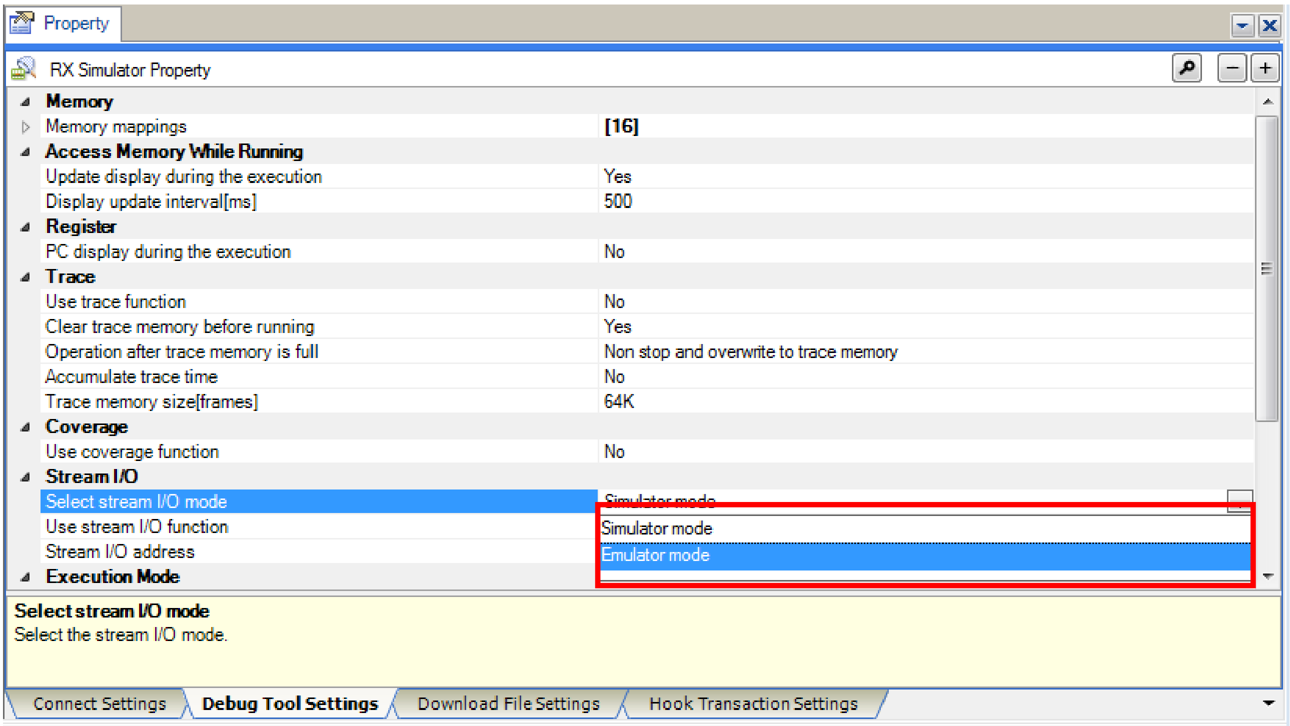
RX E1(JTAG) Property

Internal ROM/RAM
Size of internal ROM[KBytes]
Size of internal RAM[KBytes]
Size of DataFlash memory[KBytes]
Operatin
Allow cha
Connec
Emulator serial No.

Figure 3-3

(4) Select Stream I/O mode

Perform the following settings in the [Stream I/O] category of the [Debug Tool Settings] page.



Property

RX Simulator Property

Memory	
Memory mappings	**[16]**
Access Memory While Running	
Update display during the execution	Yes
Display update interval[ms]	500
Register	
PC display during the execution	No
Trace	
Use trace function	No
Clear trace memory before running	Yes
Operation after trace memory is full	Non stop and overwrite to trace memory
Accumulate trace time	No
Trace memory size[frames]	64K
Coverage	
Use coverage function	No
Stream I/O	
Select stream I/O mode	Simulator mode
Use stream I/O function	Simulator mode Emulator mode
Stream I/O address	
Execution Mode	

Select stream I/O mode
Select the stream I/O mode.

