To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

SuperH RISC engine C/C++ Compiler Package

Application notes: [Introduction guide] Sample file Guide for SH-3, SH-4, and SH-4A

This document explains precautions for generating files and performing initial coding in High-performance Embedded Workshop (herein as *HEW*), for SuperH RISC engine C/C++ compiler V.9.

Table of contents

1. Generating a Sample Program

1.1 Project Generator Settings

This document explains the sample program generated when the following operations are performed in the project generator (started in HEW by choosing **New workspace** from the **File** menu). Note that SH7750 selected for CPU types is selected for illustration purpose only.

(1) Create a new workspace

For the project type, choose **Application**.

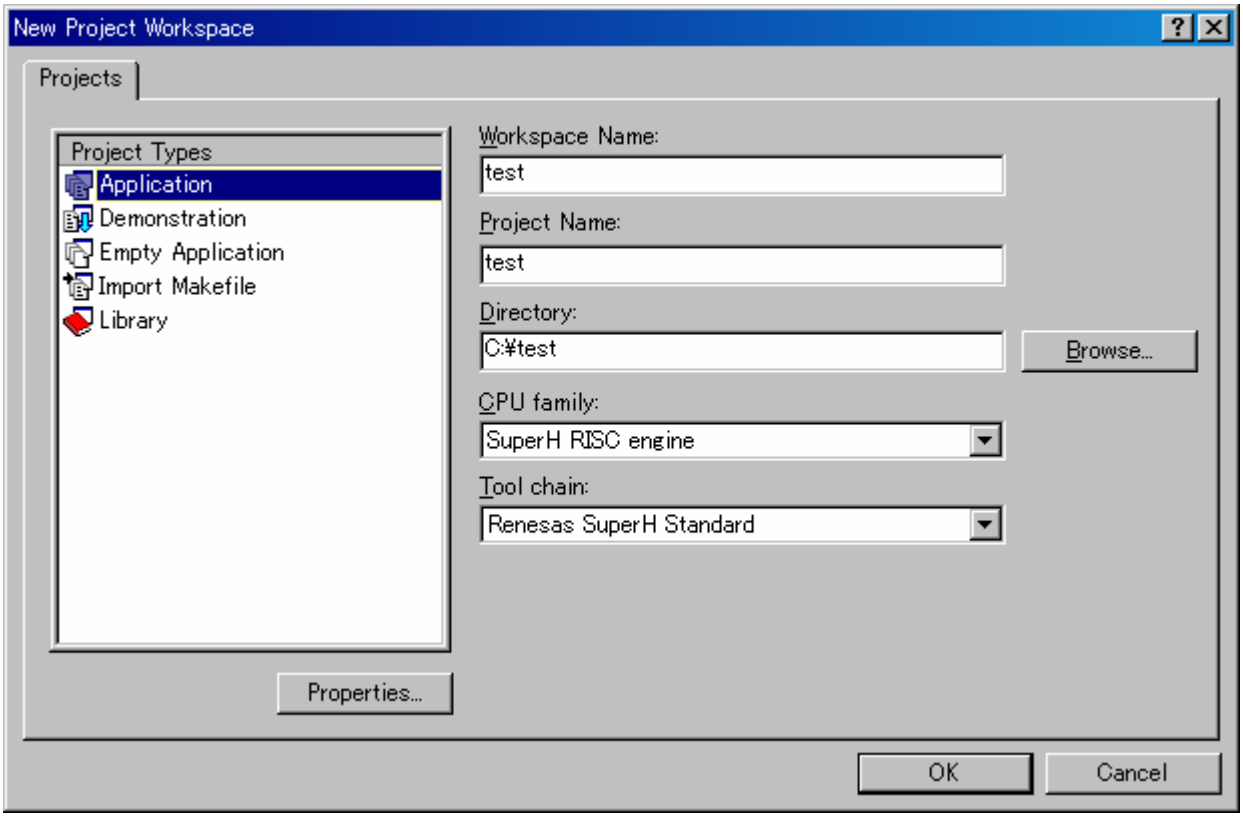

Figure 1-1

(2) Select the CPU

For **CPU Series**, select **SH-4**

For **CPU Type**, select **SH7750**.

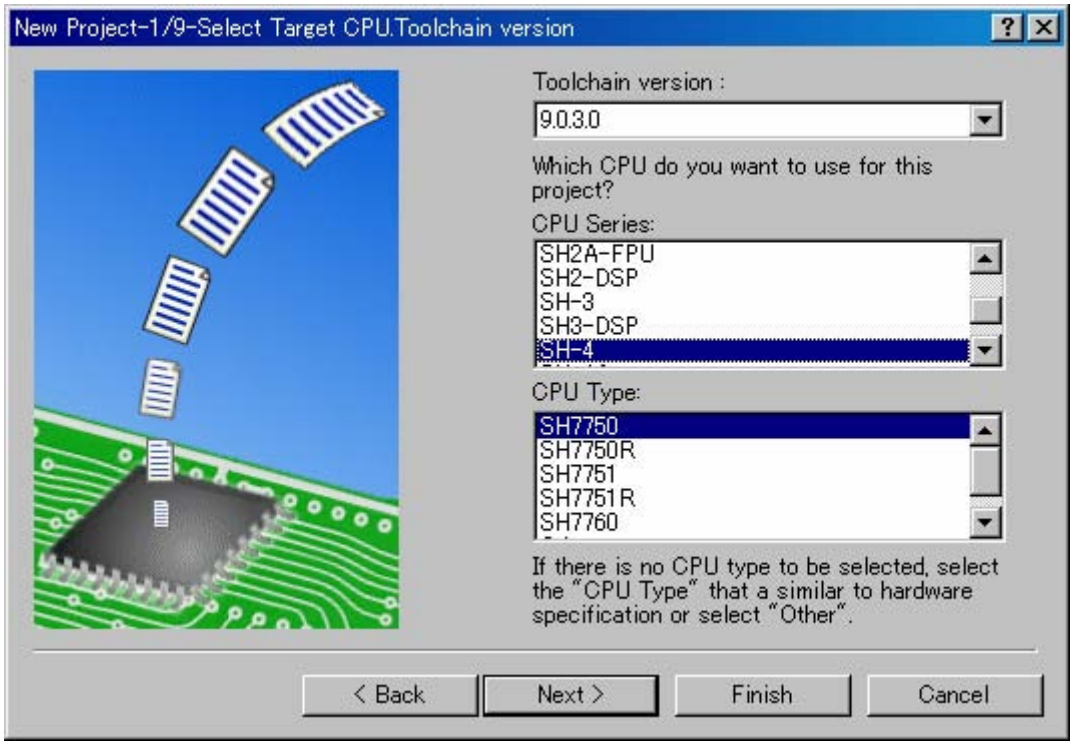

Figure 1-2

Notes:

- The **CPU Series** setting is reflected in the **CPU** page of the SuperH RISC engine Standard Toolchain dialog box (herein as *Toolchain dialog box*).
- The **CPU Type** setting is reflected in the contents of `intprg.src`, `vecttbl.src`, `iodefine.h`, and `vect.inc`, and the memory placement setting for the optimization linkage editor. If the CPU to be selected does not exist, use DeviceUpdater to add the CPU type. DeviceUpdater can be downloaded from the Renesas web site.

(3) Optional settings

Proceed with the default settings.

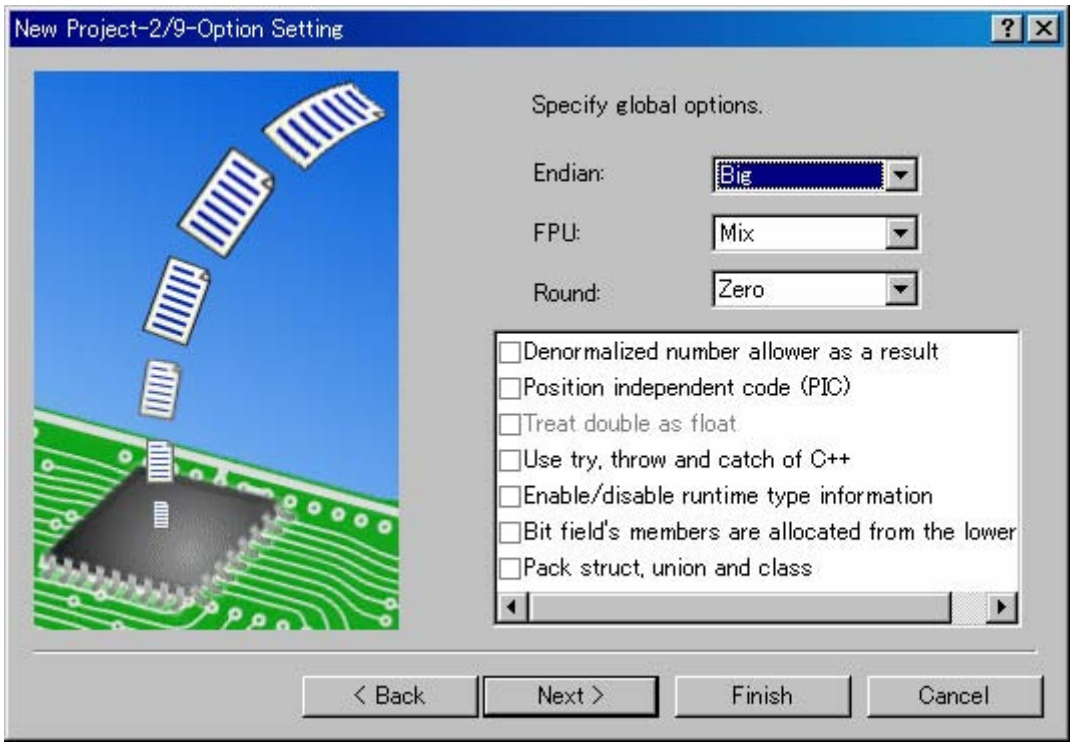

Figure 1-3

Note:

- The settings in this dialog box specify the options set for all projects. The setting items are reflected in the **CPU** page of the Toolchain dialog box. The items that can be selected differ depending on the selection from *(2) Select the CPU*.

(4) Set the generation file

Select **Use I/O library**.

Specify `20 for` **Number of I/O Streams**.

