SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Compiler use guide] C Coding Guide (Using FPU)

This document explains usage and gives precautions for FPUs (SH-2E, SH2A-FPU, SH-4, SH-4A), for the SuperH RISC engine C/C++ Compiler V.9.

Table of contents

Notice: There are some errata on page 15.

1. Floating-point Processing Unit (FPU) 2
1.1 Specification for Floating-point Numbers 2
1.1.1 Internal Representation of Floating-point Numbers 2
1.1.2 Single-precision (float type) data format 3
1.1.3 Double-precision (double type) data format 5
1.2 Register 6
1.2.1 Floating-point status / control registers (FPSCR) 6
1.2.2 Floating-point registers 7
1.2.3 Floating-point communication register (FPUL) 11
1.2.4 Status register (SR) 11
1.3 Rounding 11

2. Options for Floating-point Calculations and #pragma 13
2.1 Conversion from double -> float (DOuble=Float) for SH-2E 13
2.2 Floating-point Calculation Mode (FPu={Single|Double}) for SH2A-FPU, SH-4, and SH-4A 14
2.3 Rounding Methods (Round={Zero|Nearest}) for SH2A-FPU, SH-4, and SH-4A 15
2.4 Handling Non-normalized Numbers (DENormalize={OFF|ON}) for SH-4 and SH-4A 17
2.5 Converting Floating-point Division into Multiplication (APproxdiv) 18
2.6 Converting Floating-point Division (FDIv) 19
2.7 Switching the FPSCR Register Precision Mode (FPScr={Aggressive|Safe}) for SH2A-FPU, SH-4, and SH-4A 20
2.8 Skipping Range Checking during Floating-point Number-integer Conversion (SIMple_float_conv) 22
2.9 Suppressing Save/Restore for Floating-point Registers (IFUnc, #pragma ifunc) 23

3. Efficient Programming Techniques 27
3.1 Using Floating-point Instructions 27

4. Frequently Asked Questions 28
4.1 Floating-point Calculation Results 28
4.2 Values for Floating-point Numbers in the Watch Window 29
Website and Support <website and support,ws> 31

1. Floating-point Processing Unit (FPU)

SH-2E, SH2A-FPU, SH-4, and SH-4A come with a built-in FPU for performing high-speed floating-point calculations. SH2A-FPU, SH-4, and SH-4A can perform calculations using double-precision (`double` type) or single-precision (`float` type) in the FPU, while SH-2E can use only single-precision calculations

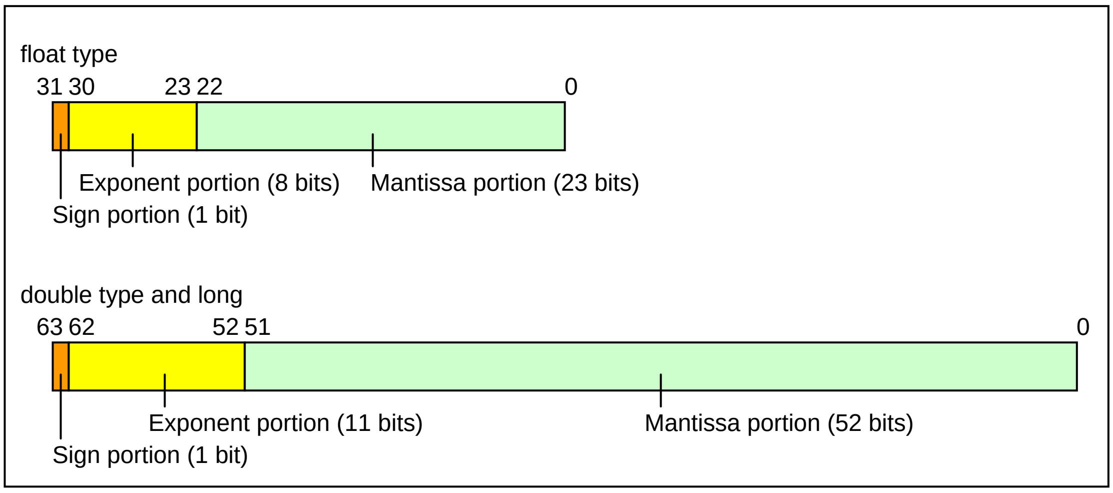
The FPU offers the following characteristics:

- Two rounding modes: Round to Nearest and Round to Zero
  (SH-2E only has Round to Zero)
- Two processing modes for non-normalized numbers (only on SH-4 and SH-4A)
- Six exception causes, allowing occurring exceptions to be masked or enabled for each.
  - FPU error, invalid calculation, division by zero, overflow, underflow, and inaccurate
    (SH-2E only has invalid calculation and division by zero)

1.1 Specification for Floating-point Numbers

1.1.1 Internal Representation of Floating-point Numbers

- Internal representation format
  `float` types are represented in the IEEE single-precision format (32-bit), and `double` types and `long double` types are represented in the IEEE double-precision format (64-bit).
- Internal representation configuration
  Figure 1-1 shows the configuration for the internal representation of the `float` type, `double` type, and `long double` type.


float type
31 30
23 22
0
Exponent portion (8 bits)
Mantissa portion (23 bits)
Sign portion (1 bit)
double type and long
63 62
52 51
0
Exponent portion (11 bits)
Mantissa portion (52 bits)
Sign portion (1 bit)

Note: When `double=float` is specified, the `double` type and `float` type have the same internal representation.
When both `cpu=sh2afpu|sh4|sh4a` and `fpu=single` are specified, the `double` type and `long double` type have the same internal representation as the `float` type.
When both `cpu=sh2afpu|sh4|sh4a` and `fpu=double` are specified, the `float` type has the same internal representation as the `double` type.

Figure 1-1

The following explains the meaning of each configuration element in an internal representation.

(i) Sign portion
Indicates the sign of the floating-point number. `0` indicates positive sign, and `1` indicates negative sign.

(ii) Exponent portion
Indicates the exponent of the floating-point number as a power of two.

(iii) Mantissa portion
Data about the significant digits in the floating-point number.

- Types of values expressed

  In addition to regular real numbers, floating-point numbers can also represent infinity and other values. The following lists the types of values that can be represented by a floating-point number.

  (i) Normalized number

  When the exponent portion is `0` or not all bits are `1`. This represents a regular real number.

  (ii) Non-normalized number

  When the exponent portion is `0`, and the mantissa portion is not `0`. This represents a real number with a small absolute value.

  (iii) Zero

  When the exponent portion and mantissa portion are `0`. This represents the value 0.0.

  (iv) Infinity

  When all bits in the exponent portion are `1` and the mantissa portion is `0`. This represents infinity.

  (v) Non-number

  When all bits in the exponent portion are `1` and the mantissa portion is not `0`. This is obtained when the calculation results are non-numerical, such as for 0.0/0.0, ∞/∞, and ∞-∞.

  Table 1-1 lists the conditions used to determine the values representing floating-point numbers.

Table 1-1 Types of values that represent floating-point numbers

Mantissa portion	Exponent portion		
	`0`	Not all bits are either `0` or `1`	All bits are `1`
0	0	Normalized number	Infinity
Anything other than 0	Non-normalized number	Normalized number	Non-number

Note: A non-normalized number represents a floating-point number with a small absolute value in a range that cannot be expressed by a normalized number, but which has fewer significant digits than a normalized number. As such, when the calculation results or any temporary results are a non-normalized number, the significant digits of the results are not guaranteed.

1.1.2 Single-precision (float type) data format

The internal representation of the `float` type consists of a 1-bit sign portion, an 8-bit exponent portion, and a 23-bit mantissa portion.

- Normalized numbers

  The sign portion is `0` (positive) or `1` (negative), indicating the sign of the value.

  The exponent portion is a value from 1 to 254 ($2^8 - 2$). The actual exponent is a value 127 less than this value, with a range from –126 to 127.

  The mantissa portion is a value from 0 to $2^{23} - 1$. The actual mantissa is interpreted as a value whose 2^{23} bit is `1`, with a decimal point immediately following.

  Values expressing normalized numbers are as follows:

  $(-1)^{\textit{sign-portion}} \times 2^{\textit{exponent-portion}-127} \times (1 + \text{mantissa-portion} \times 2^{-23})$

Example:

31	30 23	22 0
1	10000000	11000000000000000000000

Sign: –
Exponent: $10000000_{(2)} - 127 = 1$
Mantissa: $1.11_{(2)} = 1.75$
Value: $-1.75 \times 2^{1} = -3.5$
Note: $_{(2)}$ indicates a decimal number.

- Non-normalized numbers
  The sign portion is 0 (positive) or 1 (negative), indicating the sign of the value.
  The exponent portion is 0, with an actual exponent of –126.
  The mantissa portion is from 1 to 2^{23}–1, with the actual mantissa interpreted as a value whose 2^{23} bit is 0, with a decimal point immediately following.
  Values expressing non-normalized numbers are as follows:
  $(-1)^{\textit{sign-portion}} \times 2^{-126} \times (\textit{mantissa-portion} \times 2^{-23})$

  Example:

31	30 23	22 0
0	00000000	11000000000000000000000

Sign: +
Exponent: –126
Mantissa: $0.11_{(2)} = 0.75$
Value: 0.75×2^{-126}
Note: $_{(2)}$ indicates a decimal number.

- Zero
  The sign portion is 0 (positive) or 1 (negative), indicating either +0.0 or –0.0.
  Both the exponent portion and mantissa portion are 0.
  Both +0.0 and –0.0 indicate a value of 0.0.

- Infinity
  The sign portion is 0 (positive) or 1 (negative), indicating either +∞ or –∞.
  The exponent portion is 255 (2^{8}–1).
  The mantissa portion is 0.

- Non-number
  The exponent portion is 255 (2^{8}–1).
  The mantissa portion is a value other than 0.

  Note: When the CPU is SH-2E, SH2A-FPU, SH-4, or SH-4A, a non-number for which the highest order bit of the mantissa portion is 0 is called *qNaN*, a non-number for which the highest order bit of the mantissa portion is 1 is called *sNaN*.
  The values of other mantissa fields, and sign portions are not defined.

1.1.3 Double-precision (double type) data format

The internal representations of the `double` type and `long double` type consist of a 1-bit sign portion, an 11-bit exponent portion, and a 52-bit mantissa portion.

- Normalized numbers

  The sign portion is `0` (positive) or `1` (negative), indicating the sign of the value.

  The exponent portion is a value from 1 to 2046 (2^{11}–2). The actual exponent is a value 1023 less than this value, with a range from –1022 to 1023.

  The mantissa portion is a value from 0 to 2^{52}–1. The actual mantissa is interpreted as a value whose 2^{52} bit is `1`, with a decimal point immediately following.

  Values expressing normalized numbers are as follows:

  $(-1)^{\textit{sign-portion}} \times 2^{\textit{exponent-portion}-1023} \times (1 + \textit{mantissa-portion} \times 2^{-52})$

  Example:

63	62 52	51 0
0	01111111111	1110000000000000000000000000000000000000000000000000

  Sign: +
  Exponent: $1111111111_{(2)}–1023 = 0$
  Mantissa: $1.111_{(2)} = 1.875$
  Value: $1.875 \times 2^{0} = 1.875$
  Note: $_{(2)}$ indicates a decimal number.

- Non-normalized number

  The sign portion is `0` (positive) or `1` (negative), indicating the sign of the value.

  The exponent portion is `0`, with an actual exponent of –1022.

  The mantissa portion is from 1 to 2^{52}–1, with the actual mantissa interpreted as a value whose 2^{52} bit is `0`, with a decimal point immediately following.

  Values expressing non-normalized numbers are as follows:

  $(-1)^{\text{sign-portion}} \times 2^{-1022} \times (\text{mantissa-portion} \times 2^{-23})$

  Example:

63	62 52	51 0
1	00000000000	1110000000000000000000000000000000000000000000000000

  Sign: –
  Exponent: –1022
  Mantissa: $0.111_{(2)} = 0.875$
  Value: 0.875×2^{-1022}
  Note: $_{(2)}$ indicates a decimal number.

- Zero

  The sign portion is `0` (positive) or `1` (negative), indicating either +0.0 or –0.0.

  Both the exponent portion and mantissa portion are `0`.

  Both +0.0 and –0.0 indicate a value of 0.0.

- Infinity

  The sign portion is `0` (positive) or `1` (negative), indicating either $+\infty$ or $-\infty$.

  The exponent portion is 2047 (2^{11}–1).

  The mantissa portion is `0`.