Figure 3-4

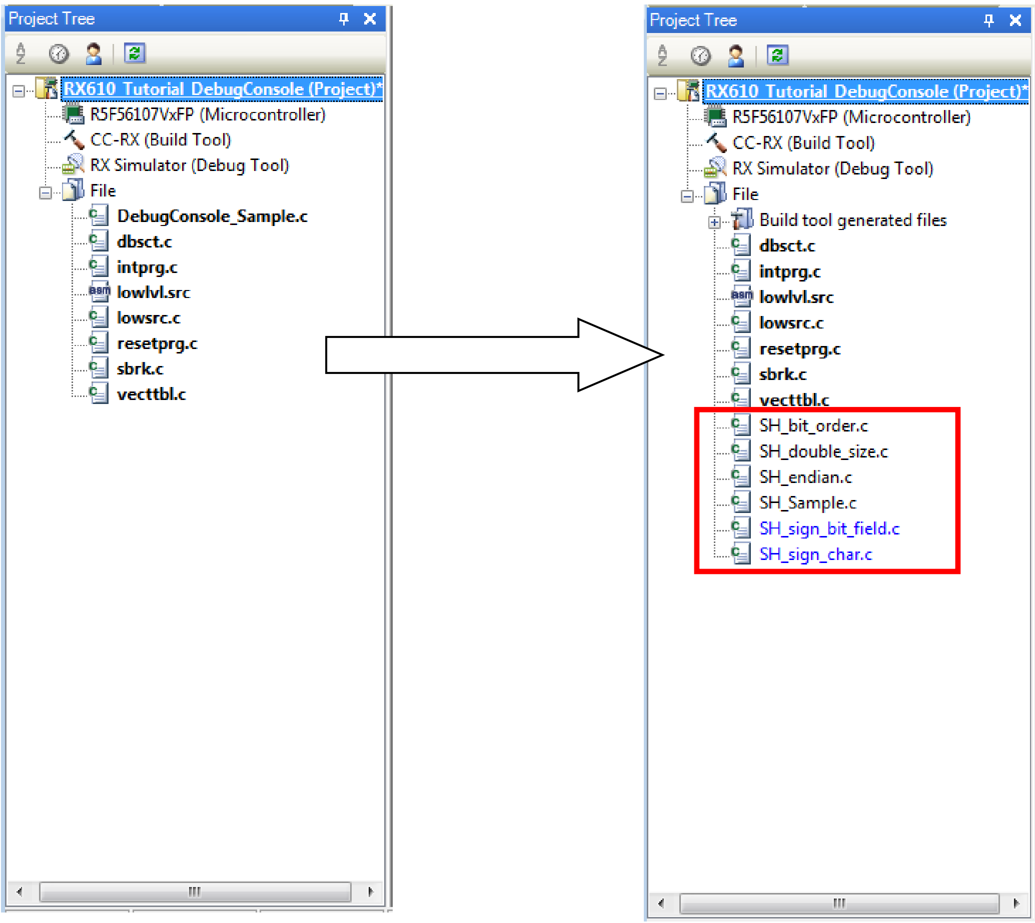
3.2.2 Migrating main processing source files

Copy, and add to the created RX project, the files comprising main processing for the SuperH sample project explained in *3.1 SuperH sample project overview*.

(1) Copy the files from the SuperH sample project folder

Copy the six files explained in *3.1 SuperH sample project overview*.

[Before copy] **[After copy]**



[Before copy]

```
Project Tree
RX610_Tutorial_DebugConsole (Project)*
  R5F56107VxFP (Microcontroller)
  CC-RX (Build Tool)
  RX Simulator (Debug Tool)
  File
    DebugConsole_Sample.c
    dbsct.c
    intprg.c
    lowlvl.src
    lowsrc.c
    resetprg.c
    sbrk.c
    vecttbl.c
```

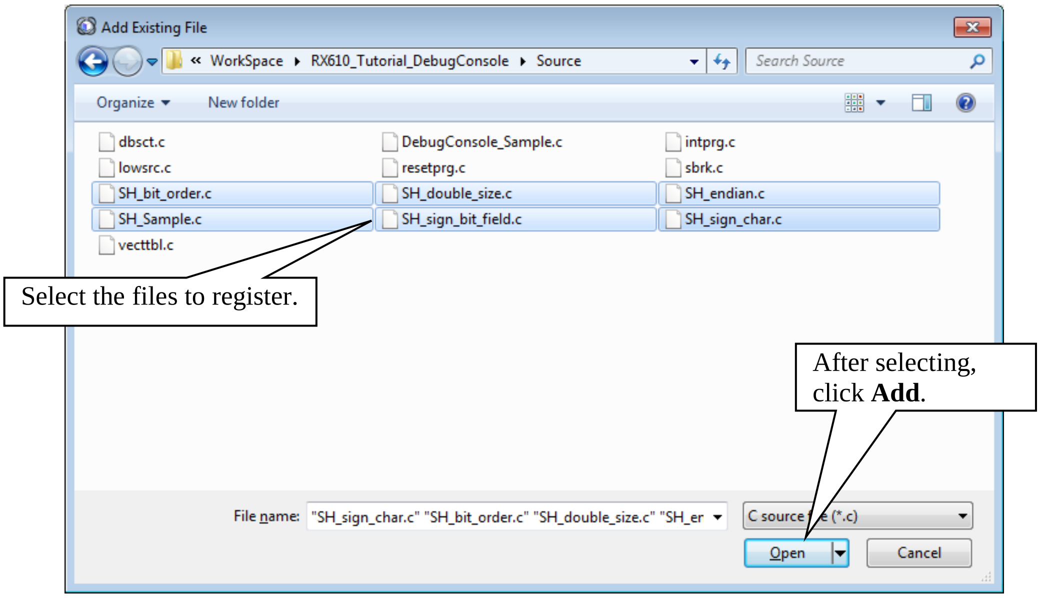
[After copy]

```
Project Tree
RX610_Tutorial_DebugConsole (Project)*
  R5F56107VxFP (Microcontroller)
  CC-RX (Build Tool)
  RX Simulator (Debug Tool)
  File
    Build tool generated files
    dbsct.c
    intprg.c
    lowlvl.src
    lowsrc.c
    resetprg.c
    sbrk.c
    vecttbl.c
    SH_bit_order.c
    SH_double_size.c
    SH_endian.c
    SH_Sample.c
    SH_sign_bit_field.c
    SH_sign_char.c
```

Figure 3-5

(2)Add the copied files to the project

Perform the following settings in the dialog box displayed by choosing [Project →Add → Add Existing File] in CS+.



Select the files to register.

After selecting, click **Add**.

Figure 3-6

(3) Remove any unnecessary files

Since the ' DebugConsole_Sample.c ' main function file in RX sample project is no longer needed, remove it (since the main function file has been copied from the SuperH project).