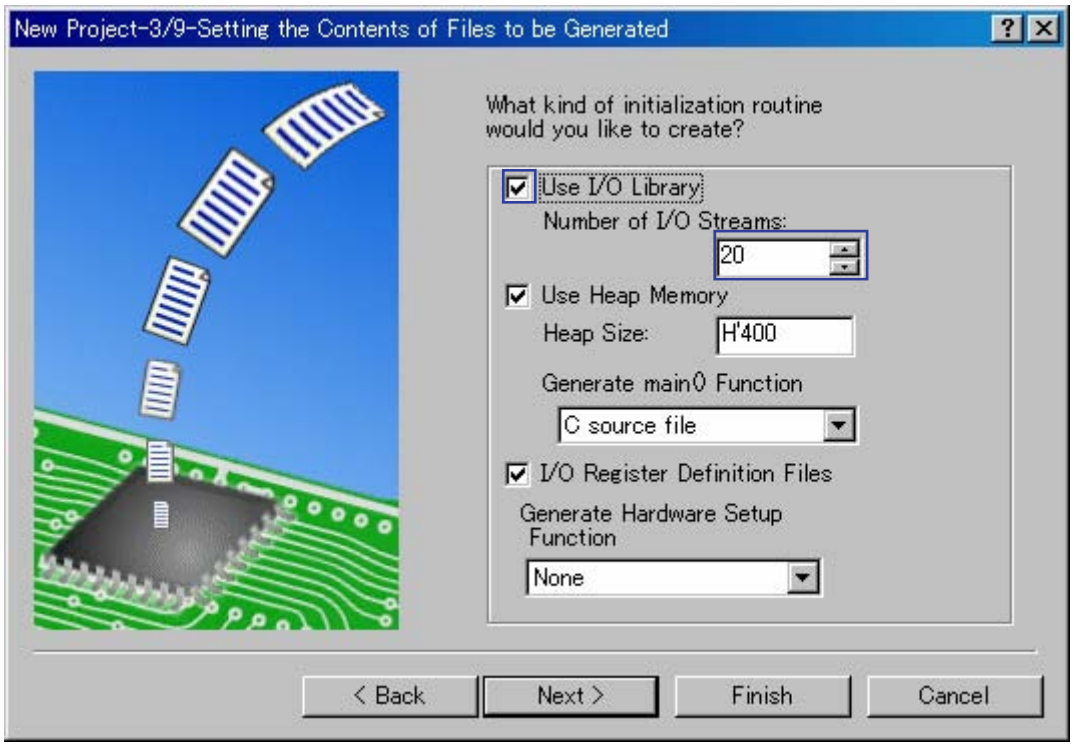

Figure 1-4

Notes:

- When **Use I/O library** is selected, the low-level I/O-related interface routines (`open`, `close`, `write`, `read`, and `lseek`) and sample programs (`lowlvl.src`, `lowsrc.c`, and `lowsrc.h`) for the standard library initialization programs (`_INIT_IOLIB` and `_CLOSEALL`) are generated.
- The value set for **Number of I/O Streams** is reflected in lowsrc.h.
- When **Use Heap Memory** is selected, sample programs (`sbrk.h` and `sbrk.c`) for the low-level memory-management interface routine (`sbrk`) are generated.
- The value set for **Heap Size** is reflected in `sbrk.h`.
- The **Generate main() Function** setting is used to generate the main function (C source file or C++ source file) and abort function template.
- When **I/O Register Definition File** is selected, `iodefine.h` is generated.
- The **Generate Hardware Setup Function** setting is used to generate `hwsetup.c`, `hwsetup.cpp`, and `hwsetup.src`.
  In the hardware setup function, perform the necessary hardware initialization processing for the target system, including bus state controller (BSC) initialization and serial initialization. Note that if the C/C++ languages are used for programming, neither the languages nor the compile option can control when a stack is used. As such, when a stack area is reserved in SDRAM or other memory that requires initialization, the memory may end up being accessed before initialization. In this case, use assembly language to perform memory initialization before program execution in C.

(5) Set the standard library

Proceed with the default settings.

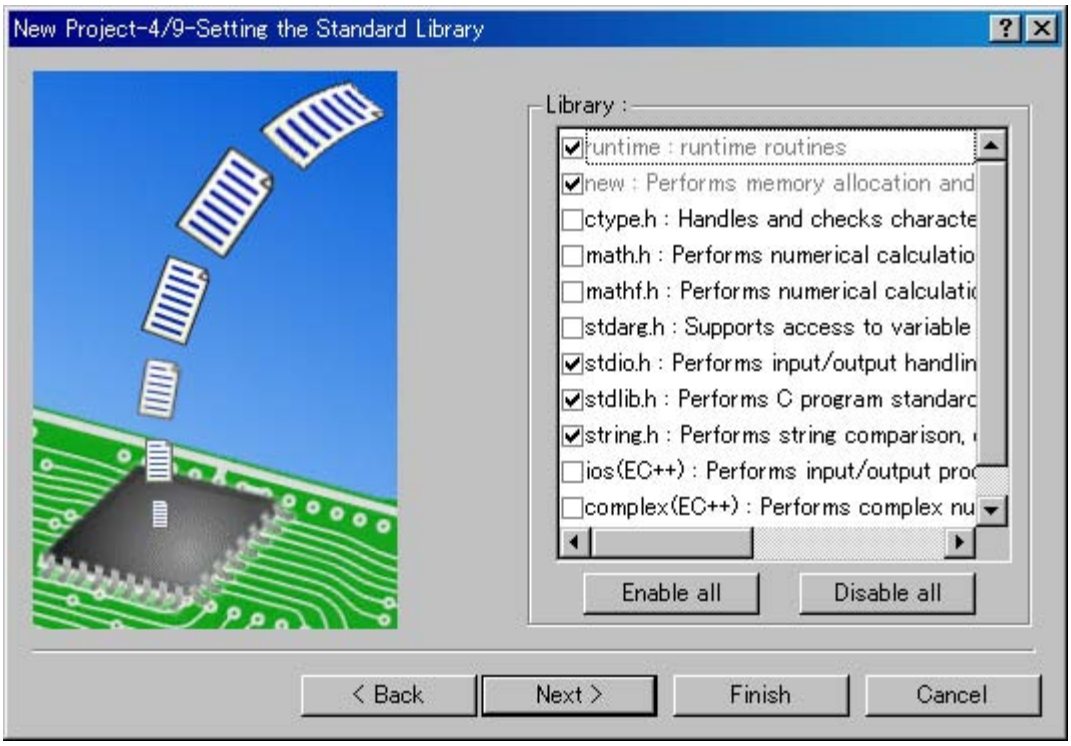

Figure 1-5

Notes:

- This dialog box is used to select the library to be configured by the standard library configuration tool.
- The settings in this dialog box are reflected in the **Standard Library** page of the Toolchain dialog box.

(6) Set the stack area

Proceed with the default settings.

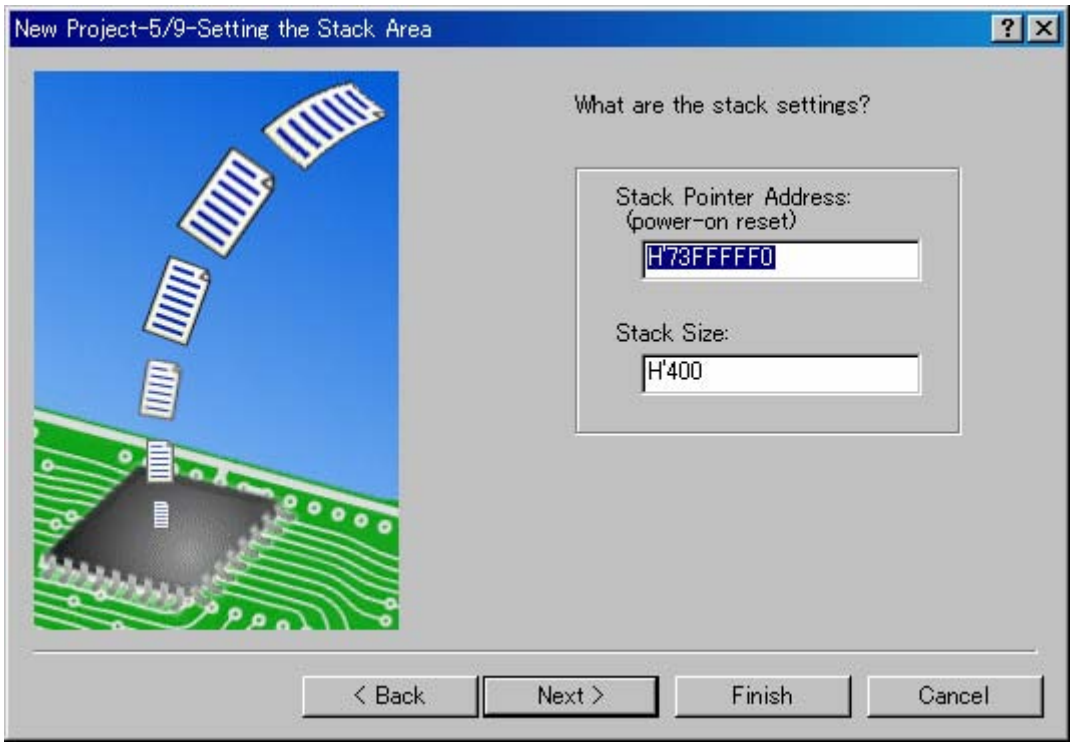

Figure 1-6

Notes:

- The **Stack Pointer Address** setting is reflected in the S section settings in the optimization linkage editor.
- The **Stack Size** setting is reflected in `stacksct.h`.
  Note that when **Vector Definition Files** is selected in *(7) Set the vector*, `stacksct.h` is not generated.

(7) Set the vector

Proceed with the default settings.

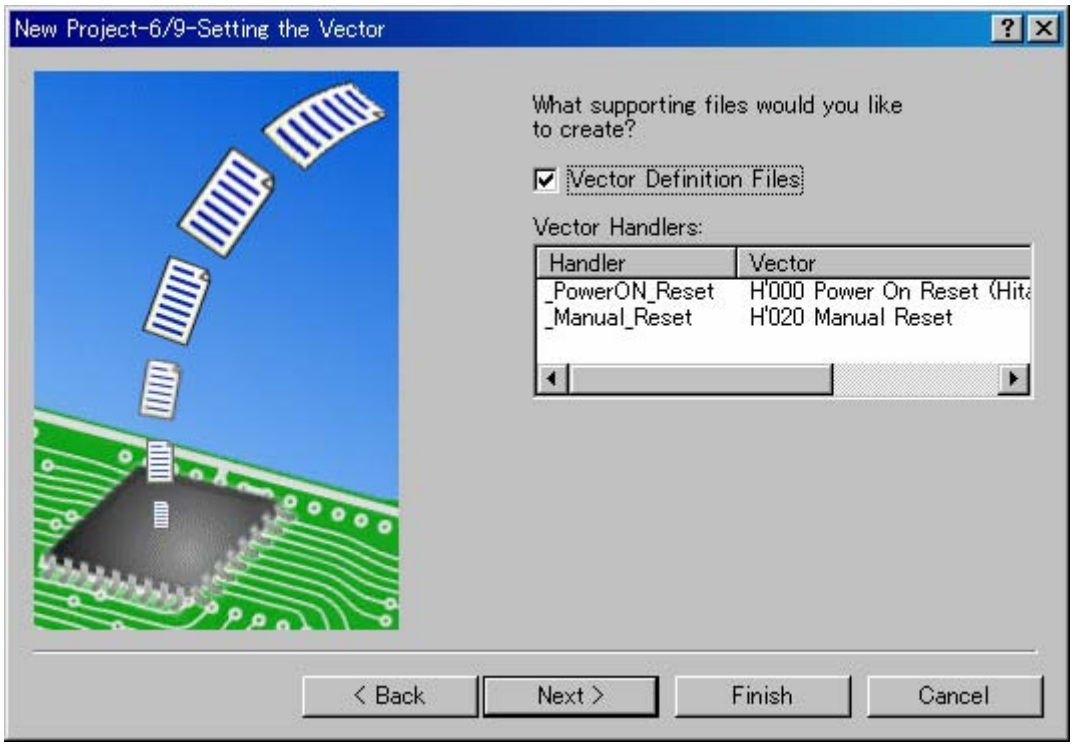

Figure 1-7

Note:

- When **Vector Definition Files** is selected, `env.src`, `intprg.src`, `resetprg.c`, `stacksct.h`, `vecttbl.src`, `vect.inc`, and `vhandler.src` are generated.

(8) Set the debugger target

Proceed with the default settings.