- Non-number
  The exponent portion is 2047 (2^{11}–1).
  The mantissa portion is any value other than `0`.
  Note: When the CPU is SH2A-FPU, SH-4, or SH-4A, a non-number for which the highest order bit of the mantissa portion is `0` is called *qNaN*, a non-number for which the highest order bit of the mantissa portion is `1` is called *sNaN*.
  The values of other mantissa fields, and sign portions are not defined.

1.2 Register

1.2.1 Floating-point status / control registers (FPSCR)

FPSCR is a 32-bit register that controls storage of detailed information about the rounding mode, asymptotic underflow (non-normalized number), and FPU exceptions.

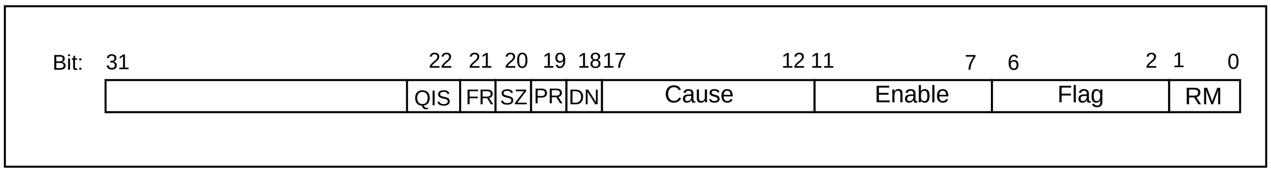


Figure 1-2

- `QIS`: (SH2A-FPU)
  qNaN or ±∞ is handled as sNaN. This only takes effect when `V=1` is set for `Enable` in FPSCR.
  `QIS=0`: Processing is performed as qNaN or ±∞.
  `QIS=1`: An exception occurred (same processing as sNaN).

- `FR`: floating-point register bank (SH-4 and SH-4A)
  Allocation is switched between floating-point registers `FPR0_BANK0` to `FPR15_BANK0` and `FPR0_BANK1` to `FPR15_BANK1`.

- `SZ`: transfer size mode (SH2A-FPU, SH-4, and SH-4A)
  `SZ=0`: The data size for FMOV instructions is 32 bits.
  `SZ=1`: The data size for FMOV instructions is 32 bit pairs (64 bits).

- `PR`: precision mode (SH2A-FPU, SH-4, and SH-4A)
  `PR=0`: Floating-point instructions are executed as single-precision calculations.
  `PR=1`: Floating-point instructions are executed as double-precision calculations. The results of instructions for which double-precision is not supported are not defined.

Note:
Do not set both `SZ` and `PR` to `1` at the same time for SH-4. This setting is reserved. `[SZ, PR]` = `11`: Reserved (The FPU calculation instructions are not defined).

- `DN`: non-normalized mode (SH-2E, SH2A-FPU, SH-4, and SH-4A)
  `DN=1` is always set for SH-2E and SH2A-FPU.
  `DN=0`: Non-normalized numbers are handled as non-normalized numbers.
  `DN=1`: Non-normalized numbers are handled as `0`.

- `Cause`: FPU exception cause field (SH-2E, SH2A-FPU, SH-4, and SH-4A)
- `Enable`: FPU exception enable field (SH-2E, SH2A-FPU, SH-4, and SH-4A)
- `Flag`: FPU exception flag field (SH-2E, SH2A-FPU, SH-4, and SH-4A)
  When an FPU calculation instruction is executed, the FPU exception cause field is first set to `0`.
  When the next FPU exception occurs, the corresponding bit in the FPU exception cause field and FPU exception flag field is set to `1`.
  The FPU exception flag field keeps the status of the exceptions that occur after the FPU exception flag field is last cleared.

Table 1-2 Bit allocation for FPU exception processing

		FPU error (E)	Invalid calculation (V)	0 division (Z)	Overflow (O)	Underflow (U)	Undetermined (I)
`Cause`	FPU exception cause field	Bit 17	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12
`Enable`	FPU exception enable field	None	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7
`Flag`	FPU exception flag field	None	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2

- `RM`: rounding mode (SH-2E, SH2A-FPU, SH-4, and SH-4A)
  This is always `01` (Round to Zero) for SH-2E.
  `RM=00`: Round to Nearest
  `RM=01`: Round to Zero

1.2.2 Floating-point registers

SH-2E and SH2A-FPU have 16 32-bit floating-point registers, while SH-4 and SH-4A have 32. Details are as follows.

- SH-2E
  Figure 1-3 shows the floating-point registers. There are 16 floating-point registers (`FR`*n*), from `FR0` to `FR15`, each of which is 32 bits long. Floating-point registers are used for floating-point instructions.

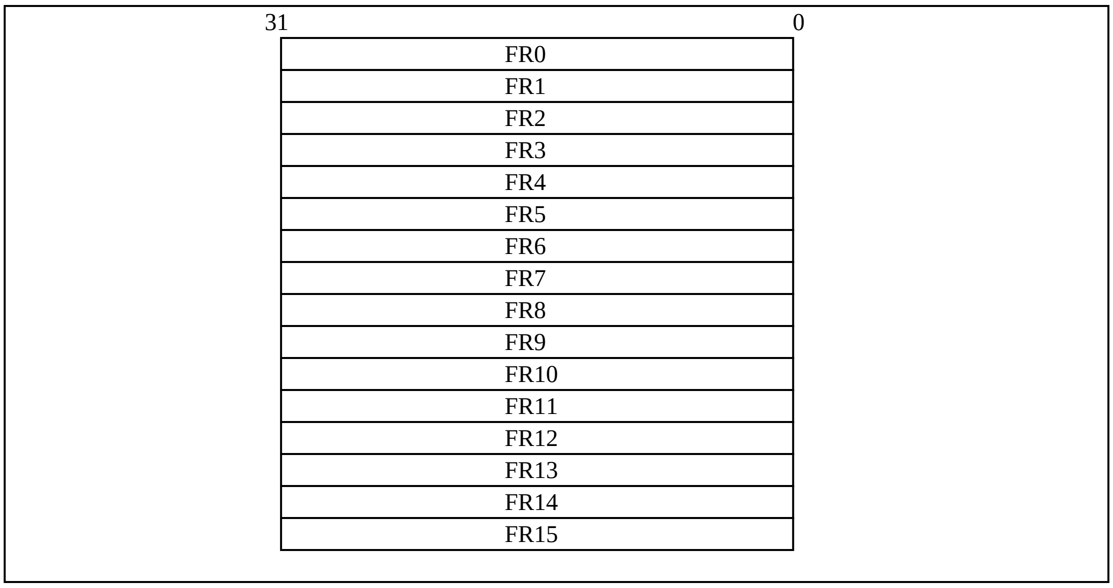


Figure 1-3

- SH2A-FPU

  Figure 1-4 shows the floating-point registers. There are 16 floating-point registers, from FR0 to FR15, each of which is 32 bits long. These 16 registers are referenced from FR0 to FR15, as DR0/2/4/6/8/l0/12/14. The correspondence between FPR*n* and reference name is determined by the FPSCR PR bit and SZ bit.

  (1) Floating-point registers FPR*n* (16 registers)
  FPR0, FPRl, FPR2, FPR3, FPR4, FPR5, FPR6, FPR7, FPR8, FPR9, FPR10, FPR11, FPR12, FPR13, FPR14, and FPR15

  (2) Single-precision floating-point registers FR*i* (16 registers)
  FR0 to FR15 are allocated to FPR0 to FPR15.

  (3) Double-precision floating-point registers, or single-precision floating-point register pairs DR*i* (8 registers)
  DR0 = {FPR0, FPR1}, DR2 = {FPR2, FPR3},
  DR4 = {FPR4, FPR5}, DR6 = {FPR6, FPR7},
  DR8 = {FPR8, FPR9}, DR10 = {FPR10, FPR11},
  DR12 = {FPR12, FPR13}, DR14 = {FPR14, FPR15}

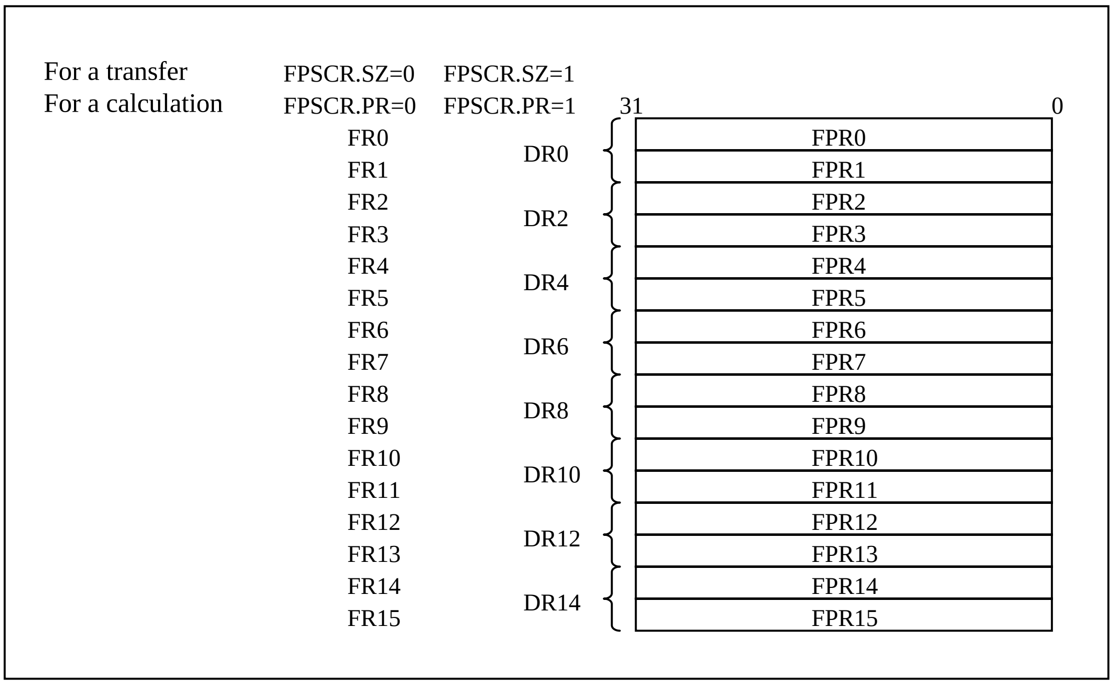


Figure 1-4

- SH-4 and SH-4A

  Figure 1-5 shows the decimal point registers, for which there are thirty-two 32-bit floating-point registers. Each is comprised of two banks, `FPR0_BANK0` to `FPR15_BANK0`, and `FPR0_BANK1` to `FPR15_BANK1`. These 32 registers are referenced as `FR0` to `FR15`, `DR0/2/4/6/8/l0/12/14`, `FV0/4/8/12`, `XF0` to `XF15`, `XD0/2/4/6/8/l0/12/14`, and `XMTRX`. The correspondence between `FPR`*n*`_BANK`*i* and the reference name is determined by the `FR` bit for FPSCR.

  (1) Floating-point registers `FPR`*n*`_BANK`*i* (32 registers)
  FPR0_BANK0, FPRl_BANK0, FPR2_BANK0, FPR3_BANK0,
  FPR4_BANK0, FPR5_BANK0, FPR6_BANK0, FPR7_BANK0,
  FPR8_BANK0, FPR9_BANK0, FPR10_BANK0, FPR11_BANK0,
  FPR12_BANK0, FPR13_BANK0, FPR14_BANK0, and FPR15_BANK0
  FPR0_BANK1, FPR1_BANK1, FPR2_BANK1, FPR3_BANK1,
  FPR4_BANK1, FPR5_BANK1, FPR6_BANK1, FPR7_BANK1,
  FPR8_BANK1, FPR9_BANK1, FPR10_BANK1, FPR11_BANK1,
  FPR12_BANK1, FPR13_BANK1, FPR14_BANK1, and FPR15_BANK1

  (2) Single-precision floating-point registers `FR`*i* (16 registers)
  When FPSCR.FR = 0 is set, FR0 to FR15 are allocated to FPR0_BANK0 to FPR15_BANK0.
  When FPSCR.FR = 1 is set, FR0 to FR15 are allocated to FPR0_BANK1 to FPR15_BANK1.