Select [DebugConsole_Sample.c] in project tree, and select [Remove from Project].

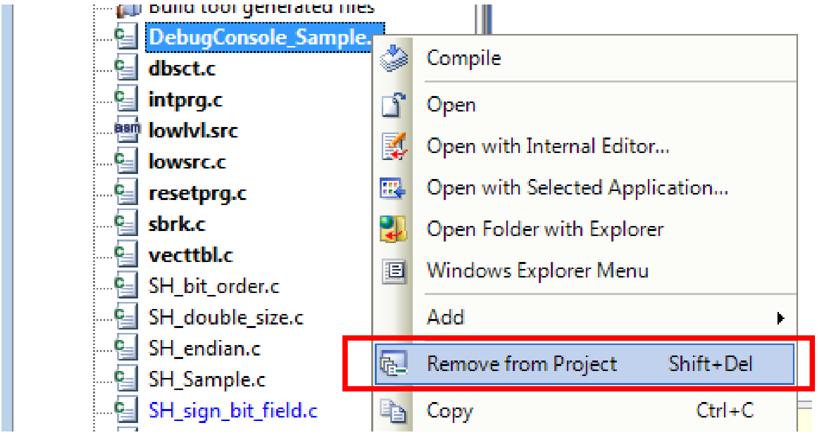


Figure 3-7

3.2.3 Performing a build

Build the RX project for which the main processing files have been copied and registered.

To start a build, choose [Build], and then [Build Project] in CS+.

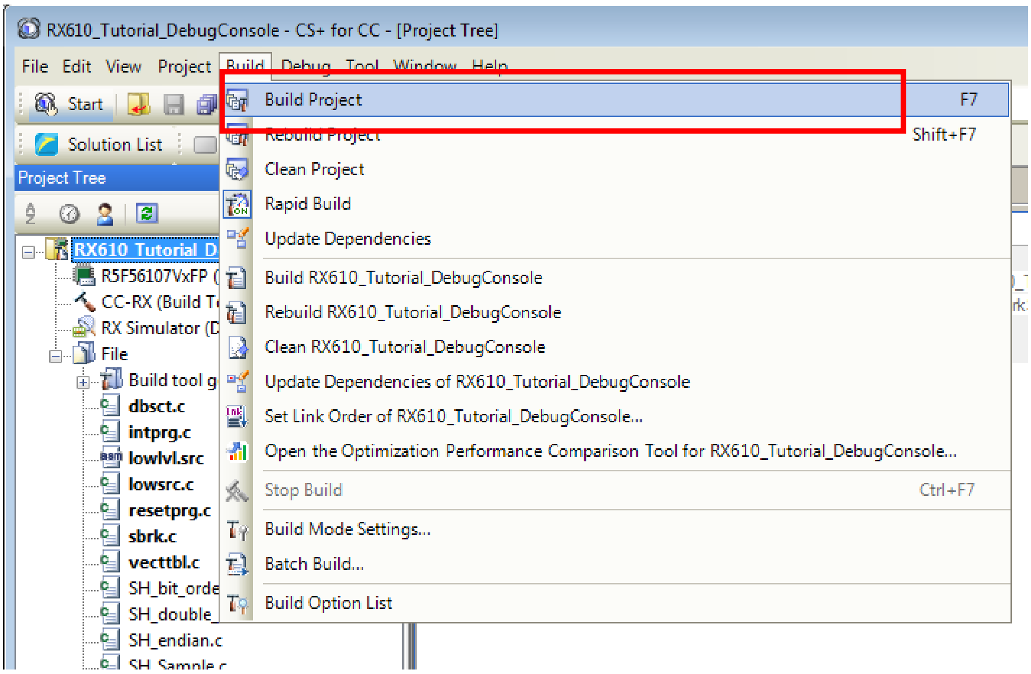

RX610_Tutorial_DebugConsole - CS+ for CC - [Project Tree]
File Edit View Project Build Debug Tool Window Help
Start
Solution List
Project Tree
RX610_Tutorial_D
R5F56107VxFP (
CC-RX (Build T
RX Simulator (D
File
Build tool g
dbsct.c
intprg.c
lowlvl.src
lowsrc.c
resetprg.c
sbrk.c
vecttbl.c
SH_bit_orde
SH_double_
SH_endian.c
SH_Sample.c
Build Project F7
Rebuild Project Shift+F7
Clean Project
Rapid Build
Update Dependencies
Build RX610_Tutorial_DebugConsole
Rebuild RX610_Tutorial_DebugConsole
Clean RX610_Tutorial_DebugConsole
Update Dependencies of RX610_Tutorial_DebugConsole
Set Link Order of RX610_Tutorial_DebugConsole...
Open the Optimization Performance Comparison Tool for RX610_Tutorial_DebugConsole...
Stop Build Ctrl+F7
Build Mode Settings...
Batch Build...
Build Option List