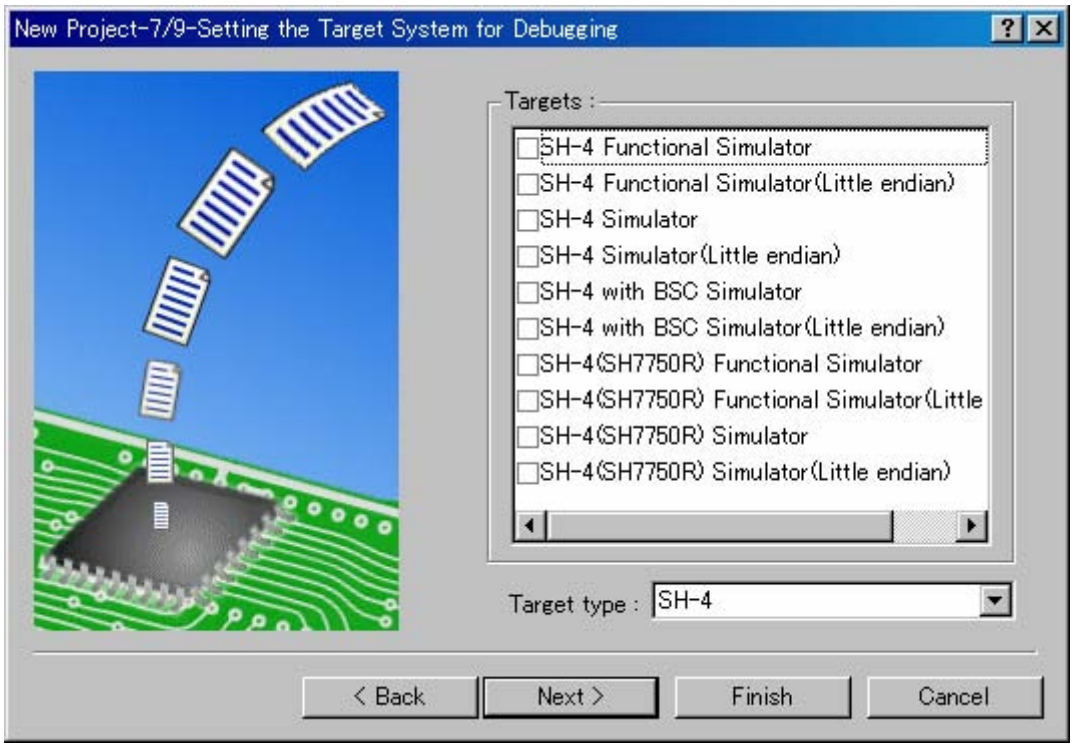

Figure 1-8

(9) Change the name of the generation file

Select **Finish**.

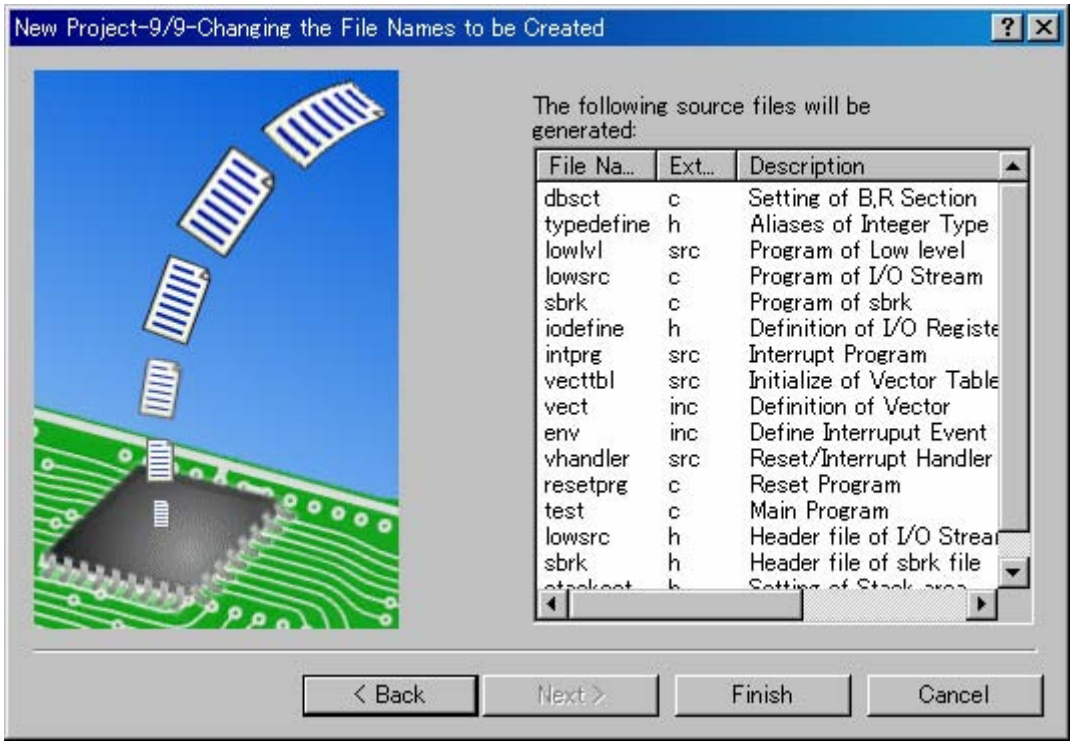

Figure 1-9

1.2 List of Generated Files

The following table lists the sample files auto-generated by the project generator.

Table 1-1 List of auto-generated sample files (1)

`intprg.src`	Interrupt function • Defines the interrupt function (dummy). • Generated according to the specification in *(7) Vector Definition Files*. For details, see *3.4 Exception Processing Routine (intprg.src)*.
`lowlvl.src`	Low-level I/O interface routine • Defines `_charput` and `_charget`, which are called from the low-level interface routine (`write` and `read`). • This program only runs in the simulator. • Generated according to the specification in *(4) Use I/O library*. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)*.
`vecttbl.src`	Vector table • Defines the exception processing vector table. • Generated according to the specification in *(7) Vector Definition Files*. For details, see *2.1 Reset Handlers (vhandler.src, vecttbl.src, env.src, env.inc)*.
`vhandler.src`	Exception processing handlers • Defines the exception processing handlers. • Generated according to the specification in *(7) Vector Definition Files*. For details, see *2.1 Reset Handlers (vhandler.src, vecttbl.src, env.src, env.inc)*.
`dbsct.c`	Memory initialization target specification • Defines RAM initialization and targets for transfer processing from ROM to RAM areas. For details, see *4.1 Memory Initialization Function _INTSCT* (dbsct.c).
`lowsrc.c`	I/O low-level interface routine • Defines the low-level interface routines (`write`, `read`, `open`, `close`, and `lseek`). • This program is for simulators that only support standard I/O functions. • Generated according to the specification in *(4) Use I/O library*. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)*.
`resetprg.c`	Reset function • Defines the reset function (`PowerON_Reset`). • Generated according to the specification in *(7) Vector Definition Files*. For details, see *2.2 Reset Function (resetprg.c)*.
`sbrk.c`	Memory management-related low-level interface routine • Defines the low-level interface routine for memory management (`sbrk`). • Generated according to the specification in *(4) Use Heap Memory*. For details, see *5.1 Memory Management (sbrk.c, sbrk.h)*.
`test.c` (`test.cpp`)	Main routine • Defines the main function, as well as the `abort` function when C++ is used. • The file name specified in *(1) Project Name* is used.
`env.inc`	Address definition for exception processing registers • Defines the addresses in which the exception event register (`EXPEVT`) and interrupt event register (`INTEVT`) are placed. For details, see *2.1 Reset Handlers (vhandler.src, vecttbl.src, env.src, env.inc)*.

Table 1-2 List of auto-generated sample files (2)

`lowsrc.h`	I/O low-level function header • Defines the `IOSTREAM` macro, which specifies the file handler count (number of files that can be used at the same time. • Generated according to the specification in *(4) Use I/O library*. • The value set in *(4) Number of I/O Streams* reflected. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h).*
`sbrk.h`	Low-level header for memory management • Defines the `HEAPSIZE` macro, which specifies the total size of the heap area. • Generated according to the specification in *(4) Use Heap Memory*. • The value set in *(4) Heap Size* is reflected. For details, see *5.1 Memory Management (sbrk.c, sbrk.h)*.
`stacksct.h`	Stack section size header • Defines the size of the stack section. • Generated according to the specification in *(7) Vector Definition Files*. • The value set in *(6) Stack Size* is reflected. For details, see *2.3 Stack Size Settings (stacksct.h)*.
`typedefine.h`	Type alias declaration header • Declares type aliases.
`vect.inc`	Header for vector tables • Declares the prototype for the reset function and interrupt function. • Generated according to the specification in *(7) Vector Definition Files*.

2. Reset Processing

The following explains operation sample programs generated by HEW after power-on reset.

2.1 Reset Handlers (vhandler.src, vecttbl.src, env.src, env.inc)

The CPU performs the following operations when an exception occurs due to one of the five reset causes shown in Table 2-1.

1. It sets the program counter (PC) to H'A0000000.
2. It sets the exception code in the exception event register (EXPEVT).
3. It sets the mode bit (MD), register bank bit (RB), and exception / interrupt block bit (BL) of the status register (SR) to 1, sets the FPU disable bit (FD) to 0, and the interrupt mask bits (I3 to I0) to B'1111.
4. It sets the vector base register (VBR) to H'A0000000.

Table 2-1 Exception list (reset cause)

Exception	Exception code	Vector base	Offset from _RESET_Vectors	Exception processing routine
Power-on reset	H'000	H'A0000000	H'000	_PowerON_Reset
Manual reset	H'020	H'A0000000	H'004	_Manual_Reset
H-UDI	H'000	H'A0000000	H'000	_PowerON_Reset
Instruction TLB	H'140	H'A0000000	H'028	_TBL_Reset
Data TLB	H'140	H'A0000000	H'028	_TBL_Reset

The cause of an exception is determined based on the value of EXPEVT. The sample program references the exception code set for EXPEVT in _ResetHandler defined in vhandler.src, and jumps to the processing function for each exception cause.

This processing to determine exceptions is called a *reset handler*, and the processing function for each exception cause is called an *exception processing routine*.

Details about _ResetHandler

(1) The value of EXPEVT is loaded. (a), (b)
(2) The value of EXPEVT is used to calculate the offset from _RESET_Vectors (EXPEVT / 8). (c), (d)
(3) The value from (2) is added to the address for _RESET_Vectors. (e), (f)
(4) The address of the exception processing routine is obtained from the address in (3). (g)
(5) Jump is performed to the exception processing routine obtained in (4). (h)

```
                .include        "env.inc"
                .include        "vect.inc"

                .import         _RESET_Vectors
                .import         _INT_Vectors
                .import         _INT_MASK
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       reset                                                               ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .section    RSTHandler,code
_ResetHandler:
                mov.l   #EXPEVT,r0              (a)
                mov.l   @r0,r0                  (b)
                shlr2   r0                      (c)
                shlr    r0                      (d)
                mov.l   #_RESET_Vectors,r1      (e)
                add     r1,r0                   (f)
                mov.l   @r0,r0                  (g)
                jmp     @r0                     (h)
                nop
```

List 2-1

Note 1:

The address for the exception event register (EXPEVT) is set for the EXPEVT symbol in `env.inc` (List 2-2).

```
EXPEVT:         .equ    H'FF000024
```

List 2-2

Note 2:

`_RESET_Vectors` is defined in `vecttbl.src` (List 2-3).