  (3) Double-precision floating-point registers, or pairs of single-precision floating-point registers `DR`*i* (8 registers)
  A `DR` register consists of two `FR` registers.
  `DR0` = {`FR0`, `FR1`}, `DR2` = {`FR2`, `FR3`},
  `DR4` = {`FR4`, `FR5`}, `DR6` = {`FR6`, `FR7`},
  `DR8` = {`FR8`, `FR9`}, `DR10` = {`FR10`, `FR11`},
  `DR12` = {`FR12`, `FR13`}, and `DR14` = {`FR14`, `FR15`}

  (4) Single-precision floating-point vector registers `FV`*i* (4 registers)
  An `FV` register consists of four `FR` registers.
  `FV0` = {`FR0`, `FR1`, `FR2`, `FR3`},
  `FV4` = {`FR4`, `FR5`, `FR6`, `FR7`},
  `FV8` = {`FR8`, `FR9`, `FR10`, `FR11`}, and
  `FVl2` = {`FR12`, `FR13`, `FR14`, `FR15`}

  (5) Single-precision floating-point extended registers `XF`*i* (16 registers)
  When `FPSCR.FR = 0` is set, `XF0` to `XF15` are allocated to `FPR0_BANK1` to `FPR15_BANK1`.
  When `FPSCR.FR = 1` is set, `XF0` to `XF15` are allocated to `FPR0_BANK0` to `FPR15_BANK0`.

  (6) Single-precision floating-point extended register pairs `XD`*i* (8 registers)
  An `XD` register consists of two `XF` registers.
  `XD0` = {`XF0`, `XF1`}, `XD2` = {`XF2`, `XF3`},
  `XD4` = {`XF4`, `XF5`}, `XD6` = {`XF6`, `XF7`},
  `XD8` = {`XF8`, `XF9`}, `XD10` = {`XF10`, `XF11`},
  `XD12` = {`XF12`, `XF13`}, and `XD14` = {`XF14`, `XF15`}

  (7) Single-precision floating-point extended register matrix `XMTRX`
  `XMTRX` consists of 16 `XF` registers.

$$\mathtt{XMTRX} = \begin{pmatrix} \mathtt{XF0} & \mathtt{XF4} & \mathtt{XF8} & \mathtt{XF12} \\ \mathtt{XF1} & \mathtt{XF5} & \mathtt{XF9} & \mathtt{XF13} \\ \mathtt{XF2} & \mathtt{XF6} & \mathtt{XF10} & \mathtt{XF14} \\ \mathtt{XF3} & \mathtt{XF7} & \mathtt{XF11} & \mathtt{XF15} \end{pmatrix}$$

FPSCR.FR=0			31 — 0	FPSCR.FR=1		
FV0	DR0	FR0	FPR0_BANK0	XR0	XR0	XMTRX
		FR1	FPR1_BANK0	XR1		
	DR2	FR3	FPR2_BANK0	XR3	XR2	
		FR3	FPR3_BANK0	XR3		
FV4	DR4	FR4	FPR4_BANK0	XR4	XR4	
		FR5	FPR5_BANK0	XR5		
	DR6	FR6	FPR6_BANK0	XR6	XR6	
		FR7	FPR7_BANK0	XR7		
FV8	DR8	FR8	FPR8_BANK0	XR8	XR8	
		FR9	FPR9_BANK0	XR9		
	DR10	FR10	FPR10_BANK0	XR10	XR10	
		FR11	FPR11_BANK0	XR11		
FV12	DR12	FR12	FPR12_BANK0	XR12	XR12	
		FR13	FPR13_BANK0	XR13		
	DR14	FR14	FPR14_BANK0	XR14	XR14	
		FR15	FPR15_BANK0	XR15		

			31 — 0			
XMTRX	XR0	XR0	FPR0_BANK1	FR0	DR0	FV0
		XR1	FPR1_BANK1	FR1		
	XR2	XR3	FPR2_BANK1	FR3	DR2	
		XR3	FPR3_BANK1	FR3		
	XR4	XR4	FPR4_BANK1	FR4	DR4	FV4
		XR5	FPR5_BANK1	FR5		
	XR6	XR6	FPR6_BANK1	FR6	DR6	
		XR7	FPR7_BANK1	FR7		
	XR8	XR8	FPR8_BANK1	FR8	DR8	FV8
		XR9	FPR9_BANK1	FR9		
	XR10	XR10	FPR10_BANK1	FR10	DR10	
		XR11	FPR11_BANK1	FR11		
	XR12	XR12	FPR12_BANK1	FR12	DR12	FV12
		XR13	FPR13_BANK1	FR13		
	XR14	XR14	FPR14_BANK1	FR14	DR14	
		XR15	FPR15_BANK1	FR15		

Figure 1-5

1.2.3 Floating-point communication register (FPUL)

The FPUL register is used to relay information between the FPU and CPU. The 32-bit FPUL register is the system register, and can be accessed from the CPU through LDS and STS instructions. For example, the processing flow for converting an integer stored in general register `R1` to a single-precision floating-point is as follows:

`R1` -> (LDS instruction) -> FPUL -> (single-precision FLOAT instruction) -> `FR1`

1.2.4 Status register (SR)

When the `FD` bit of a status register (SR) is `1`, FPU instructions throw general FPU suppression exceptions, and if the FPU instruction is in a delay slot, a slot FPU suppression exception occurs (FPU instruction: H'F*** instruction, LDS (`.L`) / STS (`.L`) instruction for FPUL/FPSCR).

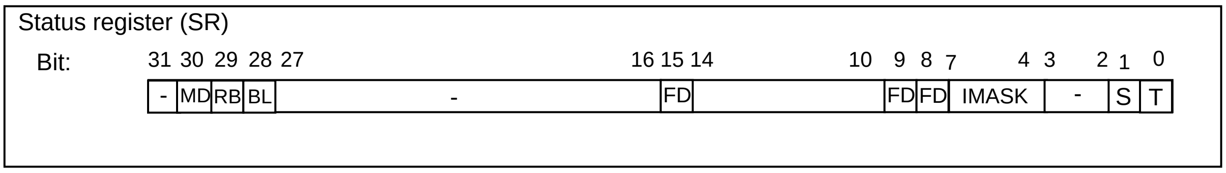


Figure 1-6

1.3 Rounding

The following rounding is performed during arithmetic calculation for floating-point numbers or constant substitution when the mantissa of the internal representation of a proper value exceeds the significant digits:

- When the CPU is SH-2E, Round to Zero is performed (the portion beyond the significant digits is truncated).
- When the CPU is SH2A-FPU, SH-4, or SH-4A, either Round to Zero or Round to Nearest can be selected for the FPSCR `RM`.
- For CPUs with no FPU, floating-point calculations are processed by a run-time routine, and Round to Nearest is performed.

Round to Nearest

With Round to Nearest, of two approximating floating-point numbers, the value is rounded to that with the closer internal representation.

The direction in which the value is approximated is determined by the value after the final digit of the mantissa. Note that when the value before approximation is exactly between the two approximated floating-point numbers, rounding is performed to the value with `0` as the final digit of the mantissa.

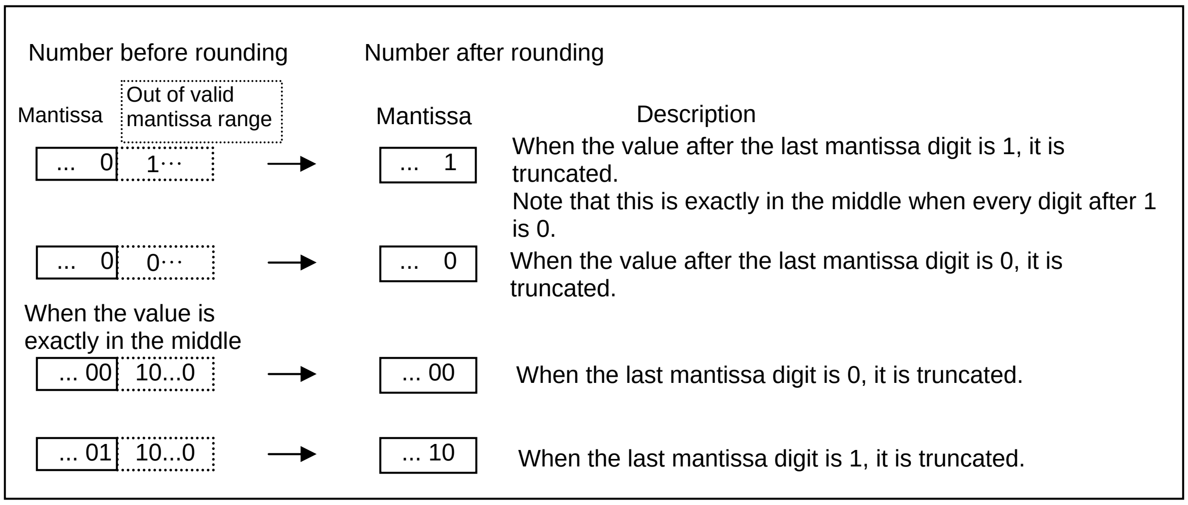

Number before rounding
Number after rounding
Mantissa
Out of valid mantissa range
Mantissa
Description
... 0
1···
... 1
When the value after the last mantissa digit is 1, it is truncated.
Note that this is exactly in the middle when every digit after 1 is 0.
... 0
0···
... 0
When the value after the last mantissa digit is 0, it is truncated.
When the value is exactly in the middle
... 00
10...0
... 00
When the last mantissa digit is 0, it is truncated.
... 01
10...0
... 10
When the last mantissa digit is 1, it is truncated.

Figure 1-7

For example, since 0.1 cannot be expressed properly within the significant digits for the single-precision floating-point number format, it is rounded.

It is represented as `0x3DCCCCCC` for Round to Zero, or `0x3DCCCCCD` for Round to Nearest.

This value will become 0.0999..., the approximated value closest to 0.1.

2. Options for Floating-point Calculations and #pragma

2.1 Conversion from double -> float (DOuble=Float) for SH-2E

This option handles floating-point numbers of the `float` type and `double` type (anything other than that declared as a `long double` type) as used within the program, as a single-precision number. Since the FPU for SH-2E only supports single-precision, double-precision calculation are processed by calling a run-time routine. This option can be specified to enable the FPU to handle floating-point calculations for types other than the `long double` type, thereby increasing floating-point calculation speed.

Note:

This option can also be specified on CPUs with no built-in FPU (SH-1, SH-2, SH-2A, SH2-DSP, SH-3, SH3-DSP, and SH4AL-DSP).

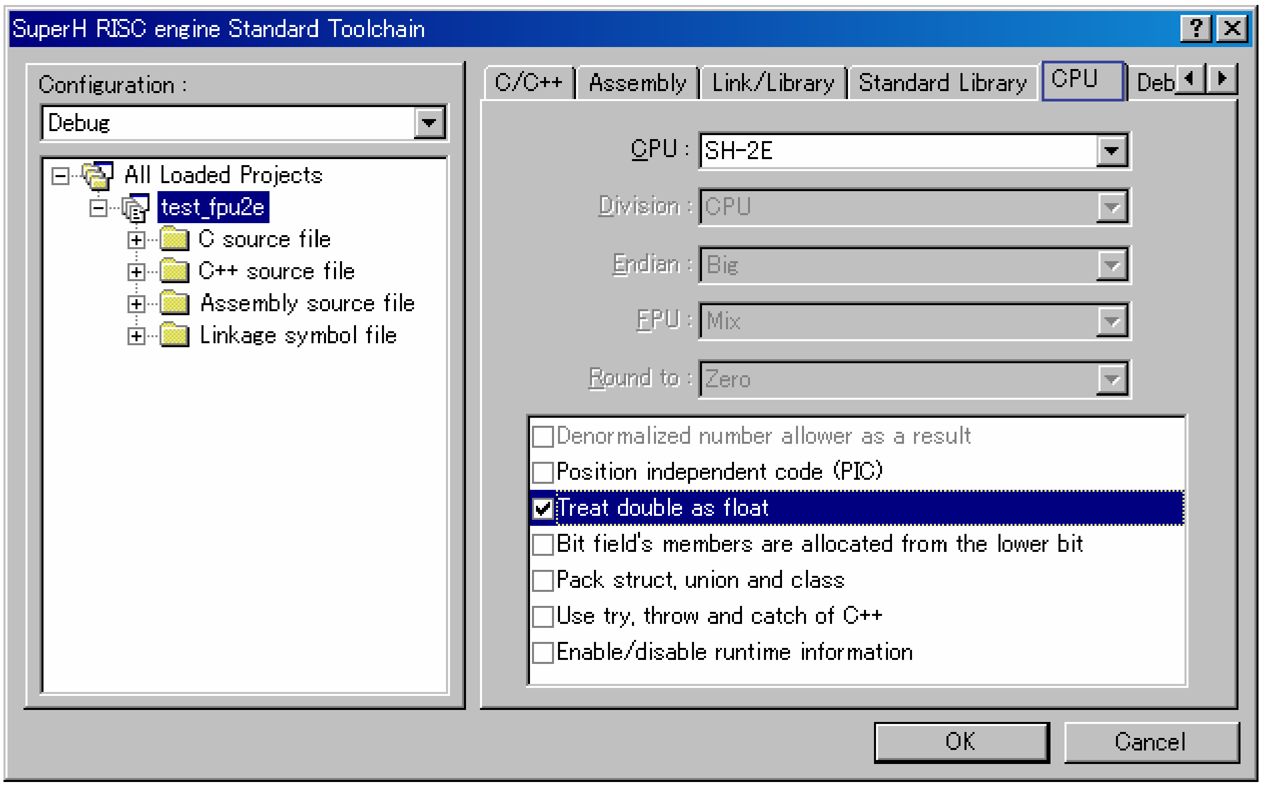
Example:

```
Source code
double func (double a, float b)
{
    return a + b;
}
```

```
Expanded assembly code when double=float is
unspecified (default)
_func:
    STS.L       PR,@-R15
    MOV         R15,R2
    ADD         #8,R2
    MOV.L       @ (4,R2),R1 ;  (part of) a
    MOV.L       @R2,R4     ;  (part of) a
    MOV.L       R1,@-R15
    MOV.L       R4,@-R15
    ADD         #-8,R15
    MOV         R15,R4
    MOV.L       R4,@-R15
    MOV.L       L11,R5     ; __ftod_a
    JSR         @R5
    FMOV.S      FR4,FR0
    ADD         #4,R15
    MOV.L       @ (20,R15),R6
    MOV.L       L11+4,R7   ; __addd_a
    JSR         @R7
    MOV.L       R6,@-R15
    ADD         #20,R15
    LDS.L       @R15+,PR
    RTS
    NOP
L11:
    .DATA.L     __ftod_a
    .DATA.L     __addd_a
```

```
Expanded assembly code when double=float is
specified
_func:
    FADD        FR5,FR4
    RTS
    FMOV.S      FR4,FR0
```