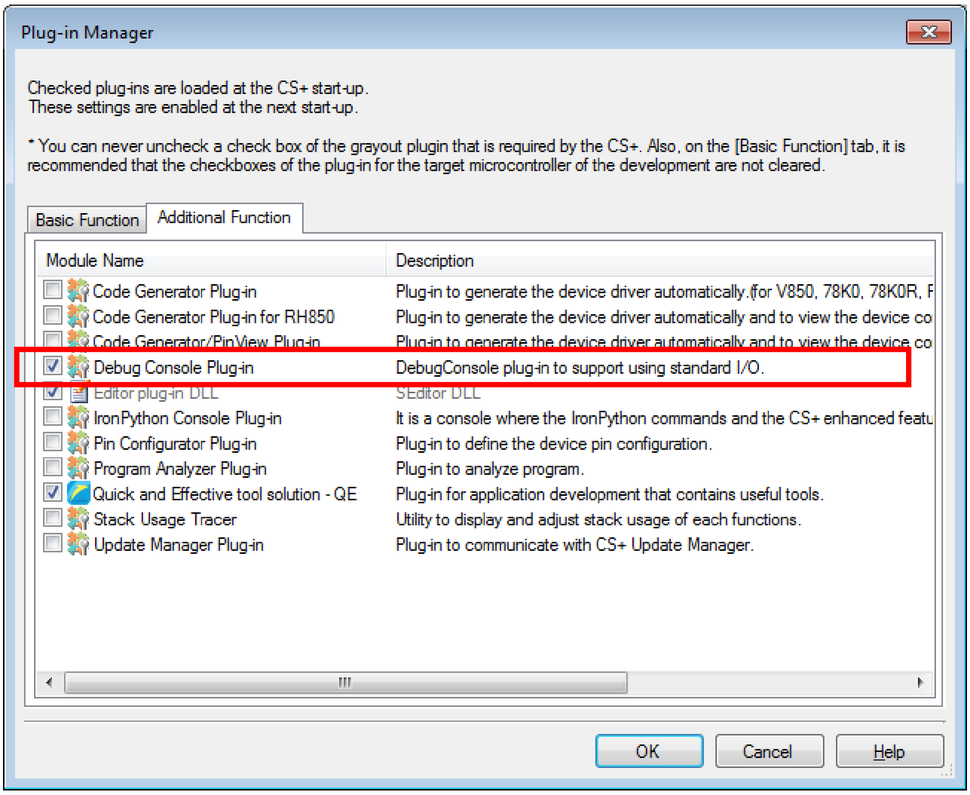
Figure 3-8

3.2.4 Executing the simulator

Execute the built RX project load module in the simulator.

(1) Setting up Debug Console

The execution results of the source program are output to the standard output.
Debug Console plug-in needs to be enabled to display the standard output.
Choose [tool], and then [Plug-in Manager] and select as follows from dialog in CS+.



Plug-in Manager

Checked plug-ins are loaded at the CS+ start-up.
These settings are enabled at the next start-up.

* You can never uncheck a check box of the grayout plugin that is required by the CS+. Also, on the [Basic Function] tab, it is recommended that the checkboxes of the plug-in for the target microcontroller of the development are not cleared.

Basic Function | Additional Function

Module Name	Description
□ Code Generator Plug-in	Plug-in to generate the device driver automatically.(for V850, 78K0, 78K0R, F
□ Code Generator Plug-in for RH850	Plug-in to generate the device driver automatically and to view the device co
□ Code Generator/PinView Plug-in	Plug-in to generate the device driver automatically and to view the device co
☑ Debug Console Plug-in	DebugConsole plug-in to support using standard I/O.
☑ Editor plug-in DLL	SEditor DLL
□ IronPython Console Plug-in	It is a console where the IronPython commands and the CS+ enhanced featu
□ Pin Configurator Plug-in	Plug-in to define the device pin configuration.
□ Program Analyzer Plug-in	Plug-in to analyze program.
☑ Quick and Effective tool solution - QE	Plug-in for application development that contains useful tools.
□ Stack Usage Tracer	Utility to display and adjust stack usage of each functions.
□ Update Manager Plug-in	Plug-in to communicate with CS+ Update Manager.

OK Cancel Help

Figure 3-9

(2) Download to Debug Tool

Select [Debug → Download] in CS+ to download load module to debug tool.