For example, when power-on reset occurs, the exception code `H'000` is set for EXPEVT. Then, since the exception code `H'000` is used to calculate the offset value `0`, jump is performed to the `_PowerON_Reset` function, at the beginning of `_RESET_Vectors`.

```
        .include        "vect.inc"

        .section        VECTTBL,data
        .export         _RESET_Vectors

_RESET_Vectors:
;<<VECTOR DATA START (POWER ON RESET)>>
        ;H'000 Power On Reset (Hitachi-UDI RESET)
        .data.l  _PowerON_Reset
;<<VECTOR DATA END (POWER ON RESET)>>
;<<VECTOR DATA START (MANUAL RESET)>>
        ;H'020 Manual Reset
        .data.l  _Manual_Reset
;<<VECTOR DATA END (MANUAL RESET)>>
; Reserved
        .datab.l        8,H'00000000
;<<VECTOR DATA START (TBL RESET)>>
        ;H'140 TBL Reset (DATA TBL Reset)
        .data.l         _TBL_Reset
;<<VECTOR DATA END (TBL RESET)>>
```

List 2-3

Note 3

Since the position for PC after the reset cause exception occurs is 0xA0000000, the reset handler needs to be placed in the 0xA0000000 position. Since in the sample program the reset handler (_ResetHandler) is placed in the RSTHandler section, the RSTHandler section is placed in the 0xA0000000 position, in the linker section (Figure 2-1).

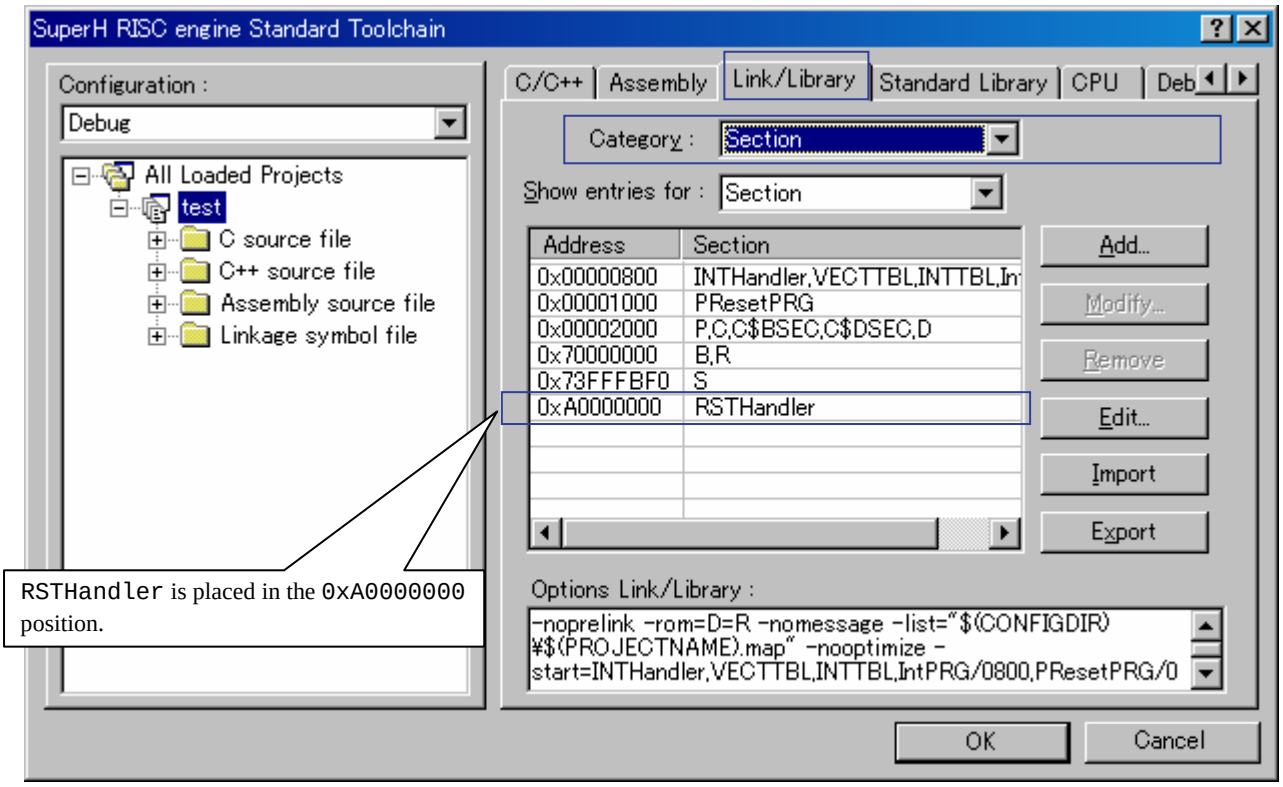

Figure 2-1

2.2 Reset Function (resetprg.c)

The following shows the processing contents for the `PowerON_Reset` reset function, when power-on reset is performed.

C source	Description
`#include <machine.h>`	When an embedded function such as `set_cr`, `set_vbr`, or `sleep` is used, `include` is performed for `machine.h`.
`#include <_h_c_lib.h>`	When the `_INITSCT` function is used, `include` is performed for `_h_c_lib.h`.
`//#include <stddef.h>`	When `errno` is used, `include` is performed for `stddef.h`.
`//#include <stdlib.h>`	When the `rand` function is used, `include` is performed for `stdlib.h`.
`#include "typedefine.h"`	Type alias declaration is performed in `typedefine.h`.
`#include "stacksct.h"`	`#pragma stacksize` is specified.
`#define SR_Init 0x000000F0`	The value set for the status register (SR) is defined as a macro. The 4th to 7th bits of the SR are the interrupt mask bits (I3 to I0), and `H'F` (`B'1111`) is set as interrupt mask level 15 (no interrupt).
`#define INT_OFFSET 0x100UL`	The size of the reset vector table is defined as a macro. This is used as an offset value during processing to set the vector base register (VBR).
`extern void INTHandlerPRG(void);`	A `INTHandlerPRG` prototype declaration is performed.

`#ifdef __cplusplus`
`extern "C" {`
`#endif`
	When C++ is used, an `extern "C"` declaration is performed.

`Void PowerON_Reset(void);`
`Void Manual_Reset(void);`
`Void main(void);`
`#ifdef __cplusplus`
`}`
`#endif`
	A `PowerON_Reset` prototype declaration is performed.
	A `Manual_Reset` prototype declaration is performed.
	A `main` prototype declaration is performed.

`#ifdef __cplusplus`
`extern "C" {`
`#endif`
`extern void INIT IOLIB(void);`
	A prototype declaration is performed for I/O-related standard library initialization processing.

`extern void CLOSEALL(void);`
`#ifdef __cplusplus`
`}`
`#endif`
	A prototype declaration is performed for the I/O-related standard library end function.

`//extern void srand(_UINT);`	When the `rand` function is used, an `srand` prototype declaration is performed.
`//extern _SBYTE *_s1ptr;`	When the `strtok` function is used, a declaration for the `_s1ptr` variable is enabled.

`//#ifdef __cplusplus`
`//extern "C" {`
`//#endif`
`//extern void HardwareSetup(void);`
`//#ifdef __cplusplus`
`//}`
`//#endif`
	When `HardwareSetup` is called, a prototype declaration is performed.

C source	Description

`//#ifdef __cplusplus`
`//extern "C" {`
`//#endif`
`//extern void _CALL_INIT(void);`
	A prototype declaration for constructor call processing. This is enabled when global classes are used.

`//extern void _CALL_END(void);`	A prototype declaration for destructor call processing. This is enabled when global classes are used.

`//#ifdef __cplusplus`
`//}`
`//#endif`	

`#pragma section ResetPRG`	The reset function is placed in the `PResetPRG` section.
`#pragma entry PowerON_Reset`	Specifies the entry function for the `PowerON_Reset` function. When this is specified in the function, save/restore code for the register can be suppressed. Note that since a `#pragma stacksize` specification exists, code that sets the stack address at `R15` at the beginning of the `PowerON_Reset` function is generated.
`void PowerON_Reset(void)` `{`	
`set_vbr((void *)((_UINT *)& INTHandlerPRG - INT_OFFSET));`	Setting processing is performed for the vector base register (VBR). For details, see *3.3. Setting Vector Base Registers (VBR) (set_vbr function)*.
`INITSCT();`	A function to process memory is called. For details, see *4. Memory Initialization*.
`//  CALL_INIT();`	Constructor call processing is performed for global class objects. For details, see *6. Precautions Regarding C++ Usage (_CALL_INIT Function and CALL_END Function)*.
`_INIT_IOLIB();`	The I/O-related standard library is initialized. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)*.
`//  errno=0;`	This is for `errno` initialization processing. This is enabled when `errno` is used.
`//  srand((_UINT)1);`	When the `rand` function is used, `srand` needs to be called to initialize the random number table.
`//  _s1ptr=NULL;`	When the `strtok` function is used, the `_s1ptr` variable needs to be initialized.
`//  HardwareSetup();`	A dummy function for hardware setting processing is called.
`set_cr(SR_Init);`	Setting processing is performed for the status register (SR).
`main();`	The `main` function is called.
`_CLOSEALL();`	End processing is performed for the I/O-related standard library.
`//  _CALL_END();`	Destructor call processing is performed. This needs to be called when global classes are used.
`sleep();` `}`	The `sleep` instruction is executed and the status changes to `sleep` so that `PowerON_Reset` cannot be avoided.

`//#pragma entry Manual_Reset`
`void Manual_Reset(void)`
`{`
`}`
	This is the manual reset function (dummy).

2.3 Stack Size Settings (stacksct.h)

The address of the stack pointer needs to be set in R15 for the user program. In the sample program, `#pragma entry` and `#pragma stacksize` extension functions are used to perform setting processing at the beginning of the `PowerON_Reset` function (List 2-4).

C source code

```
void PowerON_Reset(void)
{
    set_vbr((void *)((_UINT *)&
INTHandlerPRG - INT_OFFSET));
                ....
```

Assembly code

```
_PowerON_Reset:
        MOV.L     L12+2,R15  ; STARTOF S+SIZEOF S
        MOV.L     L12+6,R6   ; _INTHandlerPRG
        MOV       #1,R1      ; H'00000001
                          ....
```

The stack address is set in R15.

List 2-4

The compiler reserves a 0x400-byte stack area (S section), due to the `#pragma stacksize` specification in `stacksct.h` (List 2-5).

Since the stack is used from higher addresses to lower address, the start address of the S section needs to be set to (*stack-pointer-address* - *stack-size*). In the sample project, since the stack pointer address is set to `0x73FFFFF0` (Figure 1-6), the S section start address is set to `0x73FFFBF0` (0x73FFFFF0 – 0x400) for the section placement in the optimization linkage editor (Figure 2-2).

```
#pragma stacksize 0x400
```