Setting this option in High-Performance Embedded-Workshop (herein as *HEW*)


SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
test_fpu2e
C source file
C++ source file
Assembly source file
Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU | Deb

CPU : SH-2E
Division : CPU
Endian : Big
FPU : Mix
Round to : Zero

- [ ] Denormalized number allower as a result
- [ ] Position independent code (PIC)
- [✓] Treat double as float
- [ ] Bit field's members are allocated from the lower bit
- [ ] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Enable/disable runtime information

Figure 2-1

2.2 Floating-point Calculation Mode (FPu={Single|Double}) for SH2A-FPU, SH-4, and SH-4A

This option handles the floating-point number types used within the program by unifying them. Even though the SH2A-FPU, SH-4, and SH-4A FPUs support both single-precision and double-precision calculation, when calculations with different precisions are performed, the FPSCR PR bit needs to be switched. As such, performance may degrade when floating-point calculations of differing precisions are performed. This can be mitigated by using the FPU option to unify the types of floating-point numbers within a program.

- `Mix` [default]
  Calculation is performed as specified in the C/C++ source.
  The compiler generates code to switch the PR bit in the FPSCR register.
- Single (fpu=single)
  All floating-point calculations are performed using single-precision floating-point numbers (`float` type).
  The compiler does not use the PR bit in the FPSCR register.
- Double (fpu=double)
  All floating-point calculations are performed using double-precision floating-point numbers (`double` type).
  The compiler does not use the PR bit in the FPSCR register.

Note:

Since the compiler does not use the PR bit in the FPSCR register when `Single` or `Double` is selected, the initial value needs to be set by the user program.
Note that since the value of the PR bit for the initial status of the CPU is `0` (single-precision), the calculation is invalid when `Double` is selected and no initial value setting was performed. FPSCR can be set by using the `set_fpscr` embedded function.

Example:

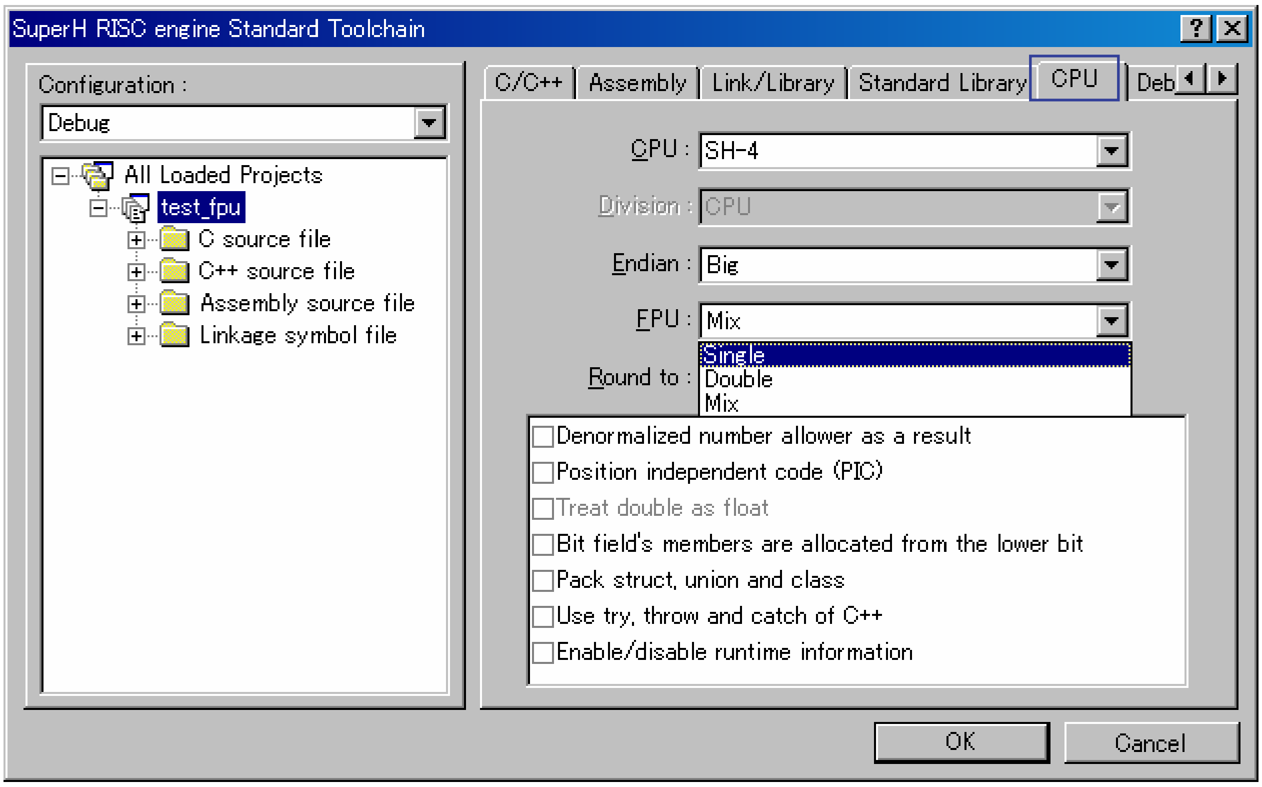
```
Source code
double func (double a, float b)
{
    return a + b;
}
```

```
Expanded assembly code when fpu is not
specified (default)

_func:
    FLDS        FR6,FPUL
    STS         FPSCR,R2
    MOV         #8,R6       ; H'00000008
    SHLL16      R6
    OR          R6,R2
    LDS         R2,FPSCR
    FCNVSD      FPUL,DR0
    RTS
    FADD        DR4,DR0
```

```
Expanded assembly code when
fpu=single is specified
_func:
    FADD        FR5,FR4
    RTS
    FMOV.S      FR4,FR0
```

```
Expanded assembly code when
fpu=double is specified
_func:
    FMOV.S      FR4,FR0
    FMOV.S      FR5,FR1
    RTS
    FADD        DR6,DR0
```

Setting this option in HEW


SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects / test_fpu / C source file / C++ source file / Assembly source file / Linkage symbol file

Tabs: C/C++ | Assembly | Link/Library | Standard Library | CPU | Deb

CPU : SH-4

Division : CPU

Endian : Big

FPU : Mix (Single / Double / Mix)

Round to :

- [ ] Denormalized number allower as a result
- [ ] Position independent code (PIC)
- [ ] Treat double as float
- [ ] Bit field's members are allocated from the lower bit
- [ ] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Enable/disable runtime information

OK | Cancel

Figure 2-2

2.3 Rounding Methods (Round={Zero|Nearest}) for SH2A-FPU, SH-4, and SH-4A

The method used for rounding can be selected for SH2A-FPU, SH-4, and SH-4A. When the rounding method is set, the setting for the RM bit for FPSCR and the setting for the compiler `round` option must be the same. The compiler round option is performed in **Round to**, in Figure 2-3.

- Zero (round=zero) [default]
  Rounding is performed using Round to Zero.
- Nearest (round=nearest)
  Rounding is performed using Round to Nearest.

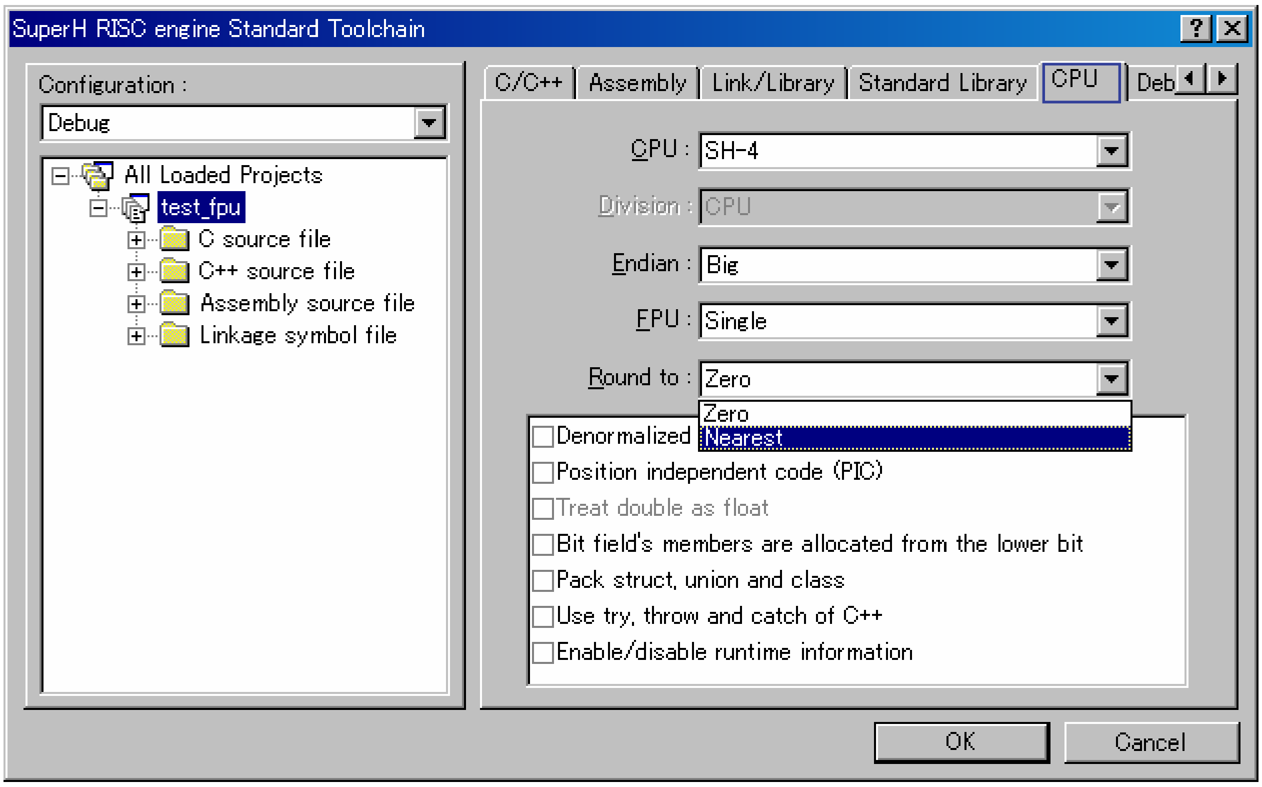
In the initial CPU status, the value of the RM bit is 01 (Round to Zero). When Nearest is specified, set the RM bit to 00 (Round to Nearest).

Since the compiler does not generate code to change the value of the RM bit in FPSCR, the RM bit must be set explicitly by the user program. ~~In the initial CPU status, the value of the RM bit is `00` (Round to Zero). When `Nearest` is specified, set the RM bit to `01` (Round to Nearest).~~ FPSCR can be set by using the `set_fpscr` embedded function.