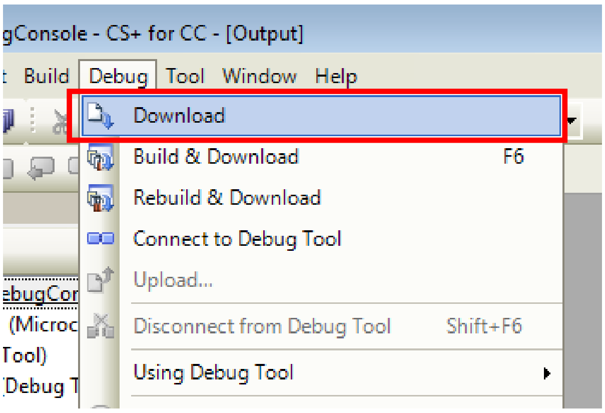
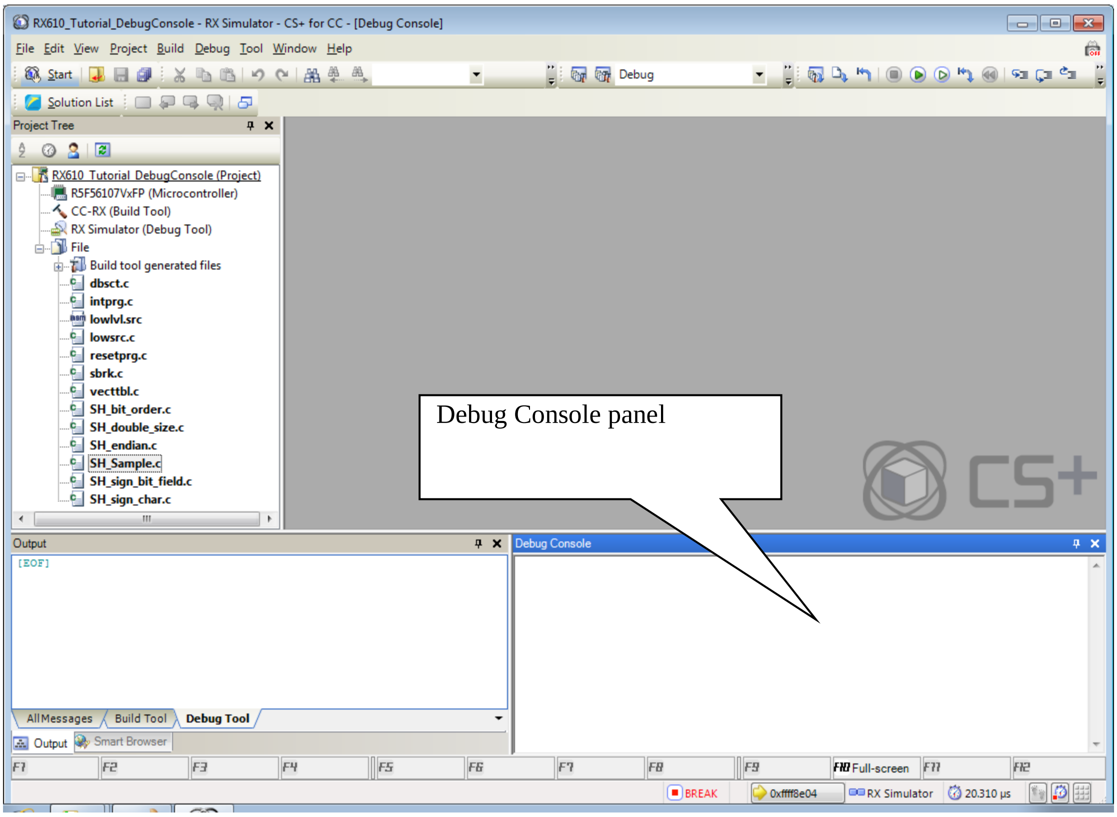


Figure 3-10

(3) Display Debug Console panel

Debug Console panel needs to be enabled to display the standard output.
Choose [View], and then **[**Debug Console] and select as follows from dialog in CS+ display Debug Console panel.



Debug Console panel

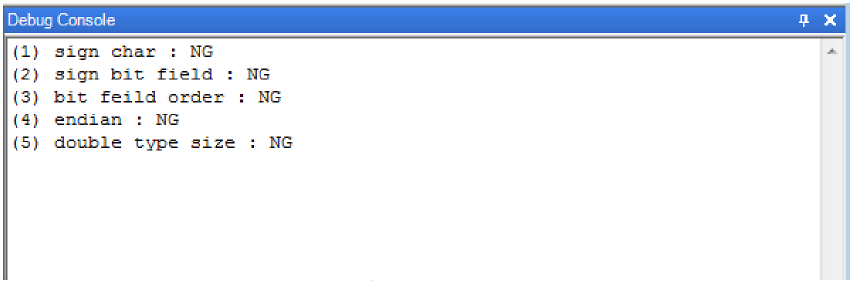
Figure 3-11

(4) Executing the simulator

Choose [Debug] and then [Execute post-reset] in CS+ to run the source program in the simulator, and display the standard output of the source program in the [Debug Console] panel.

Displayed output says "NG", and it means that the results are invalid.

<Debug Console>



```
(1) sign char : NG
(2) sign bit field : NG
(3) bit feild order : NG
(4) endian : NG
(5) double type size : NG
```

Figure 3-12

Table 3-2 I/O simulation output results

Item	OK	NG
(1)Char type without a specified sign	signed	unsigned
(2)Bit field members without a specified sign	signed	unsigned
(3) Bit field member allocation order	From the highest bit	From the lowest bit
(4) Endianness	big	little
(5) Size of the double type	8byte	4byte

3.2.5 Setting options

The simulator execution results are invalid due to differences in specifications for processing-related definitions between SuperH-family and RX-family compilers.

This chapter explains how to change the specified options for resolving the specification differences for processing-related definitions, using a RX-family project migrated from the SuperH-family as a sample.

(1) char signs

If the execution results of the "SH_sign_char.c" sample source program are "NG", this indicates a problem with the compatibility of the "unsigned_char" option specification.

For SuperH-family compilers, char types without a specified sign are handled as signed char types, whereas RX-family compilers handle them as unsigned char types.

Since the "SH_sign_char.c" sample source program was created assuming that char types without a specified sign are signed char types, if the "unsigned_char" option is specified, the operation results will differ from SuperH.