List 2-5

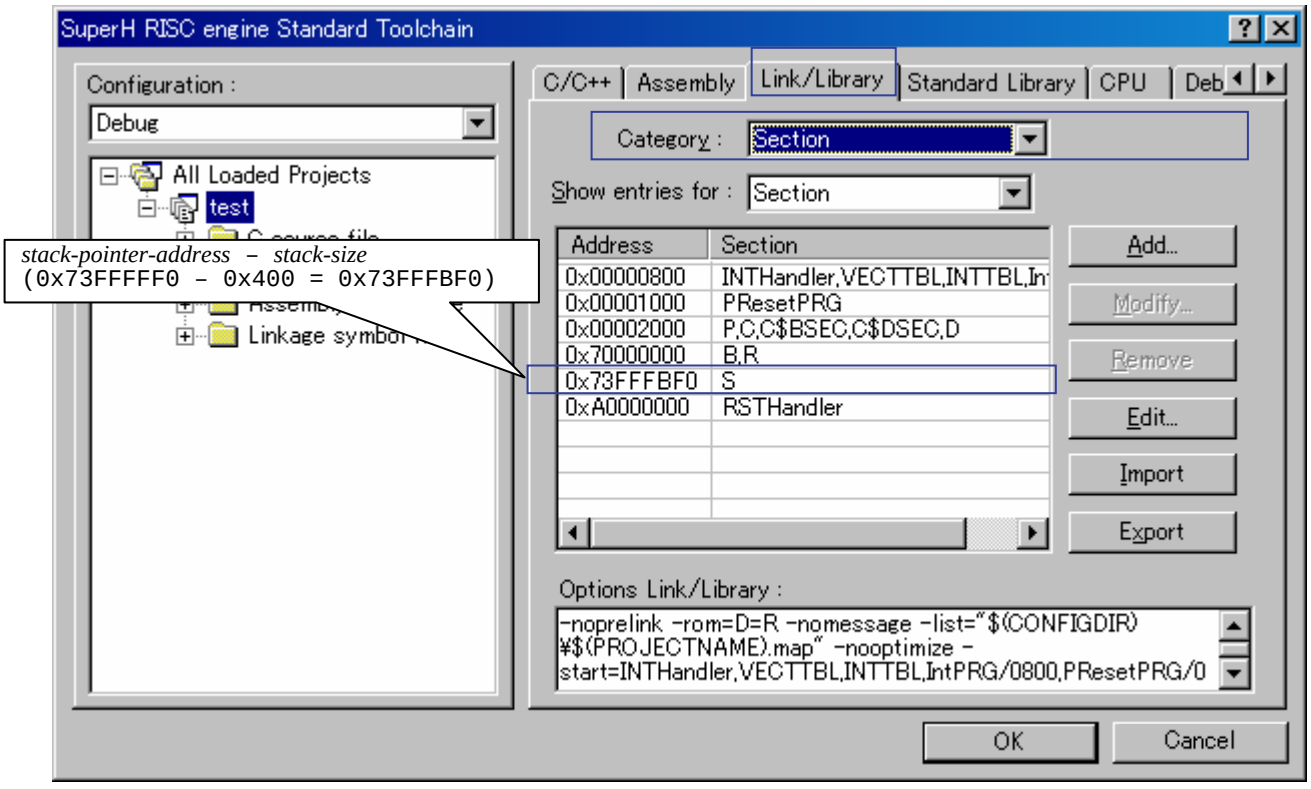

Figure 2-2

3. Non-reset Exceptions

Non-reset exceptions include general exceptions and exceptions due to interrupts. The hardware performs the following when a non-reset exception occurs:

1. It saves the status register (SR) and program counter (PC) to the save status register (SSR) and save program counter (SPC), respectively.
2. It sets the BL bit, MD bit, and RB bit for the SR to 1, and sets PC to (VBR + *offset-value-for-each-exception-cause*). This means that the bank is switched, the processing mode switches to privileged mode, and interrupts are masked.
   - BL bit
     When the BL bit is 0, exceptions and interrupts are accepted. When the BL bit is set to 1, all interrupts are masked. When an exception other than a user break occurs, the status of the internal CPU register and other module registers is reverted to that after manual reset.
   - MD bit
     When the MD bit is 0, the user mode takes effect. When the MD bit is 1, privileged mode takes effect.
   - RB bit
     When the RB bit is 0, R0_BANK0 to R7_BANK0 are accessed as general registers R0 to R7.
     When the RB bit is 1, R0_BANK1 to R7_BANK1 are accessed as general registers R0 to R7.

3.1 Processing Handlers for Non-reset Exceptions (vhandler.src, vecttbl.src, env.src)

When a non-reset exception occurs, PC is set to (VBR + *offset-value-for-each-exception-cause*). When a general exception occurs, the exception event register (EXPEVT) is set to the exception cause. When an interrupt occurs, the interrupt event register (INTEVT) is set to the interrupt cause.

Table 3-1 lists some of the offset values and exception codes (EXPEVT/INTEVT) for exception causes.

Table 3-1 Exceptions list (partial)

Exception cause	Offset	Exception code	Exception processing routine	Index
Data TLB miss exception (read)	H'400	H'040	INT_TLBMiss_Load	0
Data TLB miss exception (write)	H'400	H'060	INT_TLBMiss_Store	1
Initial page write exception	H'100	H'080	INT_TLBInitial_Page	2
Data TLB protection violation exception (read)	H'400	H'0A0	INT_TLBProtect_Load	3
⋮				
Unconditional trap	H'100	H'160	INT_TRAPA	9
⋮				
Non-maskable interrupt	H'600	H'1C0	INT_NMI	12
User break after instruction execution	H'100	H'1E0	INT_User_Break	13
External interrupt 0	H'600	H'200	INT_Extern_0000	14
External interrupt 1	H'600	H'220	INT_Extern_0001	15

In the sample project, the following exception processing handlers defined in vhandler.src are called:

- When a general exception occurs: _INTHandlerPRG
- When a TLB miss exception occurs: _TLBmissHandler
- When an interrupt occurs: _IRQHandler

The exception processing handler calls the processing function for each exception cause (exception processing routine), control returns to the exception processing handler again once processing is complete for the exception processing routine, and then control is returned to normal processing from the exception processing handler (Figure 3-1).

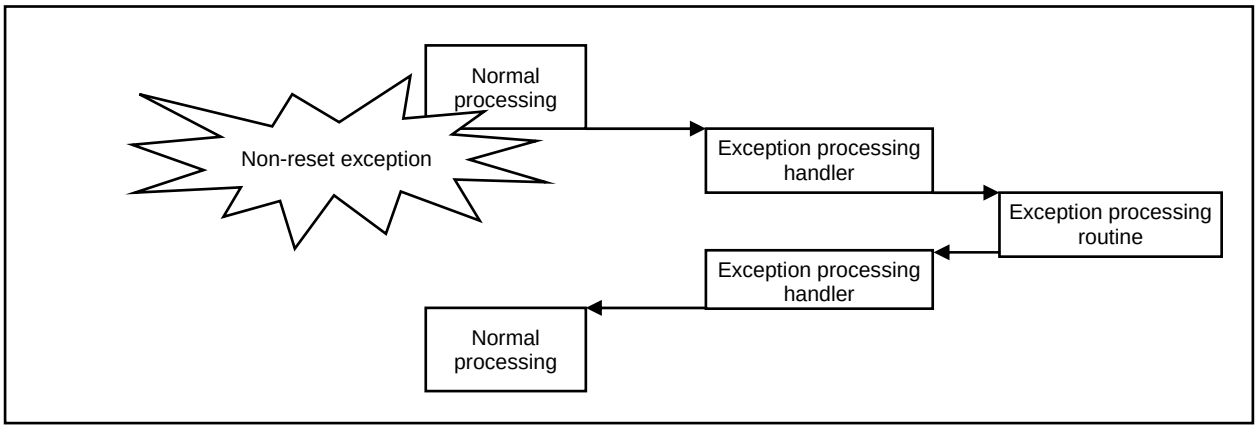

Figure 3-1

3.2 General Exception Processing Handler (_INTHandlerPRG)

The following uses the general exception processing handler `_INTHandlerPRG` (List 3-1) as an example to explain exception processing handlers.

The same processing is also performed for the TLB miss exception processing handler (`_TLBmissHandler`) and interrupt handler (`_IRQHandler`).

Note that `INTEVT` is referenced when the address of the exception processing routine is calculated in the interrupt handler.

Note:
When the exception processing handler is called, `RB=1` (bank 1) and `BL=1` (interrupt mask) are set for `SR`.

(1) Saving the register (a)
The `PUSH_EXP_BASE_REG` macro defined in `vhandler.src` (List 3-2) is called, and the general register and `SSR`, `SPC`, `PR`, and `FPSCR` are saved.

(2) Obtaining the exception processing routine address (b)
The value of `EXPEVT` is obtained to get the offset from `_INT_Vectors` ((EXPEVT – 0x40) / 8), and the address of the exception processing routine is calculated.

(3) Obtaining the interrupt mask (c)
The value of `EXPEVT` is used to calculate the offset value (EXPEVT – 0x40) / 16) from `INT_MASK` (defined in `vecttbl.src`), and the interrupt mask value corresponding to the interrupt cause is obtained.

(4) Setting the save status register (`SSR`) (d)
The `RB` and `BL` bits are cleared to `0` for the current status register (`SR`) value, and the value set for the interrupt mask obtained in (3) is set for `SSR`.

(5) Setting the save program counter (`SPC`) and status register (`SR`) (e)
The address of the exception processing routine obtained in (2) is set for `SPC`, and the address of the `__int_term` function is set for `PR`.

(6) Executing RTE instructions (f)
An RTE instruction is executed. The RTE instruction restores `SPC` to `PC` and `SSR` to `SR`, and branches to the `SPC` address.

Since the address of the exception processing routine is stored in SPC by the processing from (5), it is moved to the exception processing routine.
The status value obtained from the processing in (4) is stored in SSR. As such, when the transition to exception processing routine is performed, it is switched to bank 0, and interrupts greater than the mask value can be accepted. If an interrupt is accepted during execution of an exception processing routine, multiple interrupts occur. Unless the BL bit is cleared to 0 in the processing in (4), multiple interrupts are prohibited.

(7) Returning from the exception processing routine (g)
In the sample program, the exception processing routine is coded as a normal function (do not specify `#pragma interrupt`). As such, an RTS instruction is used to perform return from the exception processing routine. Since the address of `__int_term` is stored in PR through the processing in (5), it is moved to `__int_term`. In `__int_term`, the POP_EXP_BASE_REG macro defined in `vhandler.src` in List 3-2 is called, and the general register and SSR, SPC, PR, and FPSCR are restored. Finally, an RTE instruction is used to performed return from the exception processing handler.

Figure 3-2 shows the flow of exception processing statuses in the sample program.

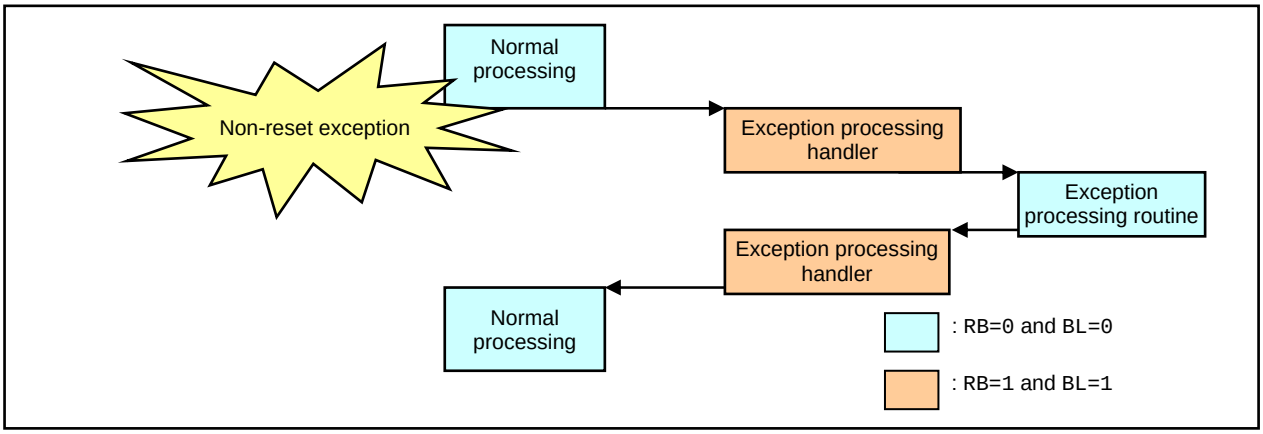

Figure 3-2

The assembler source for the general exception processing handler (`INTHandlerPRG`) defined in `vhandler.src` is as follows.

```
IMASKclr:       .equ    H'FFFFFF0F
RBBLclr:        .equ    H'CFFFFFFF
MDRBBLset:      .equ    H'70000000

                .import         _RESET_Vectors
                .import         _INT_Vectors
                .import         _INT_MASK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        exceptional interrupt                                          ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .section    INTHandler,code
        .export      _INTHandlerPRG
_INTHandlerPRG:
_ExpHandler:
                PUSH_EXP_BASE_REG               (a)
;
                mov.l    #EXPEVT,r0             (b)
                mov.l    @r0,r1
                mov.l    #_INT_Vectors,r0
                add      #-(h'40),r1
                shlr2    r1
                shlr     r1
                mov.l    @(r0,r1),r3
;
                mov.l    #_INT_MASK,r0          (c)
                shlr2    r1
                mov.b    @(r0,r1),r1
                extu.b   r1,r1
;
                stc      sr,r0                  (d)
                mov.l    #(RBBLclr&IMASKclr),r2
                and      r2,r0
                or       r1,r0
                ldc      r0,ssr
;
                ldc.l    r3,spc                 (e)
                mov.l    #__int_term,r0
                lds      r0,pr
;
                rte                             (f)
                nop
;
                .pool
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        Interrupt terminate                                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .align   4
__int_term:                                     (g)
                mov.l    #MDRBBLset,r0
                ldc.l    r0,sr
                POP_EXP_BASE_REG
                rte
                nop
;
                .pool
;
```

List 3-1

Note 1

The PUSH_EXP_BASE_REG and POP_EXP_BASE_REG macros, which perform register save and restore when called by the exception processing reset handler, are defined in vhandler.src (List 3-2).