Example:

```
Source code
float ff = 0.1f;

Expanded assembly code when round=zero is      Expanded assembly code when round=nearest is
specified (default)                            specified
_ff:                                           _ff:
        .DATA.L     H'3DCCCCCC                         .DATA.L     H'3DCCCCCD
```

Setting this option in HEW



CPU : SH-4
Division : CPU
Endian : Big
FPU : Single
Round to : Zero
Zero
Nearest

- [ ] Denormalized
- [ ] Position independent code (PIC)
- [ ] Treat double as float
- [ ] Bit field's members are allocated from the lower bit
- [ ] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Enable/disable runtime information

Figure 2-3

2.4 Handling Non-normalized Numbers (DENormalize={OFF|ON}) for SH-4 and SH-4A

SH-4 and SH-4A can handle non-normalized numbers either as non-normalized numbers, or as `0`. When handling for non-normalized number is set, the setting for the FPSCR `DN` bit and the setting for the compiler `denormalize` option need to be the same. The compiler `denormalize` option can be set by selecting the **Denormalized number allower as a result** check box in Figure 2-4.

- When the check box is not selected (`denormalize=off`) [default]
  Non-normalized numbers are handled as 0.
- When the check box is selected (`denormalize=on`)
  Non-normalized constants are handled as non-normalized numbers.

Since the compiler does not generate code to change the value of the FPSCR `DN` bit, the `DN` bit must be set explicitly by the user program. The value of the FPSCR `DN` in the initial CPU status is `1` (handle non-normalized numbers as 0). To handle non-normalized numbers as non-normalized numbers, explicitly set the `DN` bit to `0` (handle non-normalized number as non-normalized numbers). FPSCR can be set by using the `set_fpscr` embedded function.

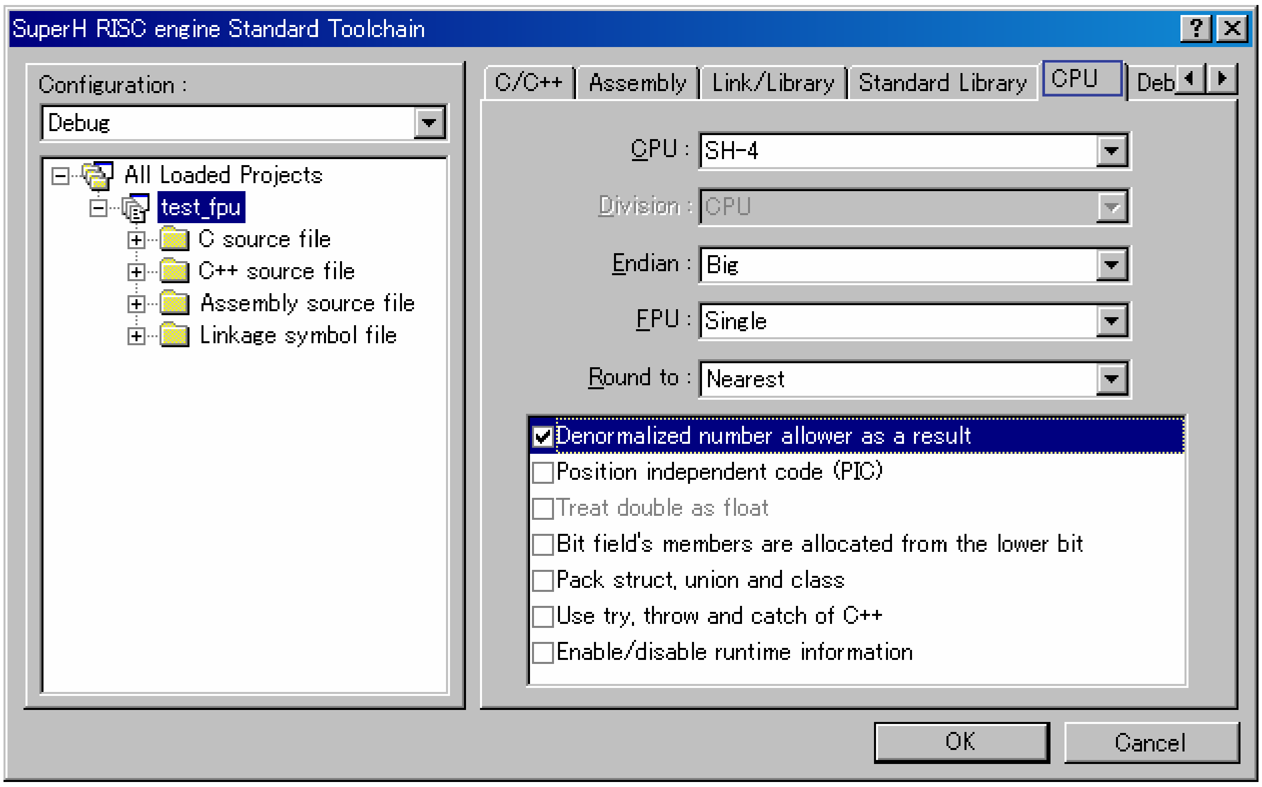
Example:

```
Source code
float ff = 1.0e-38f;
```

```
Expanded assembly code when denormalize=off is
specified (default)
_ff:
         .DATA.L     H'00000000
```

```
Expanded assembly code when denormalize=on is
specified
_ff:
         .DATA.L     H'006CE3EE
```

Setting this option in HEW



CPU : SH-4
Division : CPU
Endian : Big
FPU : Single
Round to : Nearest

- [✓] Denormalized number allower as a result
- [ ] Position independent code (PIC)
- [ ] Treat double as float
- [ ] Bit field's members are allocated from the lower bit
- [ ] Pack struct, union and class
- [ ] Use try, throw and catch of C++
- [ ] Enable/disable runtime information

Figure 2-4

2.5 Converting Floating-point Division into Multiplication (APproxdiv)

This option replaces division by a floating-point constant to multiplication by the inverse of the constant, allowing improved calculation speed. The **Approximate a float-point constant division** check box in Figure 2-5 can be selected to specify `optionapproxdiv`.

This option can be specified regardless of the FPU used.

Note:

Keep in mind that this optimization may change the margin of error for floating-point calculations.

Example: For `cpu=sh4`

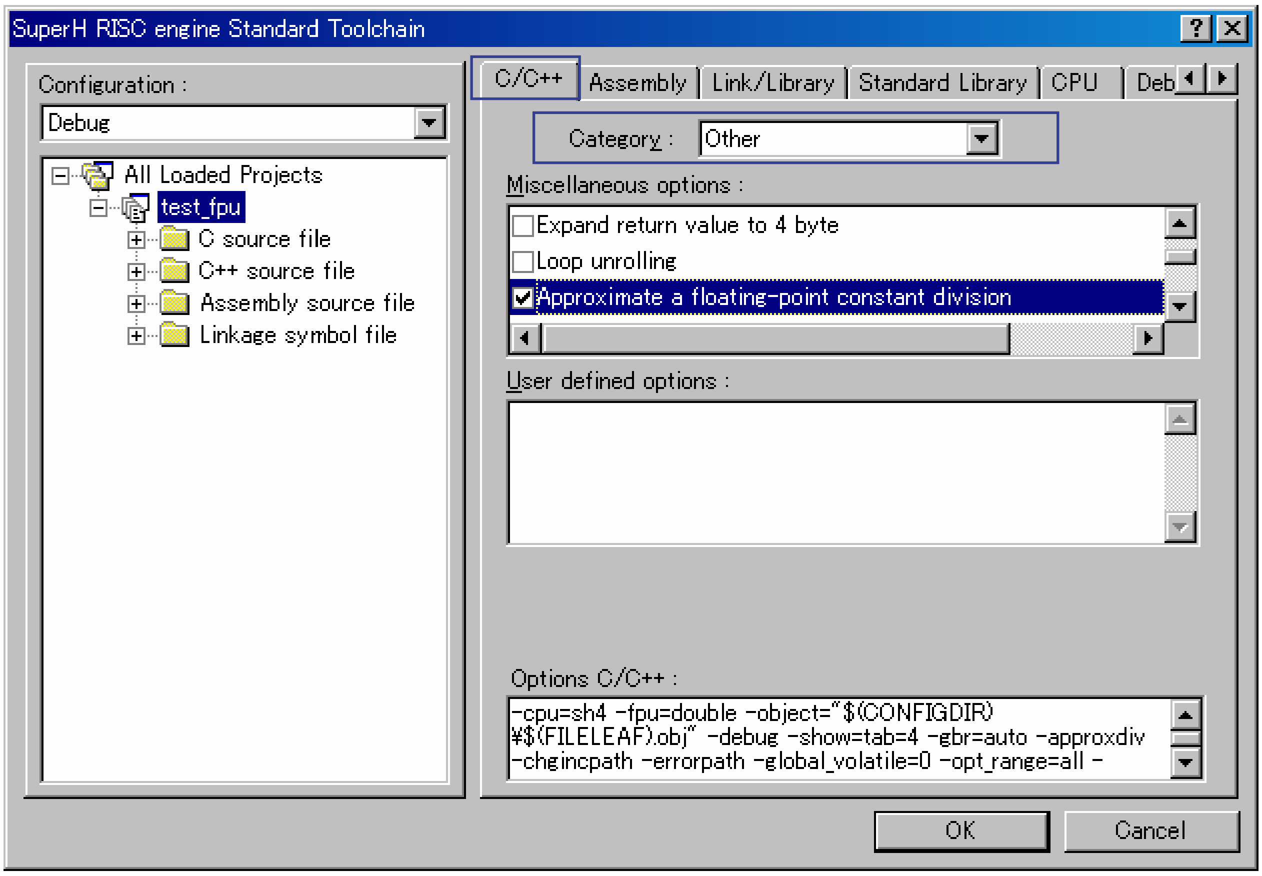
```
Source code
float x;

void f  (float y)
{
    x=y/3.0f;
}
```

Expanded assembly code when approxdiv is not specified (default)	Expanded assembly code when approxdiv is specified

```
_f:
    MOVA         L11,R0
    MOV.L        L11+4,R2    ; _x
    FMOV.S       @R0,FR8
    FDIV         FR8,FR4     ;FR8=H'40400000
                             ;      (3.0)
    RTS
    FMOV.S       FR4,@R2     ; x
L11:
    .DATA.L      H'40400000
    .DATA.L      _x
```
	```
	_f:
	    MOVA         L11,R0
	    MOV.L        L11+4,R2    ; _x
	    FMOV.S       @R0,FR8
	    FMUL         FR8,FR4     ;FR8=H'3EAAAAAA
	                             ;      (0.3333333…)
	    RTS
	    FMOV.S       FR4,@R2     ; x
	L11:
	    .DATA.L      H'3EAAAAAA
	    .DATA.L      _x
	```

In this example, division by 3.0 is converted into multiplication by its inverse.

Setting this option in HEW


SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
test_fpu
C source file
C++ source file
Assembly source file
Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU | Deb

Category : Other

Miscellaneous options :

- [ ] Expand return value to 4 byte
- [ ] Loop unrolling
- [x] Approximate a floating-point constant division

User defined options :

Options C/C++ :

```
-cpu=sh4 -fpu=double -object="$(CONFIGDIR)
¥$(FILELEAF).obj" -debug -show=tab=4 -gbr=auto -approxdiv
-chgincpath -errorpath -global_volatile=0 -opt_range=all -
```

OK | Cancel

Figure 2-5

2.6 Converting Floating-point Division (FDIv)

This option converts integer division to floating-point division. By converting integer division processed by a run-time routine into an FPU division instruction, calculation speed can be improved. The **Change integer division into floating-point** check box in Figure 2-6 can be selected to specify the `optionfdiv` option.

Note:

When `cpu=sh2afpu|sh4|sh4a` and `fpu=double` are specified, conversion is performed when both the divisor and dividend are within 4 bytes. Otherwise, conversion is performed when both the divisor and dividend are within 2 bytes.

