Introduction

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

Contents

1. Introduction .................................................................................................................... 2
   1.1 Dependencies on processing in C ............................................................................................ 2
2. Functionality Requiring Care during Migration ............................................................ 2
   2.1 Options......................................................................................................................................... 2
      2.1.1 Sign specification for the char type.................................................................................... 3
      2.1.2 Size specification for enum ................................................................................................. 4
      2.1.3 Specifying the size of double type........................................................................................... 5
      2.1.4 Endian specification ............................................................................................................. 6
      2.1.5 Sign specification for bit field members ............................................................................ 7
      2.1.6 Allocation order specification for bit field members......................................................... 8
      2.1.7 Allocation order specification for bit field members......................................................... 9
   2.2 Language specification ............................................................................................................ 10
      2.2.1 Signs for char types ........................................................................................................... 10
      2.2.2 Sizes for double types ........................................................................................................ 11
      2.2.3 Endianness .......................................................................................................................... 12
      2.2.4 Allocation order for bit fields ............................................................................................. 13
      2.2.5 Signs for bit fields ............................................................................................................... 14
      2.2.6 Extended language specification ...................................................................................... 15
      2.2.7 Predefined macros .............................................................................................................. 15
3. Migration Sample Project ............................................................................................ 16
   3.1 List of main processing files ........................................................................................... 16
   3.2 Migrating the SuperH sample project to RX........................................................................... 17
      3.2.1 Creating an RX project ....................................................................................................... 17
      3.2.2 Migrating main processing source files ........................................................................... 19
      3.2.3 Performing a build ............................................................................................................. 21
      3.2.4 Executing the simulator ..................................................................................................... 22
      3.2.5 Setting options .................................................................................................................... 25
      3.2.6 Performing a rebuild ........................................................................................................... 30
      3.2.7 Checking execution results ............................................................................................... 30
4. Correlation Lists ........................................................................................................... 31
   4.1 Options..................................................................................................................................... 31
   4.2 #pragma ................................................................................................................................... 35
   4.3 Embedded functions.............................................................................................................. 36
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 on 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>
<thead>
<tr>
<th>No</th>
<th>Functionality</th>
<th>H8 option</th>
<th>RX option</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sign specification for the char type</td>
<td>--</td>
<td>signed_char</td>
<td>2.1.1</td>
</tr>
<tr>
<td>2</td>
<td>Size specification for enum</td>
<td>auto_enum</td>
<td>auto_enum</td>
<td>2.1.2</td>
</tr>
<tr>
<td>3</td>
<td>Size specification for the double type</td>
<td>double=float</td>
<td>dbl_size</td>
<td>2.1.3</td>
</tr>
<tr>
<td>4</td>
<td>Endian specification</td>
<td>endian</td>
<td>endian</td>
<td>2.1.4</td>
</tr>
<tr>
<td>5</td>
<td>Sign specification for bit field members</td>
<td>--</td>
<td>signed_bitfield</td>
<td>2.1.5</td>
</tr>
<tr>
<td>6</td>
<td>Allocation order specification for bit field members</td>
<td>bit_order</td>
<td>bit_order</td>
<td>2.1.6</td>
</tr>
<tr>
<td>7</td>
<td>Allocation specification for structures</td>
<td>pack</td>
<td>pack</td>
<td>2.1.7</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>signed_char</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned_char</td>
</tr>
</tbody>
</table>

[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]
2.1.2 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.

![Figure 2-2](image-url)
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.

![Figure 2-3](image)

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.

![Figure 2-4](image-url)
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
unsigned_bitfield : 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.

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

\[
\text{bit\_order} = \{ \text{left} | \text{right} \} : \text{right by default}
\]

[How to specify this option in CS+]

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

![Figure 2-6](image-url)
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.

![Figure 2-7](image-url)
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

<table>
<thead>
<tr>
<th>No.</th>
<th>Functionality</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Signs for char types</td>
<td>2.2.1</td>
</tr>
<tr>
<td>2</td>
<td>Sizes for double types</td>
<td>2.2.2</td>
</tr>
<tr>
<td>3</td>
<td>Endianness</td>
<td>2.2.3</td>
</tr>
<tr>
<td>4</td>
<td>Allocation order for bit fields</td>
<td>2.2.4</td>
</tr>
<tr>
<td>5</td>
<td>Signs for bit fields</td>
<td>2.2.5</td>
</tr>
</tbody>
</table>

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.

```c
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

```c
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

```c
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