Perform save/restore of floating-point number registers as necessary. Note that when the exception processing routine is coded in C and the macsave=0 compiler option is specified, both the MACH register and MACL register need to be saved/restored.

In the macro shown in List 3-2, R0_BANK to R7_BANK are saved to the stack (stc.l rn_bank, @-r15), and restored from the stack (ldc.l @r15+, rn_bank), and save/restore is not performed for the general register (R0 to R7). This is because when an interrupt is accepted, the RB bit of the SR register is automatically set to 1, so that the general register before the exception occurs is the bank register.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;*              macro definition                                        *;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .macro PUSH_EXP_BASE_REG
        stc.l   ssr,@-r15               ; save ssr
        stc.l   spc,@-r15               ; save spc
        sts.l   pr,@-r15                ; save context registers
        sts.l   fpscr,@-r15             ; save fpscr registers
        stc.l   r7_bank,@-r15
        stc.l   r6_bank,@-r15
        stc.l   r5_bank,@-r15
        stc.l   r4_bank,@-r15
        stc.l   r3_bank,@-r15
        stc.l   r2_bank,@-r15
        stc.l   r1_bank,@-r15
        stc.l   r0_bank,@-r15
                .endm
;
                .macro POP_EXP_BASE_REG
        ldc.l   @r15+,r0_bank           ; recover registers
        ldc.l   @r15+,r1_bank
        ldc.l   @r15+,r2_bank
        ldc.l   @r15+,r3_bank
        ldc.l   @r15+,r4_bank
        ldc.l   @r15+,r5_bank
        ldc.l   @r15+,r6_bank
        ldc.l   @r15+,r7_bank
        lds.l   @r15+,fpscr
        lds.l   @r15+,pr
        ldc.l   @r15+,spc
        ldc.l   @r15+,ssr
                .endm
```

List 3-2

Note 2

In SH7760, some exceptions and interrupt use the same exception code (Table **). As such, the sample program cannot perform processing to differentiate these exceptions and interrupts. Modify the sample program so that the exception processing handler for general exceptions (_INTHandlerPRG) and that for interrupts (_IRQHandler) refer to different function tables.

Table 3-2 SH7760 exception codes (exception/interrupt)

Exception cause	Exception code	Exception processing routine
General FPU compression exception	H'800	_INT_Illegal_FPU
Slot FPU compression exception	H'820	_INT_Illegal_slot_FPU
IRQ4	H'800	_INT_Illegal_FPU
IRQ5	H'820	_INT_Illegal_slot_FPU

3.3 Setting Vector Base Registers (VBR) (set_vbr function)

By setting an arbitrary address in a VBR, the non-reset exception processing handler can be placed in any address. A VBR can be set by using the embedded `set_vbr` function. In the sample program, the value set for the VBR is calculated from the placement address for `INTHandlerPRG` (List 3-3). Since `INTHandlerPRG` is placed at the beginning of the `INTHandler` section, INTHandler can be placed at an arbitrary address to place the non-reset exception processing handler at any address.

```
resetprg.c
    #define  INT_OFFSET  0x100UL
                  . . .
    set_vbr((void *)((_UINT *)&INTHandlerPRG - INT_OFFSET));
```

List 3-3

When a non-reset exception occurs, jump is performed to (VBR + *offset-value-for-each-exception-cause*). As such, each exception processing handler must be placed in the location shown in Table 3-3. In the sample program, the offset value based on `_INTHandlerPRG` is used to place `_TLBmissHandler` and `_IRQHandler` (List 3-4).

Table 3-3 Offset values from NTHandlerPRG

Exception type	Exception processing handler	Offset value from the VBR	Offset value from _INTHandlerPRG
General exception	`_INTHandlerPRG`	`H'100`	`H'000`
TLB miss exception	`_TLBmissHandler`	`H'400`	`H'300`
Interrupt	`_IRQHandler`	`H'600`	`H'500`

```
        .section    INTHandler,code
        .export     _INTHandlerPRG
_INTHandlerPRG:
      .
      .
      .
        .org    H'300
_TLBmissHandler:
      .
      .
      .
        .org    H'500
_IRQHandler:
      .
      .
      .
```

List 3-4

3.4 Exception Processing Routine (intprg.src)

For non-reset exceptions, the dummy functions for exception processing routines (such as the `_INT_TLBMiss_Load` function and `_INT_TLBMiss_Store` function) are defined in `intprg.src` (List 3-5).

```
;H'040 TLB miss/invalid (load)
_INT_TLBMiss_Load
;H'060 TLB miss/invalid (store)
_INT_TLBMiss_Store
;H'080 Initial page write
_INT_TLBInitial_Page
      .
      .
      .
;H'820 Illegal slot FPU
_INT_Illegal_slot_FPU
    sleep
    nop
    .end
```

List 3-5

When using C to code an exception processing routine, comment out the dummy function, and create a C function with the same name as the dummy function, but with the initial underscore removed. Note that `#pragma interrupt` does not need to be specified here.

Example:

```
void INT_TLBMiss_Load(void)
{

}
```

List 3-6

4. Memory Initialization

In the sample program, call memory initialization is performed for the _INITSCT function in the standard library.

The _INITSCT function performs the following initialization processing.

- Initialization for initialized data areas
- Initialization for uninitialized data areas

4.1 Memory Initialization Function _INTSCT (dbsct.c)

When using the _INITSCT function, include <_h_c_lib.h> to link the standard library.

The _INITSCT function obtains the initialization target of the initialized data area from the C$DSEC section, and the initialization target of the uninitialized data area from the C$BSEC section. In the sample program, the initialization processing target for the initialized data area is defined in the dbsct.c (Figure 4-1) structure array DTBL, and the initialization processing target for the uninitialized data area is defined in the structure array BTBL.

Line | Source

```
16  #pragma section $DSEC
17  static const struct {
18      _UBYTE *rom_s;        /* Start address of the initialized data section in ROM */
19      _UBYTE *rom_e;        /* End address of the initialized data section in ROM   */
20      _UBYTE *ram_s;        /* Start address of the initialized data section in RAM */
21  }   DTBL[] = {
22      { __sectop("D"), __secend("D"), __sectop("R") }
23  };
24  #pragma section $BSEC
25  static const struct {
26      _UBYTE *b_s;          /* Start address of non-initialized data section */
27      _UBYTE *b_e;          /* End address of non-initialized data section */
28  }   BTBL[] = {
29      { __sectop("B"), __secend("B") }
30  };
```

Figure 4-1

Initialization of initialized data areas

Initialized data is data (variables) with an initial value. The initial value needs to be held in a ROM area, but since the data can be rewritten while the program is executing, it needs to be placed in a RAM area. During initialization processing for the initialized data area of __INITSCT function, processing is performed to copy the initial value data in the ROM area to a RAM area. Also, to place the initial value in the ROM area and use the RAM area address to access data, the ROM support option needs to be specified in the linker. (For details, see *4.4 ROM*.)

In the sample project, data is specified to be copied from the D section to the R section in the DTBL structure array for dbsct.c, and the ROM support option is specified in the linker. (Figure 4-2)

Initialization of uninitialized data areas

In C/C++, static variables without initial values and external variables without initial values need to be 0.The specified sections are cleared to 0 during initialization processing for uninitialized data areas in the __INITSCT function.

In the sample program, the B section is specified to be cleared to 0 in the BTBL structure array for dbsct.c.

4.2 If Initialized Data Areas Other Than the D Section Exist

If initialized data areas exist outside of the D section, add them to the DTBL structure array.

For example, to copy the D1 section to the R1 section, add it as shown in List 4-1. Make sure that you also specify the ROM support option.

```
#pragma section $DSEC
static const struct {
       _UBYTE  *rom_s
       _UBYTE  *rom_e
       _UBYTE  *ram_s
}   DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D1"),__secend("D1"),__sectop("R1")}
};
```

List 4-1

4.3 If Unitialized Data Areas Other Than the B Section Exist

If uninitialized data areas exist outside of the B section, add them to the BTBL structure array.

For example, to clear the B1 section to 0, add it as shown in List 4-2.