Example: For `cpu=sh4` and `fpu=double`

Source code

```
int x;

func (int a, int b)
{
   x = a/b;
}
```

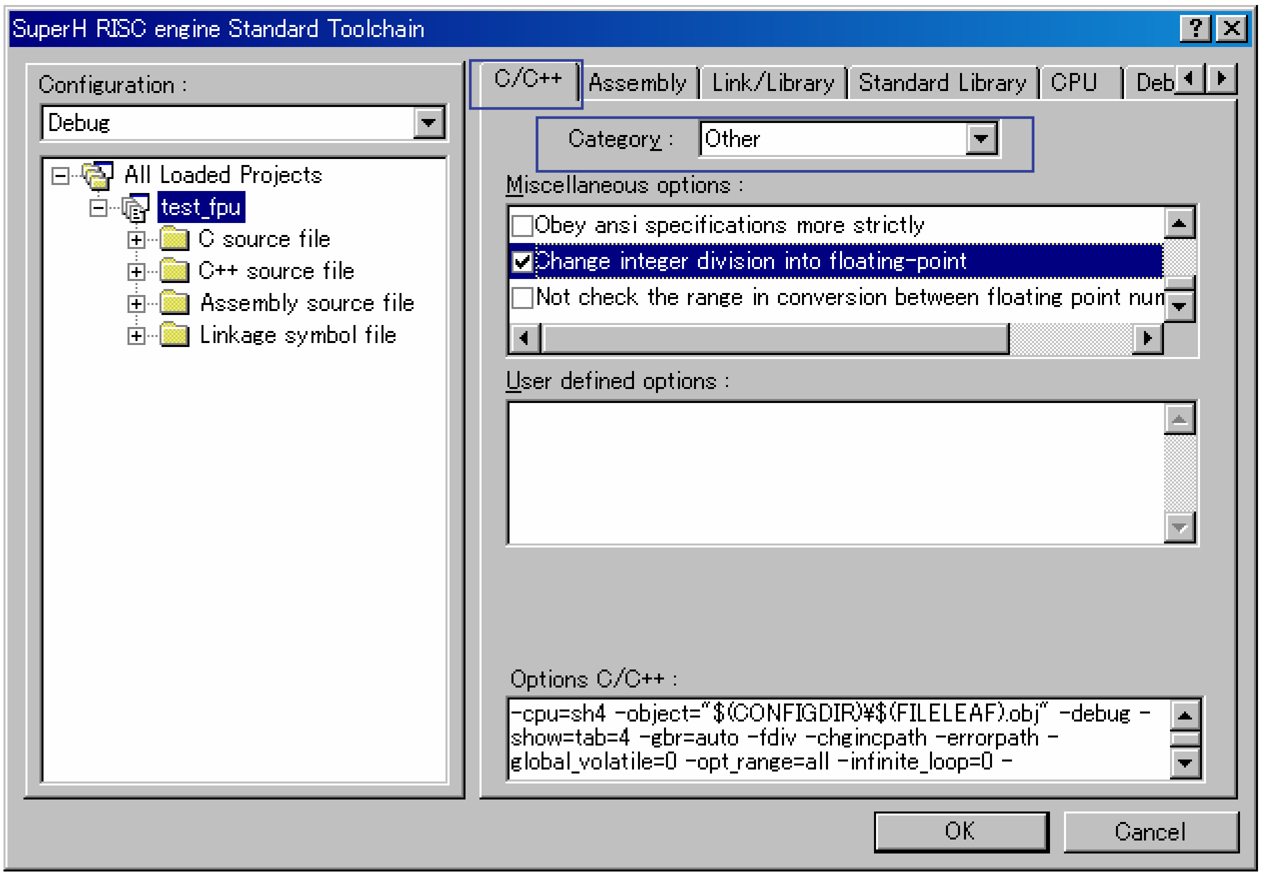
Expanded assembly code when fdiv is not specified (default)

```
_func:
    STS.L       PR,@-R15
    MOV.L       L11+2,R2    ; __divls
    MOV         R4,R1
    MOV.L       L11+6,R6    ; _x
    JSR         @R2
    MOV         R5,R0
    LDS.L       @R15+,PR
    RTS
    MOV.L       R0,@R6      ; x
L11:
    .RES.W      1
    .DATA.L     __divls
    .DATA.L     _x
```

Expanded assembly code when fdiv is specified

```
_func:
    LDS         R4,FPUL
    MOV.L       L11,R6      ; _x
    FLOAT       FPUL,DR6
    LDS         R5,FPUL
    FLOAT       FPUL,DR8
    FDIV        DR8,DR6
    FTRC        DR6,FPUL
    STS         FPUL,R2
    RTS
    MOV.L       R2,@R6      ; x
L11:
    .DATA.L     _x
```

Setting this option in HEW



Category : Other

Miscellaneous options :

- ☐ Obey ansi specifications more strictly
- ☑ Change integer division into floating-point
- ☐ Not check the range in conversion between floating point num

User defined options :

Options C/C++ :

```
-cpu=sh4 -object="$(CONFIGDIR)¥$(FILELEAF).obj" -debug -
show=tab=4 -gbr=auto -fdiv -chgincpath -errorpath -
global_volatile=0 -opt_range=all -infinite_loop=0 -
```

Figure 2-6

2.7 Switching the FPSCR Register Precision Mode (FPScr={Aggressive|Safe}) for SH2A-FPU, SH-4, and SH-4A

This option specifies whether to guarantee the FPSCR precision mode (PR bit) before and after function calls. When `fpscr=aggressivee` (default) is specified, the value of the PR bit before and after function calls is not guaranteed.

When `fpscr=safe` is specified, the value of the PR bit of a called function is guaranteed so that single-precision is always used after the function call.

When `fpscr=aggressivee` (default) is specified, the value of the PR bit when a function call returns is unknown. As such, when a floating-point calculation occurs after a function call, code is always generated to re-set the FPSCR value. However, when `fpscr=safe` is specified, FPSCR is only set when needed, since the PR bit when a function call is returned is guaranteed to be single-precision. As such, `fpscr=safe` generates more efficient code.

When specifying `fpscr=safe`, select the **Change FPSCR register if double data used** check box in Figure 2-7.

Note:

Since this option changes the function interface, it needs to be changed for all files at the same time.
Extra precaution is required when libraries created using a compiler of a previous version are linked.

Example: For `cpu=sh4`

```
Source code
extern void sub (void) ;
extern float f1, f2;

func ()
{
   sub () ;
   f1 =1.0f;
}
```

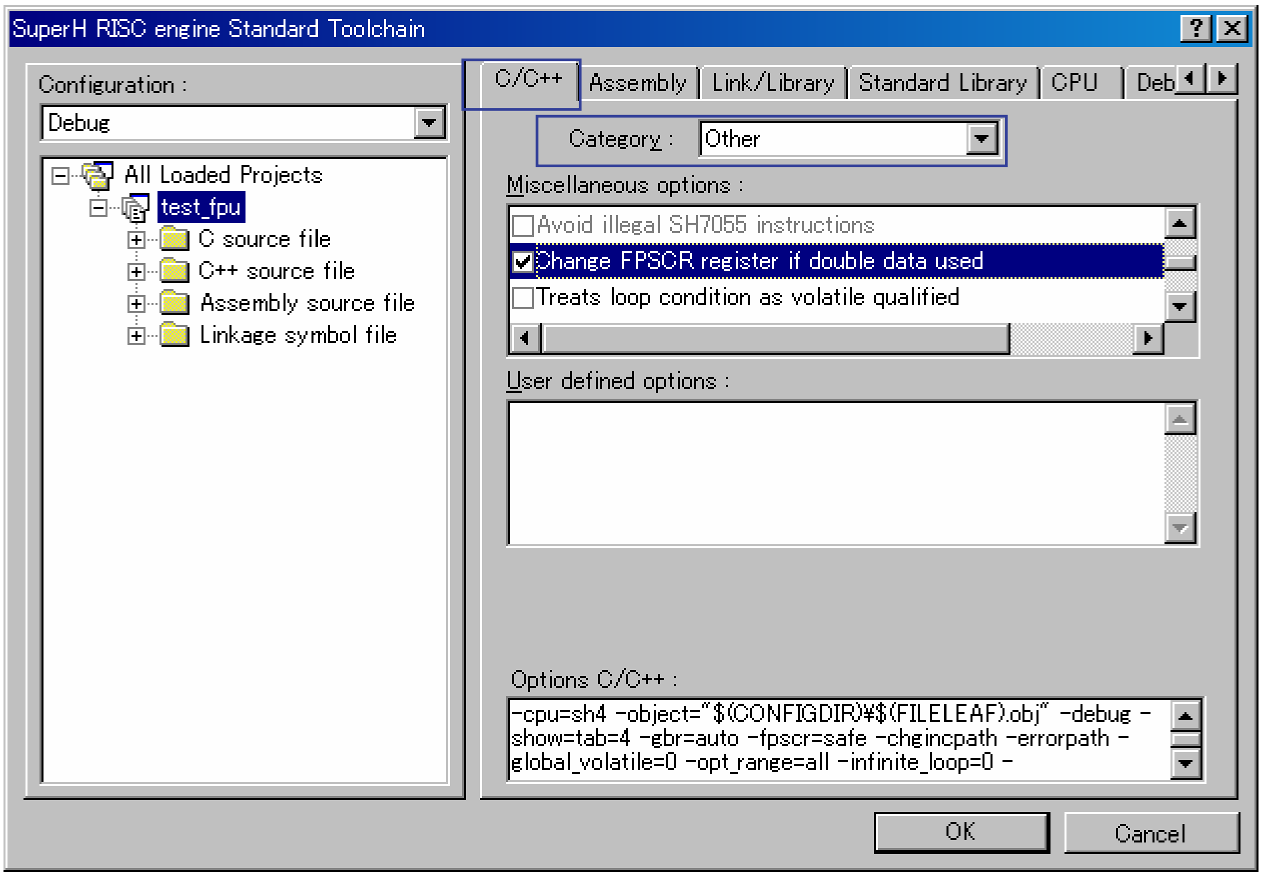
Expanded assembly code when fpscr=aggressive is specified(default)

```
_func:
      STS.L       PR,@-R15
      MOV.L       L11,R1      ; _sub
      JSR         @R1
      NOP
      STS         FPSCR,R4
      MOV.L       L11+4,R6    ; H'FFE7FFFF
      MOVA        L11+8,R0
      MOV.L       L11+12,R5   ; _f1
      AND         R6,R4
      LDS         R4,FPSCR
      FMOV.S      @R0,FR8
      LDS.L       @R15+,PR
      RTS
      FMOV.S      FR8,@R5     ; f1
L11:
      .DATA.L     _sub
      .DATA.L     H'FFE7FFFF
      .DATA.L     H'3F800000
      .DATA.L     _f1
```

Expanded assembly code when fpscr=safe is specified

```
_func:
      STS.L       PR,@-R15
      MOV.L       L11,R1      ; _sub
      JSR         @R1
      NOP
      MOVA        L11+4,R0
      MOV.L       L11+8,R4    ; _f1
      FMOV.S      @R0,FR8
      LDS.L       @R15+,PR
      RTS
      FMOV.S      FR8,@R4     ; f1
L11:
      .DATA.L     _sub
      .DATA.L     H'3F800000
      .DATA.L     _f1
```

Setting this option in HEW



☐ Avoid illegal SH7055 instructions
☑ Change FPSCR register if double data used
☐ Treats loop condition as volatile qualified

Options C/C++ :

```
-cpu=sh4 -object="$(CONFIGDIR)¥$(FILELEAF).obj" -debug -
show=tab=4 -gbr=auto -fpscr=safe -chgincpath -errorpath -
global_volatile=0 -opt_range=all -infinite_loop=0 -
```

Figure 2-7

2.8 Skipping Range Checking during Floating-point Number-integer Conversion (SIMple_float_conv)

This option generates code for skipping range checking for converted values, when type conversion is performed between an unsigned integer type and floating-point number type. By skipping value range checking, calculation speed can be improved. However, when the value before type conversion is neither an integer from 0 to 2147483647 nor a floating-point number from 0.0 to 2147483647.0, the calculation results are invalid. Do not use this option when values outside these ranges may be input.