```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) {
    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

```c
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

<table>
<thead>
<tr>
<th>SuperH</th>
<th>RX</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma pack 1</td>
<td>#pragma pack</td>
<td>1 is used for the alignment count</td>
</tr>
<tr>
<td>#pragma pack 4</td>
<td>#pragma un pack</td>
<td>The default alignment is used</td>
</tr>
<tr>
<td>#pragma un pack</td>
<td>#pragma pack option</td>
<td>The pack option is used</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Option</th>
<th>Predefined macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>endian=big</td>
<td>__BIG</td>
</tr>
<tr>
<td>endian=little</td>
<td>__LIT</td>
</tr>
<tr>
<td>double=float</td>
<td>__FLT</td>
</tr>
<tr>
<td></td>
<td>__FLT</td>
</tr>
<tr>
<td>denormalize=on</td>
<td>__DON</td>
</tr>
<tr>
<td>round=nearest</td>
<td>__RON</td>
</tr>
</tbody>
</table>

### Table 2-5 Predefined macros for RX

<table>
<thead>
<tr>
<th>Option</th>
<th>Predefined macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>endian=big</td>
<td>__BIG</td>
</tr>
<tr>
<td>endian=little</td>
<td>__LIT</td>
</tr>
<tr>
<td>double=float</td>
<td>__DBL4</td>
</tr>
<tr>
<td>denormalize=on</td>
<td>__DON</td>
</tr>
<tr>
<td>round=nearest</td>
<td>__RON</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>No</th>
<th>Functionality</th>
<th>File name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Signs for char types</td>
<td>SH_sign_char.c</td>
<td>3.2.5(1)</td>
</tr>
<tr>
<td>2</td>
<td>Sign for bit field members</td>
<td>SH_sign_bit_field.c</td>
<td>3.2.5(2)</td>
</tr>
<tr>
<td>3</td>
<td>Allocation for bit field members</td>
<td>SH_bit_order.c</td>
<td>3.2.5(3)</td>
</tr>
<tr>
<td>4</td>
<td>Endianness</td>
<td>SH_endian.c</td>
<td>3.2.5(4)</td>
</tr>
<tr>
<td>5</td>
<td>Size for the double type</td>
<td>SH_double_size.c</td>
<td>3.2.5(5)</td>
</tr>
<tr>
<td>6</td>
<td>main function</td>
<td>SH_Sample.c</td>
<td>—</td>
</tr>
</tbody>
</table>
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].

![Figure 3-1](image1)

(2) Select where to copy sample project.

Select folder to copy sample project.

![Figure 3-2](image2)
(3) Select debug tool

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

![Figure 3-3](image-url)

(4) Select Stream I/O mode

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

![Figure 3-4](image-url)
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]

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+.

![Add Existing File dialog box](image1)

Select the files to register.

After selecting, click Add.

![DebugConsole_Sample.c](image2)

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].

![Remove from Project](image3)

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+.

![Figure 3-8](image-url)
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+.

(2) Download to Debug Tool

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

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]

Figure 3-12

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

```c
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.
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

```c
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

```c
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

```c
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

```c
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+.

![Figure 3-13](image)

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.