```
#pragma section $DSEC
static const struct {
       _UBYTE  *b_s;    /*  First address for uninitialized data section  */
       _UBYTE  *b_e;    /*  Last address for uninitialized data section  */
}   BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B1"),__secend("B1")}
};
```

List 4-2

4.4 ROM Support Functionality

The following processing is performed when the ROM support functionality for the linkage editor is used.

- An area of the same size as the ROM initialized data area is reserved in RAM.
- Addresses are resolved automatically by having references for symbols declared in initialized data areas refer to RAM area addresses.

Perform the following to display the dialog box and perform settings.

Toolchain dialog box

-> Select the **Link/Library** tab, and then in Category, select **Output**.

-> In **Show entries for**, select **ROM to RAM mapped sections**.

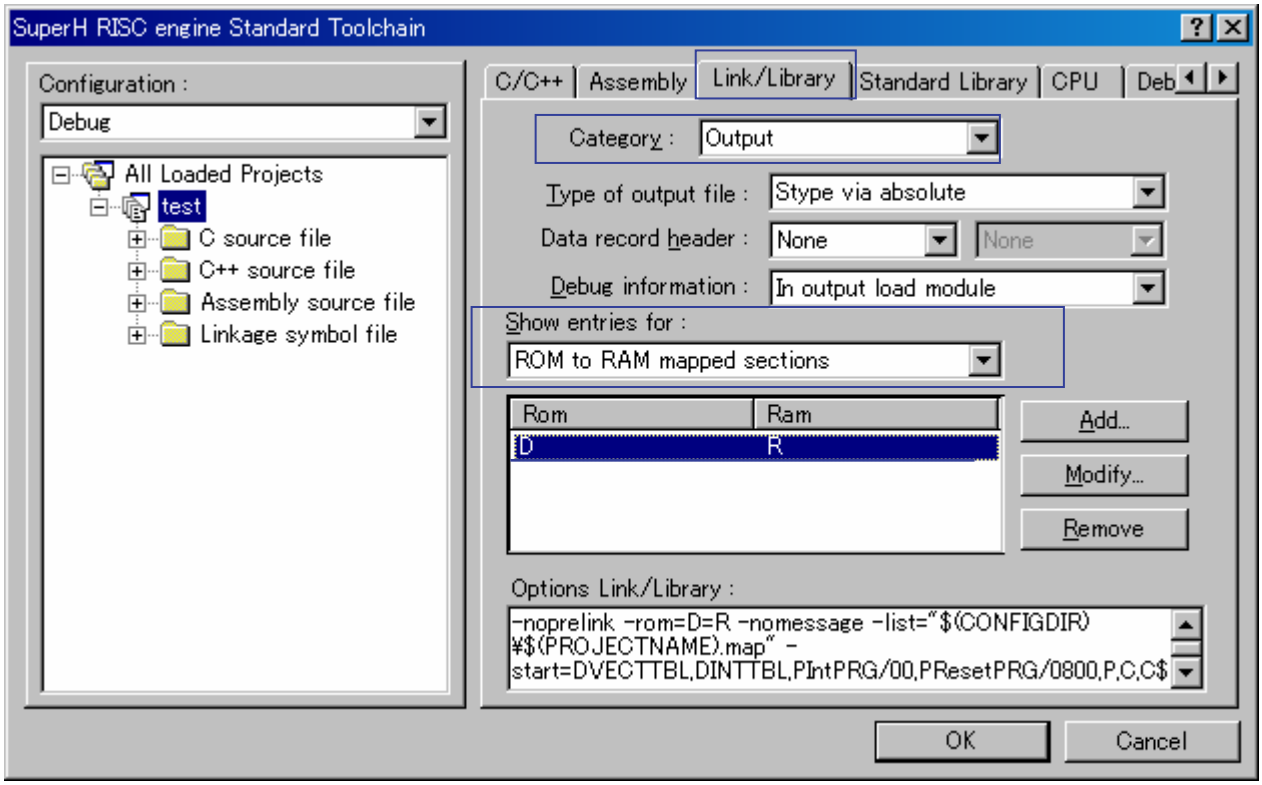

Figure 4-2

In the sample project, the D section is specified in ROM, and the R section is specified in RAM. This specification means that an R section the same size as the D section is reserved in RAM during linkage, and that addresses are resolved by having references for symbols declared in initialized data areas refer to R section RAM area addresses.

5. Low-level Interface Routine Settings

When development is performed in C/C++, functions such as those in the standard I/O library (including `fopen`, `printf`, and `scanf`) and the memory management library (including `malloc`, `free`, `new`, and `delete`) may be used. Unfortunately, not all of these functions are provided by the compiler. For example, standard output may refer to output to an LCD, hard disk, printer, or CD-R/RW drive, and standard input may refer to input from a DIP switch, keyboard, mouse, mobile phone button, or touch panel. In addition, the operations for each of these devices may differ. As such, the compiler cannot provide all processing for the standard I/O and memory management library. This is why there is a group of functions from the standard I/O and memory management library, which are called low-level interface routines. A low-level interface routine needs to be implemented by the user. Low-level interface routines include `open`, `close`, `read`, `write`, `lseek`, `sbrk`, `errno_addr`, `wait_sem`, and `signal_sem`.

For details about the specifications for each routine, see *(6) Low-level interface routines in 9.2.2 Execution environment settings in the Compiler Users Manual*.

5.1 Memory Management (sbrk.c, sbrk.h)

Table 5-1 is a sample list of low-level interface routines for memory management, as generated by HEW.

Table 5-1 Sample list of low-level interfaces (for memory management)

Source file name	Low-level interface	Function
`sbrk.c`	`sbrk()`	A function for reserving heap memory. Memory of the size specified by the argument is reserved. If this is called multiple times, memory is reserved sequentially from lower addresses. Memory is obtained until the size defined by `HEAPSIZE`.
`sbrk.h`	`HEAPSIZE`	Defines the `HEAPSIZE` macro for specifying the overall size of the heap area.

Note:

Memory management library functions call the `sbrk` function to reserve memory. The reserved memory is managed within the library function, and areas freed by the `free` or `delete` function are reused as heap memory. The size requested for memory reservation by the `sbrk` function is that specified by `_sbrk_size` (default: 1024). If reserved memory becomes insufficient, the `sbrk` function is called again. When heap memory is reserved and released repeatedly, even though the total free area size remains sufficient, since the free area is divided among several small areas, situations may occur in which large area requests may not be able to be reserved. As such, we recommend setting `_sbrk_size = HEAPSIZE`, so that the heap memory area for one `sbrk` function call is obtained in batch. When this method is used, heap memory fragmentation is reduced, and heap area management processing is more efficient.

Example:

```
SBYTE  *sbrk(size_t size);
const size_t _sbrk_size = HEAPSIZE;     /* Specifies the minimum unit of      */
/*  Clears comments and sets the HEAPSIZE to the initial value.   */
```

5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)

Table 5-2 is a sample list of low-level interface routines for I/O, as generated by HEW.

Table 5-2 Sample list of low-level interfaces (for I/O)

Source file	Low-level interface	Functionality
lowsrc.c	_INIT_IOLIB()	A function that performs file handler initialization, and opens files for standard input (stdin), standard output (stdout), and standard error output (stderr). When standard input, standard output, and standard error output are not used, delete the corresponding open processing. Do not perform file handler operations anywhere other than in the _INIT_IOLIB function. Use the setbuf or setvbuf function to set the _bufptr, _bufcnt, _bufbase, and _buflen file handler member variables after file open is performed.
lowsrc.c	_CLOSEALL()	A function that closes all unclosed files.
lowsrc.c	open()	Performs whether a file open request is for standard input, standard output, or standard error output, and checks the file mode. In the sample program, no actual processing to open files is performed.
lowsrc.c	close()	Checks the file number range and clears the file mode. If a range error occurs for a file number, -1 is returned as the error.
lowsrc.c	read()	A function that calls the charget function, which actually obtains characters, once for each character that exists, once the file mode is checked. If an error occurs, -1 is returned.
lowsrc.c	write()	A function that calls the charput function, which actually outputs characters, once for each character that exists, once the file mode is checked. If an error occurs, -1 is returned.
lowsrc.c	lseek()	A dummy function. No processing is performed in the lseek function generated by HEW.
lowsrc.h	IOSTREAM	A macro definition that specifies the file handler count (the number of files that can be used concurrently). Use the IOSTREAM macro to change the file handler count. Note that in the lowsrc.c generated by HEW, the three file handlers for standard input (stdin), standard output (stdout), and standard error output (stderr) are opened in the _INIT_IOLIB function. As such, when such open processing is enabled, the number of file handlers available to the user is (IOSTREAM - 3).
lowlvl.src	charget()	A character input function called from the read() function. This receives character input from the I/O simulation window of the simulator debugger. Note that the algorithm for this function only runs on the simulator debugger, and not on the actual target.
lowlvl.src	charput()	A character output function called from the write() function. This outputs characters to the I/O simulation window of the simulator debugger. Note that the algorithm for this function only runs on the simulator debugger, and not on the actual target.

6. Precautions Regarding C++ Usage (_CALL_INIT Function and CALL_END Function)

When C++ is used, and either globally declared variables are dynamically initialized or globally declared class objects (global class objects) exist, the `_CALL_INIT` function needs to be called ahead of time. In the following source program, (a) and (b) are global class objects.

```
class A
{
...
};

A g_A;        ...(a)
A * g_pA;
static A s_A; ...(b)

void main()
{
    A a;
    A * p_a;
    static A s_a;
    g_pA = new A; delete g_pA;
    l_pA = new A; delete l_pA;
}
```

List 6-1

If this class has a constructor, the constructor needs to be called before the class member is accessed. For example, in the following C++ program, (c) is processed before (e) is executed, and the (a) member variable for (d) needs to be initialized to 1. In other words, the (c) constructor needs to be called.

```
class A
{
private:
    int a;
public:
    A(void) { a = 1; }          ...(c)
    int Get(void) { return a; }
};

A g_a;                          ...(d)

void main()
{
    int a = g_a.Get();          ...(e)
}
```

List 6-2

The _CALL_INIT function is provided as a standard library to use this constructor call. Likewise, the _CALL_END function is also provided to call the global class object destructor. Since the _CALL_INIT function and _CALL_END function are declared in <_h_c_lib.h>, include is performed for <_h_c_lib.h> in the source file used (f). Call the _CALL_INIT function before application start (g), and call the _CALL_END function once the application has been terminated (h).

```
#include <_h_c_lib.h>     ...(f)

void PowerON_Reset_PC(void)
{
    _INITSCT();
    _CALL_INIT();         ...(g)

    main();

    _CALL_END();          ...(h)
    sleep();
}
```

List 6-3

Note that information to call the constructor and destructor is generated in the C$INIT section, which is automatically generated by the compiler. Use the memory placement setting for the optimization linkage editor to place the C$INIT section in the ROM area.

7. Using C to Code Exception Processing Programs

In the HEW sample program, the exception processing handler is coded using assembly language. The #pragma extension function can be used to code, in C, exception processing handlers and exception processing routines that use register banks.

7.1 Without Multiple Interrupts

The following shows how to code exception processing handlers and exception processing routines in C, when multiple interrupts are not allowed.

#pragma extension function used

Exception processing handler

- #pragma interrupt *function-name*(*bank*)
  - This is terminated by an RTE instruction.
  - Rules for saving/restoring registers
    Do not save/restore SPC or SSR (because the sr_jsr embedded function is not used).
    Do not save/restore R0 to R7.
    Perform save/restore only for used registers other than those above.

Exception processing routine

- #pragma interrupt *function-name*(rts)
  - This is terminated by an RTS instruction.
  - Rules for saving/restoring registers
    Do not save/restore SPC or SSR.
    Do not save/restore R0 to R7.
    Perform save/restore only for used registers other than those above.

Exception processing flow

The above extension function can be used to create exception processing handlers and exception processing routines for the following exception processing flow:

(1) Hardware operation until an exception processing handler is called
Once an exception occurs, PC and SR at the time the exception occurred are saved to SPC and SSR, respectively, RB for SR is set to 1 (BANK1 is used as the general register), BL for SR is set to 1 (interrupt requests are masked), and transition is performed to the address for the corresponding exception cause.

(2) Exception processing handler
Registers other than R0 to R7 that are used within the exception processing handler are saved, and the exception processing routine is called by a JSR instruction. #pragma interrupt *function-name*(*bank*) is specified in the exception processing handler.

(3) Exception processing routine
Actual processing for each exception cause is specified. #pragma interrupt *function-name*(rts) is specified in the exception processing routine. An RTS instruction returns control from the exception processing routine to the exception processing handler.

(4) Exception processing handler
The registers saved in (2) are restored, and an RTE instruction returns control from exception processing to normal processing.
When the RTE instruction is executed, the hardware restores SPC and SSR, as saved in (1), to SP and SR.

Exception processing status flow

Figure 7-1 shows the flow of operation when an interrupt occurs.

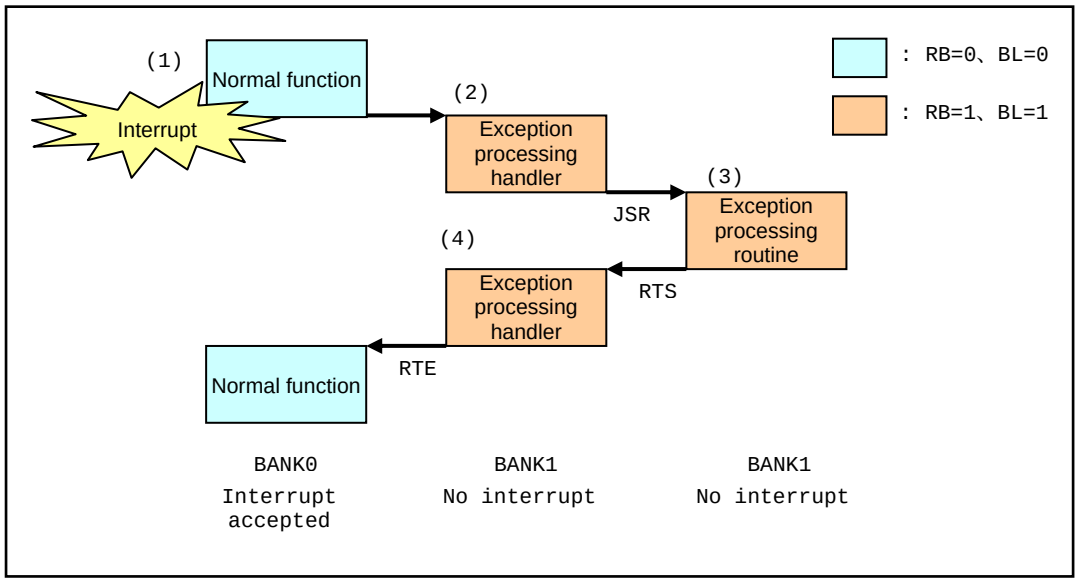

Figure 7-1

Example C source code

The following shows sample source code and its expanded assembly code.

Sample source