Example: For `cpu=sh4`

- Conversion from the `float` type to the `unsigned int` type

```
Source code
unsigned long func (float f)
{
    return  ( (unsigned int) f) ;
}
```

```
Expanded assembly code  when simple_float_conv
is not specified (default)
_func:
    MOV         #79,R2   ; 0x0000004F
    SHLL8       R2       ;
    SHLL16      R2       ; 0x4F000000
    LDS         R2,FPUL
    FSTS        FPUL,FR8
    FCMP/GT     FR4,FR8
    BT          L12
    FADD        FR8,FR8 ; when f>=0x4F000000,
    FSUB        FR8,FR4 ; the value before
                        ; conversion
                        ; is (f-0x4F800000)
L12:
    FTRC        FR4,FPUL ;float -> int conversion
    RTS
    STS         FPUL,R0
```

```
Expanded assembly code  when simple_float_conv
is specified
_func:
    FTRC        FR4,FPUL ;float -> int
                         ; conversion
    RTS
    STS         FPUL,R0
```

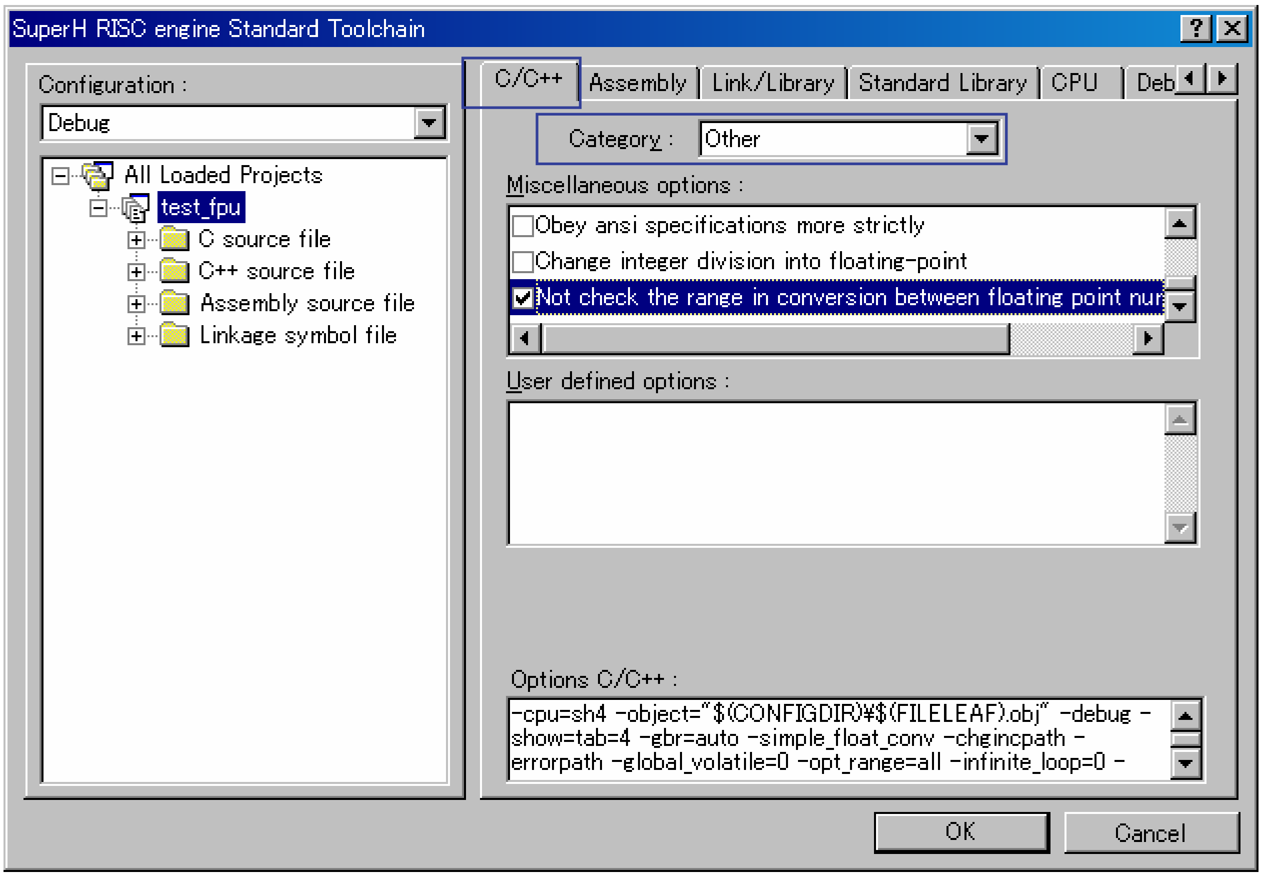
- Conversion from the `unsigned int` type to the `float` type

```
Source code
float func (unsigned int ui)
{
    return  ( (float) ui) ;
}
```

```
Expanded assembly code when simple_float_conv
is not specified (default)
_func:
    LDS         R4,FPUL
    CMP/PZ      R4
    BT/S        L12
    FLOAT       FPUL,FR0 ;int->float
                         ;conversion
    MOVA        L13+2,R
    FMOV.S      @R0,FR9 ; When u >= 0x80000000u,
    FADD        FR9,FR0 ; 0x4F800000 is added to
                        ; the converted value.
L12:
    RTS
    NOP
L13:
    RES.W      1
    DATA.L     H'4F800000
```

```
Expanded assembly code when
simple_float_conv is specified
_func:
    LDS         R4,FPUL
    RTS
    FLOAT       FPUL,FR0 ;int ->float
                         ;conversion
```

Setting this option in HEW



Category : Other

Miscellaneous options :

- ☐ Obey ansi specifications more strictly
- ☐ Change integer division into floating-point
- ☑ Not check the range in conversion between floating point nur

User defined options :

Options C/C++ :

```
-cpu=sh4 -object="$(CONFIGDIR)¥$(FILELEAF).obj" -debug -
show=tab=4 -gbr=auto -simple_float_conv -chgincpath -
errorpath -global_volatile=0 -opt_range=all -infinite_loop=0 -
```

Figure 2-8

2.9 Suppressing Save/Restore for Floating-point Registers (IFUnc, #pragma ifunc)

This option and #pragma prevent floating-point registers from being saved or restored. This function is used with the interrupt function specification (#pragma interrupt).

If a function call exists within the interrupt function, since which register to use cannot be determined from within the called function, all registers including the floating-point register are saved and restored. However, when floating-point calculation are not used within the called function, save and restore can be skipped for the floating-point register. By specifying optionifunc or #pragma ifunc, save and restore can be suppressed for a floating-point register.

When optionifunc is used, the specification takes effect for all functions in the source file, whereas a #pragma ifunc specification takes effect for a particular function. ifunc and #pragma ifunc need to be specified in both the interrupt function and the function called from the interrupt function. When the function called from an interrupt function requires a floating-point instruction, the following compile error is reported:

```
C2843 (E) Illegal floating type used in function
```

Likewise, when a function for which `ifunc` or `#pragma ifunc` is not specified is called from a interrupt function, the following warning is reported:

```
C1029 (W) Function with ifunc calls function-name without ifunc
```

Note that even when this warning is reported, suppression of floating-point register save/restore remains suppressed.

Example: For `cpu=sh4`

```
Source code (#pragma ifunc not specified)
#pragma interrupt  (func)

void sub ()
{
}
void func ()
{
   sub () ;
}
Expanded assembly code
_sub:
          RTS
          NOP
_func:
          MOV.L       R0,@-R15
          MOV.L       R1,@-R15
          MOV.L       R2,@-R15
          MOV.L       R3,@-R15
          MOV.L       R4,@-R15
          MOV.L       R5,@-R15
          MOV.L       R6,@-R15
          MOV.L       R7,@-R15
          STS.L       PR,@-R15
          FMOV.S      FR0,@-R15
          FMOV.S      FR1,@-R15
          FMOV.S      FR2,@-R15
          FMOV.S      FR3,@-R15
          FMOV.S      FR4,@-R15
          FMOV.S      FR5,@-R15
          FMOV.S      FR6,@-R15
          FMOV.S      FR7,@-R15
          FMOV.S      FR8,@-R15
          FMOV.S      FR9,@-R15
          FMOV.S      FR10,@-R15
          FMOV.S      FR11,@-R15
          STS.L       FPUL,@-R15
          STS.L       FPSCR,@-R15
          STC         SSR,@-R15
          STC         SPC,@-R15
          BSR         _sub
          NOP
          LDC         @R15+,SPC
          LDC         @R15+,SSR
          LDS.L       @R15+,FPSCR
          LDS.L       @R15+,FPUL
          FMOV.S      @R15+,FR11
          FMOV.S      @R15+,FR10
          FMOV.S      @R15+,FR9
          FMOV.S      @R15+,FR8
          FMOV.S      @R15+,FR7
          FMOV.S      @R15+,FR6
          FMOV.S      @R15+,FR5
          FMOV.S      @R15+,FR4
          FMOV.S      @R15+,FR3
          FMOV.S      @R15+,FR2
          FMOV.S      @R15+,FR1
          FMOV.S      @R15+,FR0
          LDS.L       @R15+,PR
          MOV.L       @R15+,R7
          MOV.L       @R15+,R6
          MOV.L       @R15+,R5
          MOV.L       @R15+,R4
          MOV.L       @R15+,R3
          MOV.L       @R15+,R2
          MOV.L       @R15+,R1
          MOV.L       @R15+,R0
          RTE
```

```
Source code (#pragma ifunc specified)
#pragma interrupt  (func)
#pragma ifunc  (sub,func)
void sub ()
{
}
void func ()
{
   sub () ;
}
Expanded assembly code
_sub:
          RTS
          NOP
_func:
          MOV.L       R0,@-R15
          MOV.L       R1,@-R15
          MOV.L       R2,@-R15
          MOV.L       R3,@-R15
          MOV.L       R4,@-R15
          MOV.L       R5,@-R15
          MOV.L       R6,@-R15
          MOV.L       R7,@-R15
          STS.L       PR,@-R15
          STC         SSR,@-R15
          STC         SPC,@-R15
          BSR         _sub
          NOP
          LDC         @R15+,SPC
          LDC         @R15+,SSR
          LDS.L       @R15+,PR
          MOV.L       @R15+,R7
          MOV.L       @R15+,R6
          MOV.L       @R15+,R5
          MOV.L       @R15+,R4
          MOV.L       @R15+,R3
          MOV.L       @R15+,R2
          MOV.L       @R15+,R1
          MOV.L       @R15+,R0
          RTE
          NOP
```

(1) Specifying `#pragma ifunc`

Format: `#pragma ifunc [(]`*function-name*`[)]`

Description: Suppresses floating-point register save/restore for the function specified by *function-name*.

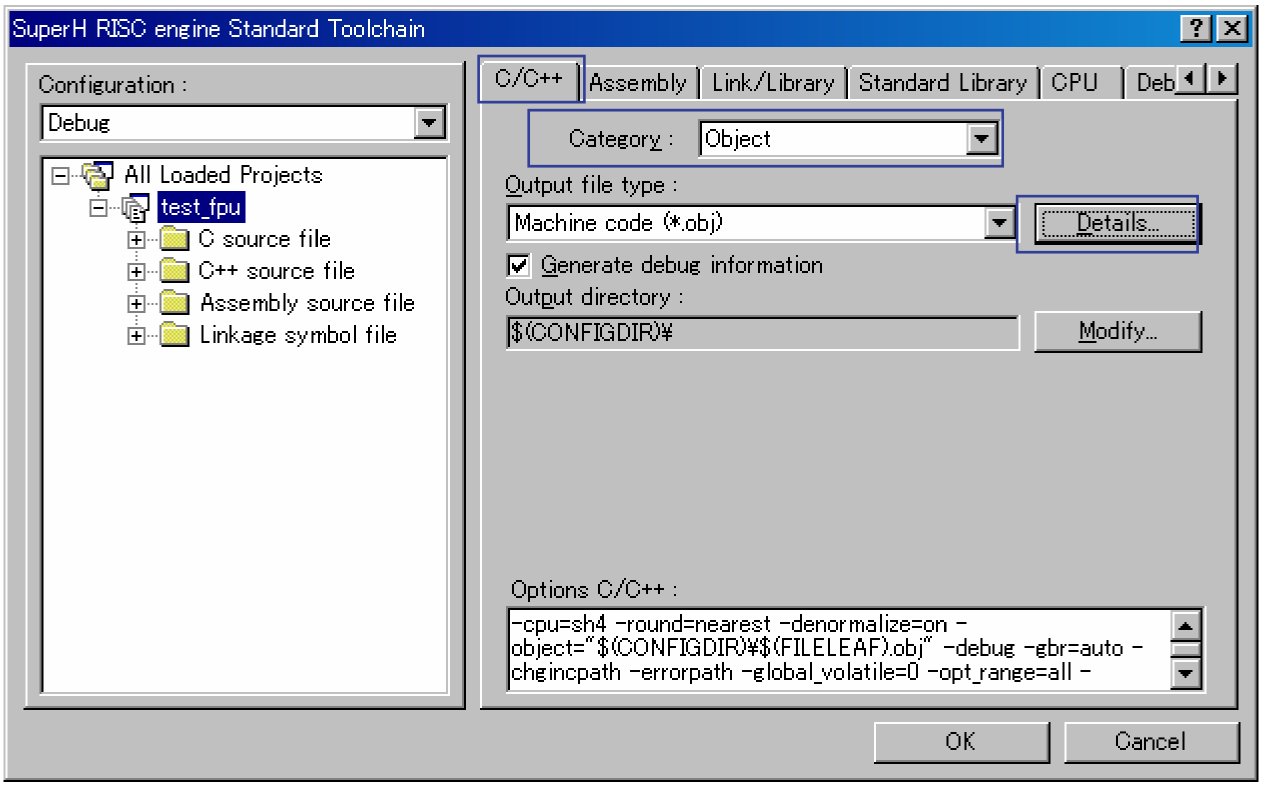
Notes:

Specify `#pragma ifunc` before the function declaration.