![Figure 3-14](image)
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

<table>
<thead>
<tr>
<th>SuperH</th>
<th>RX</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include = &lt;path-name&gt;[,...]</td>
<td>include = &lt;path-name&gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>PREInclude = &lt;file-name&gt;[,...]</td>
<td>preinclude = &lt;file-name&gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>DEFine = &lt;sub&gt;[,...]</td>
<td>define = &lt;sub&gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>M(es)Essage</td>
<td>NOMEssage</td>
<td>message</td>
</tr>
<tr>
<td>FILE_INLINE_PATH= &lt;path-name&gt;</td>
<td>file_inline_path=&lt;path-name&gt;</td>
<td></td>
</tr>
<tr>
<td>CHAnge_message =&lt;sub&gt;[,...]</td>
<td>change_message=&lt;sub&gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>PREProcessor= &lt;file-name&gt;</td>
<td>output = prep</td>
<td></td>
</tr>
<tr>
<td>Code ={ Machinecode</td>
<td>Asmcode }</td>
<td>output= { obj</td>
</tr>
<tr>
<td>DEBug</td>
<td>debug</td>
<td></td>
</tr>
<tr>
<td>SEcction = &lt;sub&gt;[,...]</td>
<td>section = &lt;sub&gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>STring = { Const</td>
<td>Data }</td>
<td></td>
</tr>
<tr>
<td>OBJECTfile = &lt;file-name&gt;</td>
<td>output = obj = &lt;file-name&gt;</td>
<td></td>
</tr>
<tr>
<td>Template = { None</td>
<td>Static</td>
<td>Used</td>
</tr>
<tr>
<td>ABS16 =&lt;sub&gt;[,...]</td>
<td></td>
<td>Same as SuperH ABS20, ABS28, and ABS32</td>
</tr>
<tr>
<td>D ivision = Cpu ={ Inline</td>
<td>Runtime }</td>
<td></td>
</tr>
<tr>
<td>IFUnc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN16</td>
<td></td>
<td>Same as SuperH ALIGN32</td>
</tr>
<tr>
<td>TBR [=&lt;section-name&gt;]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSs_order = { DEClaration</td>
<td>DEFinition }</td>
<td></td>
</tr>
<tr>
<td>STUff ={Bss</td>
<td>Data</td>
<td>Const} [,...]</td>
</tr>
<tr>
<td>Listfile [ = &lt; file-name &gt;]</td>
<td>listfile[=&lt;file-name &gt;]</td>
<td></td>
</tr>
<tr>
<td>SHow = &lt;sub&gt;[,...]</td>
<td>show = &lt;sub&gt;[,...]</td>
<td>The way in which the &lt;sub&gt; option is specified is different</td>
</tr>
<tr>
<td>OPtimize = 0</td>
<td>optimize = 1</td>
<td>Not optimized</td>
</tr>
<tr>
<td>OPtimize = 1</td>
<td>optimize = 2</td>
<td>Optimized</td>
</tr>
<tr>
<td>OPtimize = Debug_only</td>
<td>optimize = 0</td>
<td>Code that yields to the level of debugging information</td>
</tr>
<tr>
<td>SSpeed</td>
<td>speed</td>
<td></td>
</tr>
<tr>
<td>SIze</td>
<td>size</td>
<td></td>
</tr>
<tr>
<td>NOSpeed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gopimize</td>
<td>goptimize</td>
<td></td>
</tr>
<tr>
<td>MAP = &lt; file-name &gt;</td>
<td>map=&lt; file-name &gt;</td>
<td></td>
</tr>
<tr>
<td>SMap</td>
<td>smap</td>
<td></td>
</tr>
<tr>
<td>GBr = { Auto</td>
<td>User }</td>
<td></td>
</tr>
<tr>
<td>CAse = { Ifthen</td>
<td>Table }</td>
<td>case = { ifthen</td>
</tr>
<tr>
<td>SHIf = { Inline</td>
<td>Runtime }</td>
<td></td>
</tr>
<tr>
<td>BLockcopy = { Inline</td>
<td>Runtime }</td>
<td></td>
</tr>
<tr>
<td>Unaligned = { Inline</td>
<td>Runtime }</td>
<td></td>
</tr>
<tr>
<td>INLine[ = &lt; number &gt;]</td>
<td>inline[ = &lt; integer &gt;]</td>
<td></td>
</tr>
<tr>
<td>FILE_inline= &lt; file-name &gt;[,...]</td>
<td>file_inline = &lt; file-name &gt;[,...]</td>
<td></td>
</tr>
<tr>
<td>GLOBAL_Volatile={ 0</td>
<td>1 }</td>
<td>novolatile</td>
</tr>
<tr>
<td>OPT_Range={All</td>
<td>NOLoop</td>
<td>NOBlock }</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>DEL_vacant_loop</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>MAX_unroll</td>
<td><code>&lt; number &gt;</code></td>
<td>—</td>
</tr>
<tr>
<td>INFInite_loop</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>GLOBAL_Alloc</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>STRUCT_Alloc</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>CONST_Var_propagate</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>CONST_Load</td>
<td>`{ Inline</td>
<td>Literal }`</td>
</tr>
<tr>
<td>SChedule</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>SOftpipe</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SCOpe</td>
<td><code>scope</code></td>
<td>—</td>
</tr>
<tr>
<td>NOSCOpe</td>
<td><code>noscope</code></td>
<td>—</td>
</tr>
<tr>
<td>LOGIc_gbr</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ECpp</td>
<td><code>lang = ecpp</code></td>
<td>—</td>
</tr>
<tr>
<td>DSpc</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>COMment</td>
<td>`{ Nest</td>
<td>NONest }`</td>
</tr>
<tr>
<td>Macsave</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>SAvve_cont_reg</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>RTnext</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>LOp</td>
<td><code>loop[=number]</code></td>
<td>—</td>
</tr>
<tr>
<td>APProxdiv</td>
<td><code>approxdiv</code></td>
<td>—</td>
</tr>
<tr>
<td>PATch</td>
<td><code>7055</code></td>
<td>—</td>
</tr>
<tr>
<td>FPScr</td>
<td>`{ Safe</td>
<td>Aggressive }`</td>
</tr>
<tr>
<td>Volatile_loop</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AUto_enum</td>
<td><code>auto_enum</code></td>
<td>—</td>
</tr>
<tr>
<td>ENAble_register</td>
<td><code>enable_register</code></td>
<td>—</td>
</tr>
<tr>
<td>STRIct_ansi</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FDiv</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FIXED_Const</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FIXED_Max</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FIXED_Noround</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>REPeat</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SIMple_float_conv</td>
<td><code>simple_float_conv</code></td>
<td>—</td>
</tr>
<tr>
<td>CPu</td>
<td><code>&lt; CPU-type &gt;</code></td>
<td><code>cpu='&lt; CPU-type &gt;</code></td>
</tr>
<tr>
<td>ENdian</td>
<td>`{ Big</td>
<td>Little }`</td>
</tr>
<tr>
<td>FPu</td>
<td>`{ Single</td>
<td>Double }`</td>
</tr>
<tr>
<td>Round</td>
<td>`{ Zero</td>
<td>Nearest }`</td>
</tr>
<tr>
<td>DENormalize</td>
<td>`{ OFF</td>
<td>ON }`</td>
</tr>
<tr>
<td>Pic</td>
<td>`{ 0</td>
<td>1 }`</td>
</tr>
<tr>
<td>DOuble</td>
<td><code>Float</code></td>
<td><code>dbl_size = 4</code></td>
</tr>
<tr>
<td>BIT_order</td>
<td>`{ Left</td>
<td>Right }`</td>
</tr>
<tr>
<td>PACK</td>
<td>`{ 1</td>
<td>4 }`</td>
</tr>
<tr>
<td>EXception</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RTTI</td>
<td>`{ ON</td>
<td>OFF }`</td>
</tr>
<tr>
<td>DIvision</td>
<td>`{ Cpu</td>
<td>Peripheral</td>
</tr>
<tr>
<td>LANG</td>
<td>`{ C</td>
<td>CPp }`</td>
</tr>
<tr>
<td>LOGO</td>
<td><code>{ NOLOGO }</code></td>
<td>`logo</td>
</tr>
<tr>
<td>Euc</td>
<td>SJis</td>
<td>LATin1</td>
</tr>
<tr>
<td>OUtcode</td>
<td>`{ Euc</td>
<td>SJis }`</td>
</tr>
<tr>
<td>Subcommand</td>
<td><code>&lt; file-name &gt;</code></td>
<td><code>subcommand = &lt; file-name &gt;</code></td>
</tr>
<tr>
<td>STUFF_GBR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ALIGN4=(ALL</td>
<td>LOOP</td>
<td>INMOSTLOOP)</td>
</tr>
<tr>
<td>CPP_NOINLINE</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>CONST_VOLATILE=(DATA</td>
<td>CONST)</td>
<td>—</td>
</tr>
</tbody>
</table>
4.2 #pragma