```
#include<machine.h>

extern void interrupt_handler(void);
extern void interrupt_routine(void);

#pragma interrupt interrupt_handler(bank)
#pragma interrupt interrupt_routine(rts)

/* Handling function  */
void interrupt_handler()
{
   /* Normal function call */
   interrupt_routine();
}

void interrupt_routine()
{
    /* Perform processing for each interrupt cause  */
}
```

Expanded assembly code

```
_interrupt_handler:
        STS.L      PR,@-R15
        BSR
_interrupt_routine
        NOP
        LDS.L      @R15+,PR
        RTE
        NOP
_interrupt_routine:
        RTS
        NOP
```

List 7-1

7.2 With Multiple Interrupts

The following shows how to code exception processing handlers and exception processing routines in C, when multiple interrupts are allowed.

Extension function used

Exception processing handler

- #pragma interrupt *function-name*(*bank*)
  - This is terminated by an RTE instruction.
  - Rules for saving/restoring registers:
    Do not save/restore SPC or SSR (because the `sr_jsr` embedded function is not used).
    Do not save/restore `R0` to `R7`.
    Perform save/restore only for used registers other than those above.

- void sr_jsr(void(*)(void)*func*, int *imask*) embedded function
  - This function is called by the exception processing routine.
  - *func*: the address of the exception processing routine.
  - *imask*: value set for the interrupt mask bit.
    When *imask* is from 1 to 15, *imask* is set for the mask bit.
    When *imask* is 0, the mask bit is not changed.

Exception processing routine

- #pragma interrupt *function-name*(sr_rts)
  - This is terminated by an RTS instruction.
  - During exit processing, `RB=1` and `BL=1` are set for SR.
  - Rules for saving/restoring registers:
    Do not save/restore SPC or SSR.
    Perform save/restore only for used registers other than those above.

Exception processing flow

The above extension function can be used to create exception processing handlers and exception processing routines for the following exception processing flow:

(1) Hardware operation until an exception processing handler is called
Once an exception occurs, PC and SR at the time the exception occurred are saved to SPC and SSR, respectively, `RB` for SR is set to `1` (`BANK1` is used as the general register), `BL` for SR is set to `1` (interrupt requests are masked) and transition is performed to the address for the corresponding exception cause.

(2) Exception processing handler
Registers other than `R0` to `R7` that are used within the exception processing handler are saved, and the `sr_jsr` embedded function is used to call the exception processing routine. When the `sr_jsr` function is used, `RB=0` (interrupt request masking is cancelled) and `BL=0` (`BANK0` is used as the general register) are set in the called exception processing routine, and code is generated to set the interrupt mask level to *imask*. `#pragma interrupt` *function-name*`(`*bank*`)` is specified in the exception processing handler.

(3) Exception processing routine
Actual processing for each exception cause is specified. `#pragma interrupt` *function-name*`(sr_rts)` is specified in the exception processing routine. When control returns from the exception processing routine to the exception processing handler, `RB=1` and `BL=1` are set for SR.
When this exception processing routine is being executed, interrupts with levels higher than that set in (2) for *imask* may be accepted.

(4) Exception processing handler

The registers saved in (2) are restored, and an RTE instruction returns control from exception processing to normal processing.

When the RTE instruction is executed, the hardware restores SPC and SSR, as saved in (1), to PC and SR.

Exception processing status flow (with multiple interrupts)

Figure 7-2 shows operation when an interrupt of level 8 occurs after one of level 5.

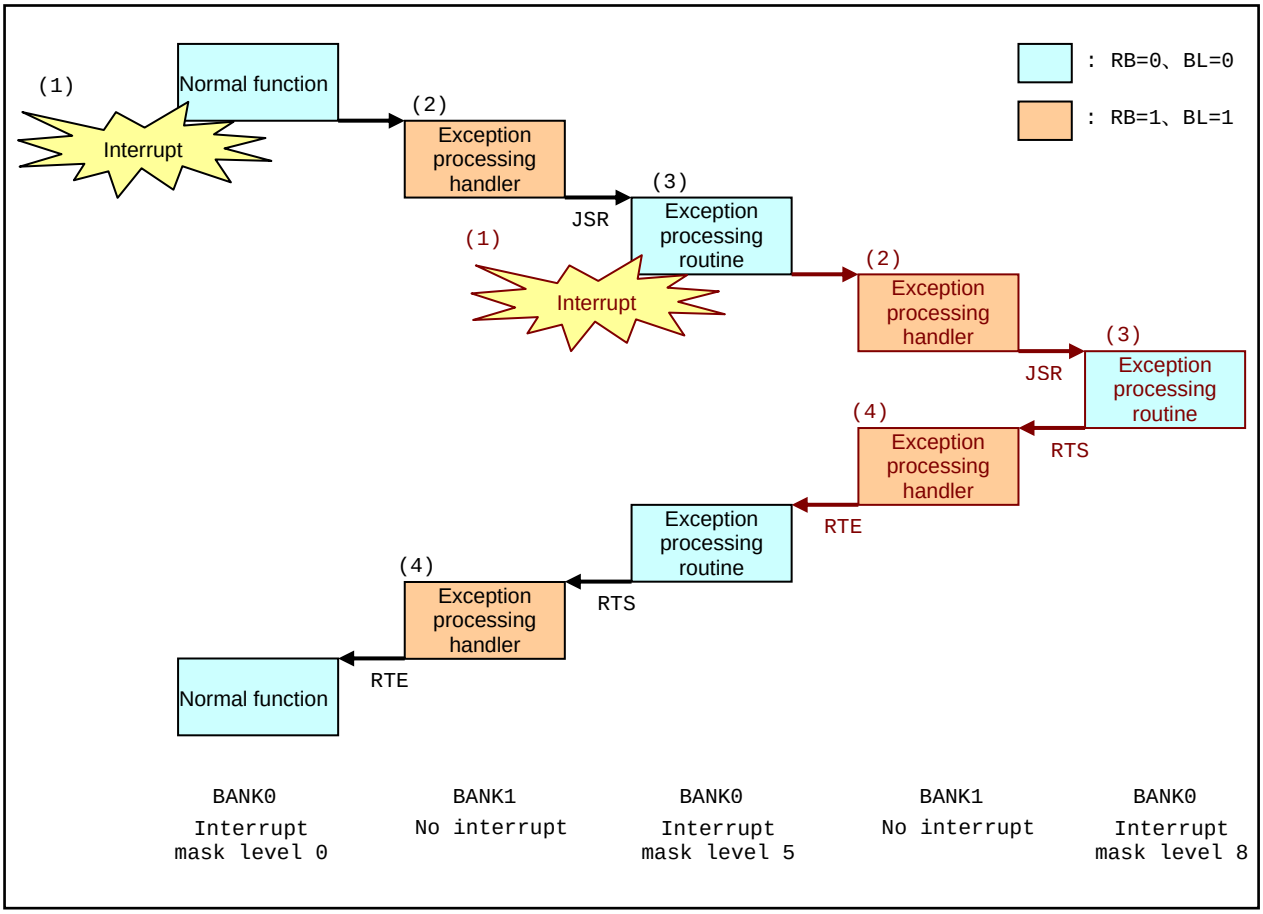

Figure 7-2

Example C source code

The following shows sample source code and its expanded assembly code.

Sample source

```
#include<machine.h>

extern void interrupt_handler(void);
extern void interrupt_routine(void);

#pragma interrupt interrupt_handler(bank)
#pragma interrupt interrupt_routine(sr_rts)

/*  Handling function  */
void interrupt_handler()
{
    /* sr_jsr and interrupt mask bit specified
       in processing function */
    sr_jsr(interrupt_routine, 5);
}

void interrupt_routine()
{
    /*  Perform processing for each interrupt
       cause */
}
```

Expanded assembly code

```
_interrupt_handler:
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          STC         SSR,@-R15
          STC         SPC,@-R15
          STC         SR,R4
          MOV.L       L12,R1      ; H'CFFFFF0F
          MOV         #80,R5      ; H'00000050
          MOV.L       L12+4,R14
                          ; _interrupt_routine
          AND         R1,R4
          OR          R5,R4
          LDC         R4,SR
          JSR         @R14
          NOP
          LDC         @R15+,SPC
          LDC         @R15+,SSR
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTE
          NOP
_interrupt_routine:
          MOV.L       R0,@-R15
          MOV.L       R1,@-R15
          STC         SR,R0
          MOV.L       L12+8,R1    ; H'30000000
          OR          R1,R0
          MOV.L       @R15+,R1
          LDC         R0,SR
          RTS
          LDC.L       @R15+,R0_BANK
```

List 7-2

8. Frequently Asked Questions

8.1 End Processing

Q:

When can the `abort()` function in the main routine (*project-name*.c) be used?

A:

The `abort` function needs to be used when exception processing is performed in C++. If the function is not defined, an error will occur during linkage.
Since the `abort` function is called when an exception occurs, use the `sleep()` and other commands to perform end processing, to prevent system abuse.

8.2 C++ Functions and Reciprocal C Function Calls

Q:

I know that `extern "C"` { and } are used to enclose function declarations, but why do they need to be enclosed?

A:

When a C function is called from a C++ function, the `extern "C"` declaration needs to be specified for prototype declarations of C functions within C++ source. When a C++ function is called from a C function, the `extern "C"` declaration needs to be specified for prototype declarations of C++ functions within C++ source.
Since C++ allows functions to be defined multiple times, there may be multiple functions with the same function name. This means that the compiler manages symbol names internally such as by appending the name of an argument to the function name. Since C functions cannot be defined more than once, this kind of symbol name management is not performed.
When the `extern "C"` declaration is performed in a C++ function, the way in which symbol names are managed is the same as for C functions. This enables reciprocal calls between C functions and C++ functions.
Note that C++ functions declared using `extern "C"` cannot be defined multiple times.

- An `extern "C"` declaration can be used to reference a function in a C object program.

```
(C++ program)

extern "C" void CFUNC();
void main(void)
{
    X XCLASS;
    XCLASS.SetValue(10);

    CFUNC();
}
```

```
(C program)

extern void CFUNC();
void CFUNC()
{
    while(1)
    {
        a++;
    }
}
```

- An `extern "C"` declaration can be used to reference a function in a C++ object program.

```
(C program)

void CFUNC()
{
    CPPFUNC();
}
```

```
(C++ program)

extern "C" void CPPFUNC();
void CPPFUNC(void)
{
    while(1)
    {
        a++;
    }
}
```

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.01.07	—	First edition issued

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.