Sample Source Program : SH_sign_char.c

```
Source code

struct S {
  char a;
} s = { -1 };

void sign_char(void)
{
  printf("(1) sign char : ");

  if (s.a < 0) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

To migrate a source program created assuming that char types with a specified sign are signed char types to RX, specify the "signed_char" option.

For details about specifying this option, see *2.1.1 Sign specification for the char type*.

Also, change the options specified for the created RX project.

(2) Bit fields signs

If the execution results of the "SH_sign_bit_field.c" sample source program are "NG", this indicates a problem with the compatibility of the "unsigned_bitfield" option specification.

For SuperH-family compilers, bit field members without a specified sign are handled as signed types, whereas RX-family compilers handle them as unsigned types.

Since the " SH_sign_bit_field.c " sample source program was created assuming that bit field members without a specified sign are signed types, if the "unsigned_bitfield" option is specified, the operation results will differ from SuperH.

Sample source program: SH_sign_bit_field.c

```
Source code

struct S {
  int a : 15;
} bit = { -1 };

void sign_bit_field(void)
{
  printf("(2) sign bit field : ");
  if (bit.a < 0) {
    printf("OK\n");
  } else {
    printf("NG\n");
  }
}
```

To migrate a source program created assuming that bit field members without a specified sign are signed to RX, specify the "signed_bitfield" option.

For details about specifying this option, see 2.1.5 Sign specification for bit field members.

Also, change the options specified for the created RX project.

(3) Bit field allocation order

If the execution results of the "SH_bit_order.c" sample source program are "NG", this indicates a problem with the compatibility of the "bit_order=right" option specification.

For SuperH-family compilers, bit field members are allocated from the highest bit, whereas for RX-family compilers, they are allocated from the lowest bit.

Since the "SH_bit_order.c" sample source program was created assuming that bit field members are allocated from the highest bit, if the "bit_order=right" option is specified, the operation results will differ from SuperH.

Sample source program: SH_bit_order.c

```
Source code
union {
   unsigned char c1;
   struct {
      unsigned char b0 : 1;
      unsigned char b1 : 1;
      unsigned char b2 : 1;
      unsigned char b3 : 1;
   } b;
  } un;

void bit_order(void)
{
   printf("(3) bit field order : ");

   un.c1 = 0xc0;
   if ((un.b.b0 == 1) && (un.b.b1 == 1) &&
     (un.b.b2 == 0) && (un.b.b3 == 0)) {
      printf("OK¥n");
   } else {
      printf("NG¥n");
   }
}
```

(4) Endian-ness

If the execution results of the "SH_endian.c" sample source program are "NG", this indicates a problem with the compatibility of the "endian=little" option specification.

For SuperH-family compilers, the byte order for data is big-endian, whereas for RX-family compilers, it is little-endian.

Since the "SH_endian.c" sample source program was created assuming that the data byte order is big-endian, if the "endian=little" option is specified, the operation results will differ from SuperH.

Sample source program: SH_endian.c

```
Source code


typedef union{
  short data1;
  struct {
   unsigned char upper;
   unsigned char lower;
  } data2;
} UN;

UN u = { 0x7f6f };

void endian(void)
{
  printf("(4) endian : ");

  if (u.data2.upper == 0x7f && u.data2.lower == 0x6f) {
    printf("OK¥n");
  } else {
   printf("NG¥n");
  }
}
```

(5) double type sizes

If the execution results of the "SH_double_size.c" sample source program are "NG", this indicates a problem with the compatibility of the "dbl_size=4" option specification.

For SuperH-family compilers, the size of a double type is 8 bytes, whereas for RX-family compilers, the size of a double type is 4 bytes.

Since the "SH_double_size.c" sample source program was created assuming that the size of a double type is 8 bytes, if the "dbl_size=4" option is specified, the operation results will differ from SuperH.

Sample source program: SH_double_size.c

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

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

  printf("(5) double type size : ");

  if (d1 > d2) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

To migrate a source program created assuming that the size of a double type is 8 bytes to RX, specify the "dbl_size=8" option.

For details about specifying this option, see *2.1.3 Size specification for the double type*.
Also, change the options specified for the created RX project.

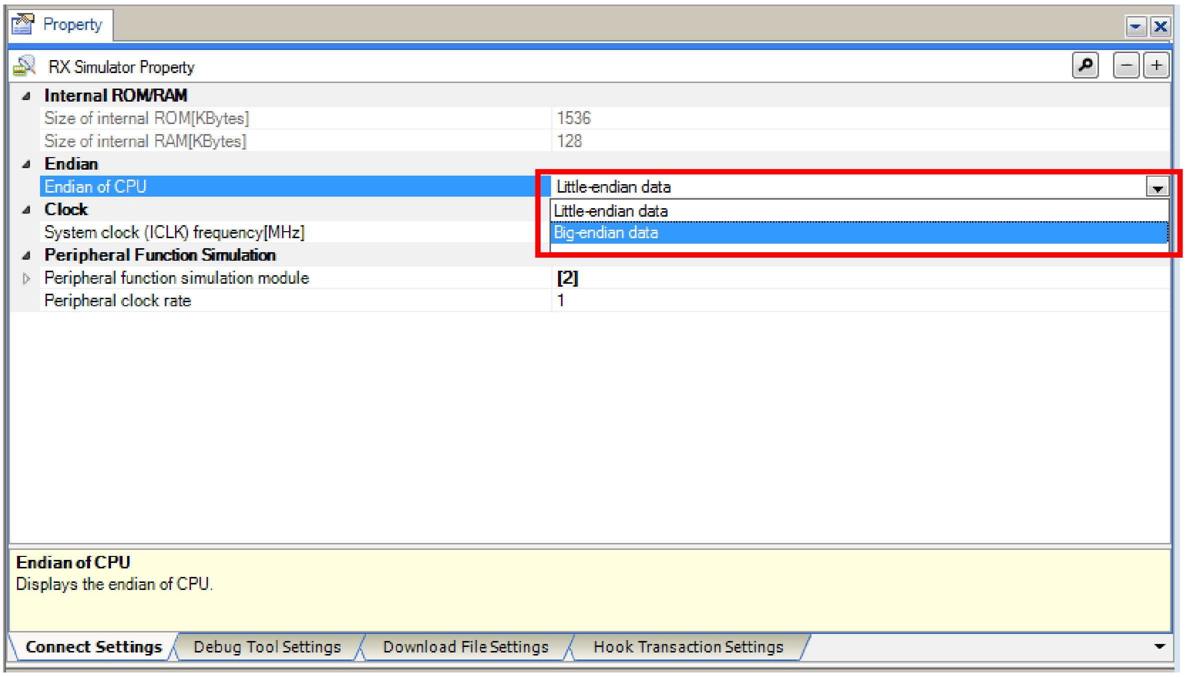
3.2.6 Performing a rebuild

(1) Setting the simulator endian

Since the endian option changed the endian from little to big, the endian of the simulator also must be changed to big.

While connecting with debug tool, simulator endian cannot be changed. First choose [debug], and then [Disconnect from Debug Tool], in CS+.

Set [Endian] category as follows in [Connect Settings] tab of [Property] page.



Property

RX Simulator Property

Internal ROM/RAM	
Size of internal ROM[KBytes]	1536
Size of internal RAM[KBytes]	128
Endian	
Endian of CPU	Little-endian data Little-endian data Big-endian data
Clock	
System clock (ICLK) frequency[MHz]	
Peripheral Function Simulation	
Peripheral function simulation module	[2]
Peripheral clock rate	1

Endian of CPU
Displays the endian of CPU.

Figure 3-13

3.2.7 Checking execution results

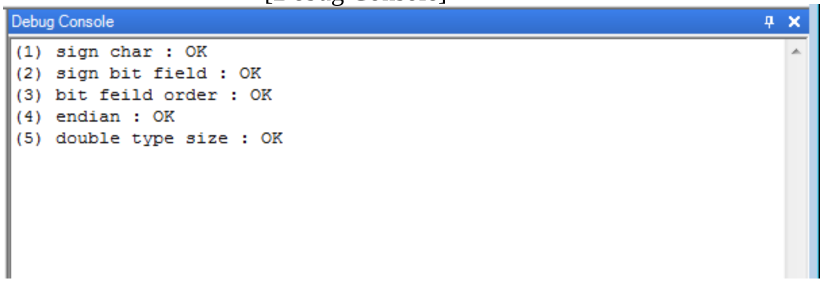
Execute the rebuilt load module in the simulator, and check that the execution results are valid.

For details about how to run the simulator, see *3.2.4 (2) Executing the simulator*.

The module is executed in the simulator, and the source program standard output is displayed in the I/O Simulation window.

Make sure that the displayed result is "OK". If it is "NG", check the specified option again.

[Debug Console]