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

```c
#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 interrupt function. Change these as necessary to comply with the RX-family C/C++ compiler specification.

[SuperH]

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack switching specification</td>
<td>sp=&lt;address&gt;</td>
</tr>
<tr>
<td>Trap instruction return specification</td>
<td>tn=&lt;trap-vector-number&gt;</td>
</tr>
<tr>
<td>Register bank specification</td>
<td>resbank</td>
</tr>
<tr>
<td>Register bank switching specification</td>
<td>sr_rts</td>
</tr>
<tr>
<td>RTS instruction return specification</td>
<td>rts</td>
</tr>
</tbody>
</table>

[RX]

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector table specification</td>
<td>vect= &lt;vector-number&gt;</td>
</tr>
<tr>
<td>High-speed interrupt specification</td>
<td>fint</td>
</tr>
<tr>
<td>Interrupt function register control specification</td>
<td>save</td>
</tr>
<tr>
<td>Multiplex interruptible specification</td>
<td>enable</td>
</tr>
</tbody>
</table>

Table 4-2 List of SuperH interrupt specifications

Table 4-3 List of RX interrupt specifications
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>
<thead>
<tr>
<th>SuperH</th>
<th>RX</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>nop</td>
<td>NOP command</td>
</tr>
<tr>
<td>swapb, swapw, end_cnvl</td>
<td>revl, revw</td>
<td>Sort</td>
</tr>
<tr>
<td>macw, macwl, macl, macll</td>
<td>rmpab, rmpaw, rmpal</td>
<td>Arithmetic operations</td>
</tr>
<tr>
<td>rotl, rotr, rotl, rotlr</td>
<td>rotl, rotr, rolc, roorc</td>
<td>Rotate</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Apr.20.10</td>
<td>—</td>
<td>First edition issued</td>
</tr>
<tr>
<td>2.00</td>
<td>Apr.20.17</td>
<td>—</td>
<td>Revised the destination to CS+ and CC-RX V2</td>
</tr>
</tbody>
</table>
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, chief, 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 underwater 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 micromachine 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 any purpose relating to the development, design, manufacture, use, stocking, 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.

(Rev.3.0-1  November 2016)