A compile error will occur when a floating-point number is used within a function for which `#pragma ifunc` is specified.

(2) Specifying `optionifunc`

This is specified on a file basis. A compile error will occur when this is specified for a source program for which floating-point instructions are generated.



Generate debug information ✓

Output directory : $(CONFIGDIR)¥

```
-cpu=sh4 -round=nearest -denormalize=on -
object="$(CONFIGDIR)¥$(FILELEAF).obj" -debug -gbr=auto -
chgincpath -errorpath -global_volatile=0 -opt_range=all -
```

Figure 2-9

Choose the **Code generation** tab, and then select the **Use no FPU instructions**.

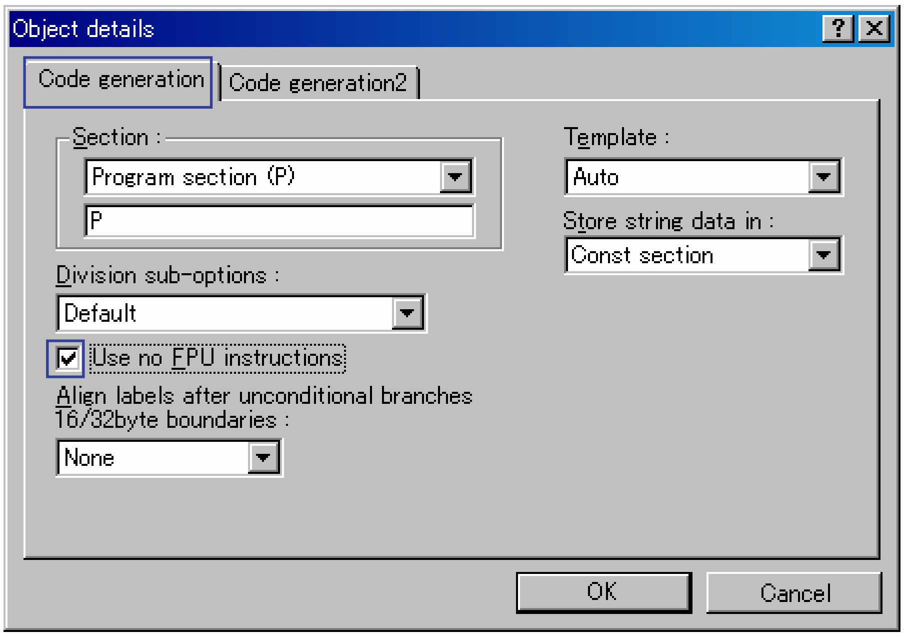


Figure 2-10

3. Efficient Programming Techniques

3.1 Using Floating-point Instructions

Overview

To use floating-point instructions `FABS` (SH-2E, SH2A-FPU, SH-4, and SH-4A) or `FSQRT` (SH2A-FPU, SH-4, and SH-4A) for single-precision, include the `mathf.h` include file, and then call the `fabsf` or `sqrtf` single-precision floating-point function. For double-precision, include the `math.h` include file, and then call the `fabs` or `sqrt` double-precision floating-point function.

Description

Perform the following to use the single-precision floating-point instruction `FABS` (SH-2E, SH2A-FPU, SH-4, and SH-4A) or `FSQRT` (SH2A-FPU, SH-4, and SH-4A):
(a) Include mathf.h.
(b) Call the fabsf function (FABS) or sqrtf function (FSQRT).

Perform the following to use the double-precision floating-point instruction `FABS` (SH2A-FPU, SH-4, and SH-4A) or `FSQRT` (SH2A-FPU, SH-4, and SH-4A):
(a) Include math.h.
(b) Call the fabs function (FABS) or sqrt function (FSQRT).

Usage examples

In the BEFORE example, since `mathf.h` is not included, the compiler does not recognize it as a standard function, and calls the `fabsf` function from the library. If `mathf.h` is included, the compiler can recognize that the function is for the `FABS` instruction, and generate a `FABS` instruction directly.

Note:
The `mathf.h` header is not an ANSI-standard C library function.

```
Source code - BEFORE
float fabsf (float) ;

float f (float x, float y) {
     return fabsf (x) +fabsf (y) ;
}
Assembly code - BEFORE
_f:
          STS.L       PR,@-R15
          FMOV.S      FR14,@-R15
          FMOV.S      FR15,@-R15
          MOV.L       L11+2,R1   ; _fabsf
          JSR         @R1
          FMOV.S      FR5,FR15
          FMOV.S      FR0,FR14
          MOV.L       L11+2,R4   ; _fabsf
          JSR         @R4
          FMOV.S      FR15,FR4
          FADD        FR0,FR14
          FMOV.S      FR14,FR0
          FMOV.S      @R15+,FR15
          FMOV.S      @R15+,FR14
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _fabsf
```

```
Source code - AFTER
#include <mathf.h>

float f (float x, float y) {
     return fabsf (x) +fabsf (y) ;
}
Assembly code - AFTER
_f:
          FABS        FR4
          FABS        FR5
          FADD        FR5,FR4
          RTS
          FMOV.S      FR4,FR0
```

4. Frequently Asked Questions

4.1 Floating-point Calculation Results

Q:

Why does a floating-point calculation not return the expected value?

A1:

The expected value may not be returned due to a rounding error.
For example, since the variable `a` in List 4-1 is of the `double` type, it is represented internally as a floating-point number. When the `round=zero` option is selected, the internal representation of this variable is `0x4023FFFFFFFFFFFF`. This value is not 10, but instead 9.9999..., the closest approximated value to 10. As such, the truncated 9s after the decimal point are substituted for variable `b` in List 4-1.

```
(Sample program)
double a = 0.1 * 100;
int b;

void func(void)
{
   b = (int) a;
}
```

List 4-1

This phenomenon occurs due to rounding errors for the floating-point number representation. As such, there is no fundamental solution. Write code as follows, as a workaround.

```
(Code without error in mind)
float f;

if ( f == 0.1f ) {
       .
       .
       .
}
```

```
(Code with error in mind)
const float s = 1.0e-10f;
float f;

if ((0.1f-s) <= f && f <= (0.1f+s)) {
       .
       .
       .
}
```

List 4-2

A2:

For SH2A-FPU, SH-4, and SH-4A, since the option and value for FPCSR may not correspond, check the following:

1. The correspondence between the compiler `fpu` option and FPCSR `PR` bit value.
2. The correspondence between the compiler `denormalize` option and FPCSR `DN` bit value (SH-4 and SH-4A only).
3. The correspondence between the compiler `round` option and FPCSR `RM` bit value (SH-4 and SH-4A only).

Floating-point calculation mode

There are two possible `fpu` options: `fpu=single` and `fpu=double`.
When `fpu=single` is selected, all floating-point data is handled as single-precision floating-point data.
When `fpu=double` is selected, all floating-point data is handled as double-precision floating-point data.
When no `fpu` option is used, floating-point data is used according to the declaration type of the source code.

When no `fpu` option is used, the compiler generates code that changes the `PR` bit according to the corresponding precision during floating-point calculations, but when `fpu=single` or `fpu=double` is selected, the compiler does not generate any code that accesses the `PR` bit.

Meanwhile, the `PR` bit of the FPSCR register is initialized to `0` during reset.
As such, when the `fpu` option is not used or `fpu=single` is selected, operation is performed correctly without concern for the `PR` bit, but when `fpu=double` is selected, the `PR` bit is changed to `1` before the FPU calculation.

Handling non-normalized numbers

When selecting the `denormalize=on` option, make sure that the FPSCR `DN` bit is set to `0`.
When selecting the `denormalize=off` option, make sure that the FPSCR `DN` bit is set to `1`.

Rounding methods

When selecting the `round=zero` option, make sure that the FPSCR `RM` bit is set to `01`.
When selecting the `round=nearest` option, make sure that the FPSCR `RM` bit is set to `00`.

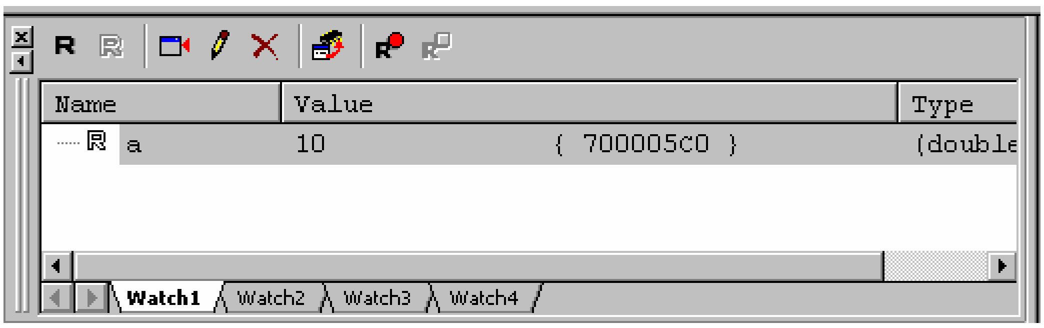
4.2 Values for Floating-point Numbers in the Watch Window

Q:

Why do the values of floating-point numbers differ between that displayed in the watch window and the actual value?

A:

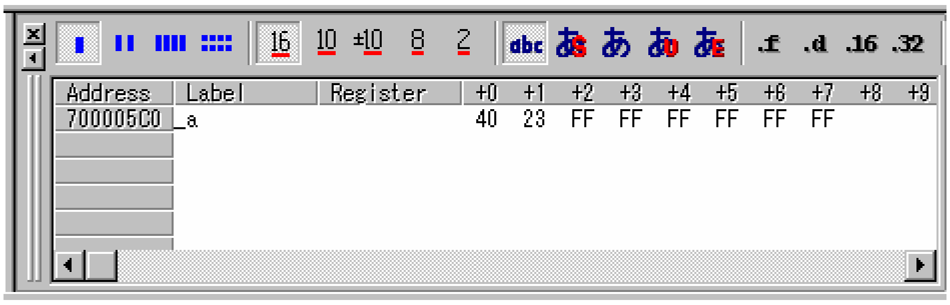
The value of floating-point numbers displayed in the watch window are approximate.
For example, the actual value of a floating-point number with the internal representation `0x4023FFFFFFFFFFFF` is 9.9999..., but the value displayed in the watch window (Figure 4-1) is 10.



Name	Value	Type
a	10 { 700005C0 }	(double

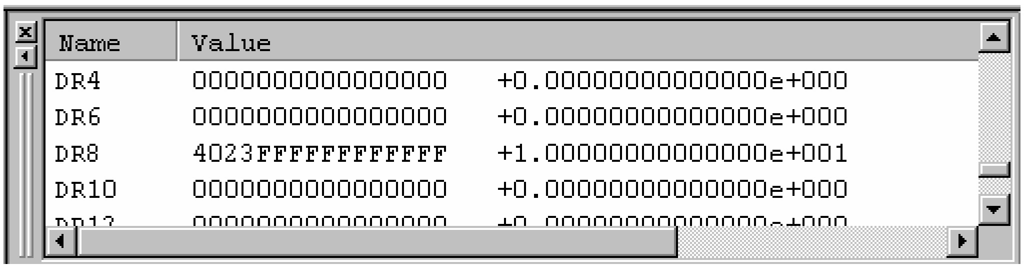
Figure 4-1

The internal representation of the actual value can be checked in the memory window (Figure 4-2) or register window (Figure 4-3).



```
Address   Label  Register  +0 +1 +2 +3 +4 +5 +6 +7 +8 +9
700005C0  _a               40 23 FF FF FF FF FF FF
```

Figure 4-2



```
Name   Value
DR4    0000000000000000   +0.0000000000000e+000
DR6    0000000000000000   +0.0000000000000e+000
DR8    4023FFFFFFFFFFFF   +1.0000000000000e+001
DR10   0000000000000000   +0.0000000000000e+000
```

Figure 4-3

Website and Support <website and support,ws>

Renesas Technology Website
http://japan.renesas.com/

Inquiries
http://japan.renesas.com/inquiry
csc@renesas.com

Revision Record <revision history,rh>

		Description	
Rev.	Date	Page	Summary
1.00	Jun.1.07	—	First edition issued