```
(1) sign char : OK
(2) sign bit field : OK
(3) bit feild order : OK
(4) endian : OK
(5) double type size : OK
```

Figure 3-14

4. Correlation Lists

4.1 Options

Hardware-dependent options for SuperH-family C/C++ compilers are not compatible with RX-family C/C++ compilers.

The following table lists the correlated options. Uppercase letters indicate characters for abbreviated format specification. RX does not have an abbreviated format.

Options for which the format differs from RX will need their specifications changed, and options that are not compatible need to be deleted.

Table 4-1 List of correlated options

SuperH	RX	Note
Include = <path-name>[,…]	include = < path-name >[,⋯]	
PREInclude = < file-name >[,…]	preinclude = < file-name >[,⋯]	
DEFine = <sub>[,…]	define = <sub>[,⋯]	
MEssage | NOMEssage	message | nomessage	
FILE_INLINE_PATH= < path-name >[,...]	file_inline_path=< path-name >[,⋯]	
CHAnge_message =<sub>[,…]	change_message=<sub>[,⋯]	
PREProcessor[= < file-name >]	output = prep	
Code ={ Machinecode | Asmcode }	output= { obj | src }	
DEBug	debug	
SEction = <sub>[,…]	section = <sub>[,⋯]	
STring = { Const | Data }	—	
OBjectfile = < file-name >	output = obj = < file-name >	
Template = { None |Static |Used |ALl |AUto }	—	
ABs16 =<sub>[,…]	—	Same as SuperH ABS20, ABS28, and ABS32
DIvision = Cpu ={ Inline | Runtime }	—	
IFUnc	—	
ALIGN16	—	Same as SuperH ALIGN32
TBR [=<section-name >]	—	
BSs_order = { DEClaration | DEFinition }	—	
STUff [={Bss | Data | Const} [,…]]	nostuff[= { B | D | C } [,⋯]]	nostuff is specified on RX for items for which stuff is specified for SuperH
Listfile [= < file-name >]	listfile[=<file-name >]	
SHow = <sub>[,…]	show = <sub>[,⋯]	The way in which the <sub> option is specified is different
OPtimize = 0	optimize = 1	Not optimized
OPtimize = 1	optimize = 2	Optimized
OPtimize = Debug_only	optimize = 0	Code that yields to the level of debugging information
SPeed	speed	
SIze	size	
NOSPeed	—	
Goptimize	goptimize	
MAP = < file-name >	map=< file-name >	
SMap	smap	
GBr = { Auto | User }	—	
CAse = { Ifthen | Table }	case = { ifthen | table | auto }	
SHIft = { Inline | Runtime }	—	
BLOckcopy = { Inline | Runtime }	—	
Unaligned = { Inline | Runtime }	—	
INLine[= < number >]	inline[= < integer >]	
FILe_inline= < file-name >[,…]	file_inline = < file-name >[,⋯]	
GLOBAL_Volatile={ 0 | 1 }	novolatile | volatile	
OPT_Range={All | NOLoop | NOBlock }	—	

DEL_vacant_loop={ 0 | 1 }	—	
MAX_unroll=< number >	—	
INFinite_loop={ 0 | 1 }	—	
GLOBAL_Alloc={ 0 | 1 }	—	
STRUCT_Alloc={ 0 | 1 }	—	
CONST_Var_propagate={ 0 | 1 }	const_copy	
CONST_Load={ Inline | Literal }	—	
SChedule={ 0 | 1 }	schedule | noschedule	
SOftpipe	—	
SCOpe	scope	
NOSCOpe	noscope	
LOGIc_gbr	—	
ECpp	lang = ecpp	
DSpc	—	
COMment = { Nest | NONest }	comment = { nest | nonest }	
Macsave = { 0 | 1 }	—	
SAve_cont_reg={ 0 | 1 }	—	
RTnext	—	
LOop	loop[=<number>]	
APproxdiv	approxdiv	
PAtch=7055	—	
FPScr = { Safe | Aggressive }	—	
Volatile_loop	—	
AUto_enum	auto_enum	
ENAble_register	enable_register	
STRIct_ansi	—	
FDIv	—	
FIXED_Const	—	
FIXED_Max	—	
FIXED_Noround	—	
REPeat	—	
SIMple_float_conv	simple_float_conv	
CPu=< CPU-type >	cpu=< CPU-type >	The <CPU-type> is different.
ENdian = { Big | Little }	endian = { big | little }	
FPu = { Single | Double }	—	
Round = { Zero | Nearest }	round = { zero | nearest }	
DENormalize = { OFF | ON }	denormalize = { off | on }	
Pic = { 0 | 1 }	—	
DOuble = Float	dbl_size = 4	
BIt_order={ Left | Right }	bit_order = { left | right }	
PACK={ 1 | 4 }	pack | unpack	
EXception	exception	
RTTI = { ON | OFF }	rtti= { on | off}	
DIvision = { Cpu | Peripheral | Nomask }	—	
LAng = { C | CPp }	lang = { c | cpp | ecpp | c99 }	
LOGO | NOLOGO	logo | nologo	
Euc | SJis | LATin1	euc | sjis | latin1 | utf8	
OUtcode = { EUc | SJis }	outcode = { euc | sjis | utf8 }	
SUbcommand = < file-name >	subcommand = < file-name >	
STUFF_GBR	—	

ALIGN4={ALL|LOOP|INMOSTLOOP}	—	
CPP_NOINLINE	—	
CONST_VOLATILE={DATA|CONST}	—	

4.2 #pragma

The following SuperH-family C/C++ compiler pragma are not compatible with RX-family C/C++ compilers.

#pragma abs16
#pragma abs20
#pragma abs28
#pragma abs32
#pragma regsave
#pragma noregsave
#pragma noregalloc
#pragma ifunc
#pragma tbr
#pragma global_register
#pragma gbr_base
#pragma gbr_base1
#pragma align4

Since these pragma are used for RX, the following warning message is output during compilation:

```
W0520161:Unrecognized #pragma
```

Also, the format is different for #pragma interrupt, for declaring an interrrupt function. Change these as necessary to comply with the RX-family C/C++ compiler specification.

[SuperH]
#pragma interrupt [(]<function-name>[(interrupt-specification)][,…][)]

Table 4-2 List of SuperH interrupt specifications

Item	Format
Stack switching specification	sp=<address>
Trap instruction return specification	tn=<trap-vector-number>
Register bank specification	resbank
Register bank switching specification	sr_rts
RTS instruction return specification	rts

[RX]
#pragma interrupt [(]<function-name>[(<interrupt-specification>[,⋯])][,⋯][)]

Table 4-3 List of RX interrupt specifications

Item	Format
Vector table specification	vect= <vector-number>
High-speed interrupt specification	fint
Interrupt function register control specification	save
Multiplex interruptible specification	enable

4.3 Embedded functions

Almost all embedded functions for SuperH-family C/C++ compilers are incompatible with RX-family C/C++ compilers. Either delete these embedded functions as needed, or replace them with embedded functions with similar functionality for RX-family C/C++ compilers. Note that DSP embedded functions cannot be used with RX.

The following table lists the embedded functions for SuperH, and their correlated RX functions.

Table 4-4 List of correlated embedded functions

SuperH	RX	Function
nop	nop	NOP command
swapb, swapw, end_cnvl	revl, revw	Sort
macw, macwl, macl, macll	rmpab, rmpaw, rmpal	Arithmetic operations
rotl, rotr, rotcl, rotcr	rotl, rotr, rolc, rorc	Rotate

Using embedded functions, make sure to include <machine.h>. <umachine.h> and <smachine.h> cannot be used with RX.

Website and Support <website and support,ws>

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

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

All trademarks and registered trademarks are the property of their respective owners.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Apr.20.10	—	First edition issued
2.00	Apr.20.17	—	Revised the destination to CS+ and CC-RX V2

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.

Notice

1. 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 or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other disputes involving 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, including but not limited to, the product data, drawing, chart, program, algorithm, application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics products.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

   "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 etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (space and undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics 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 ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of failure or malfunction of Renesas Electronics products, 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 by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall 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. You shall not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, 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.

(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.

(Rev.3.0-1 November 2016)

RENESAS

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, HAL II 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

© 2017 Renesas Electronics Corporation. All rights reserved.

Colophon 6.0