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-1, SH-2, and SH-2A

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

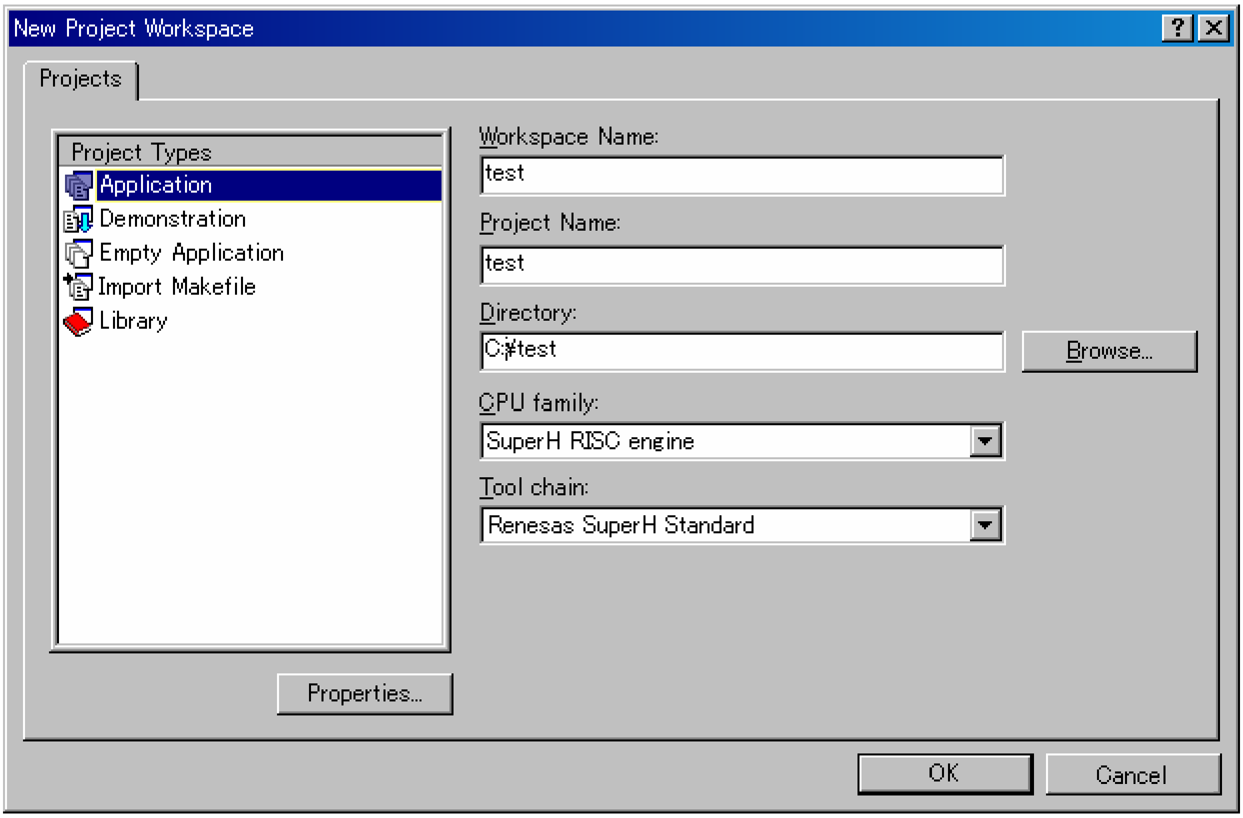
1. Generating a Sample Program

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 SH7046 selected for CPU types is selected for illustration purpose only.

(1) Create a new workspace

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



Workspace Name: test

Project Name: test

Directory: C:¥test

CPU family: SuperH RISC engine

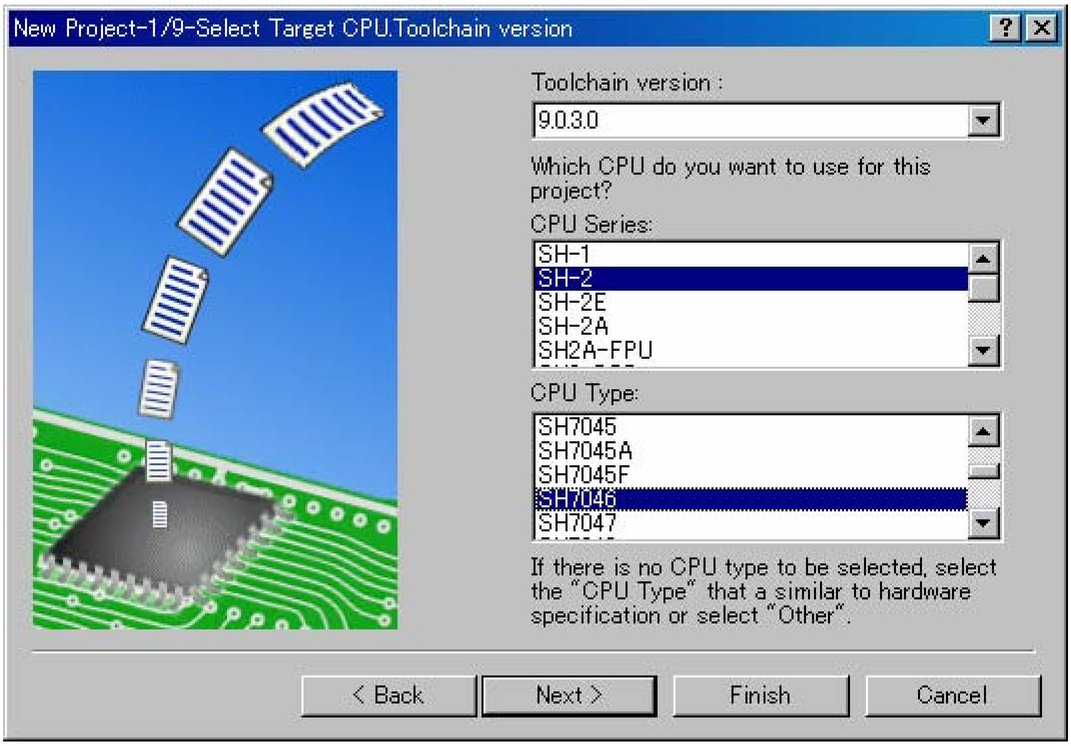
Tool chain: Renesas SuperH Standard

Figure 1-1

(2) Select the CPU

For **CPU Series**, select **SH-2**.

For **CPU Type**, select **SH7046**.



New Project-1/9-Select Target CPU.Toolchain version

Toolchain version : 9.0.3.0

Which CPU do you want to use for this project?

CPU Series: SH-2

CPU Type: SH7046

If there is no CPU type to be selected, select the "CPU Type" that a similar to hardware specification or select "Other".

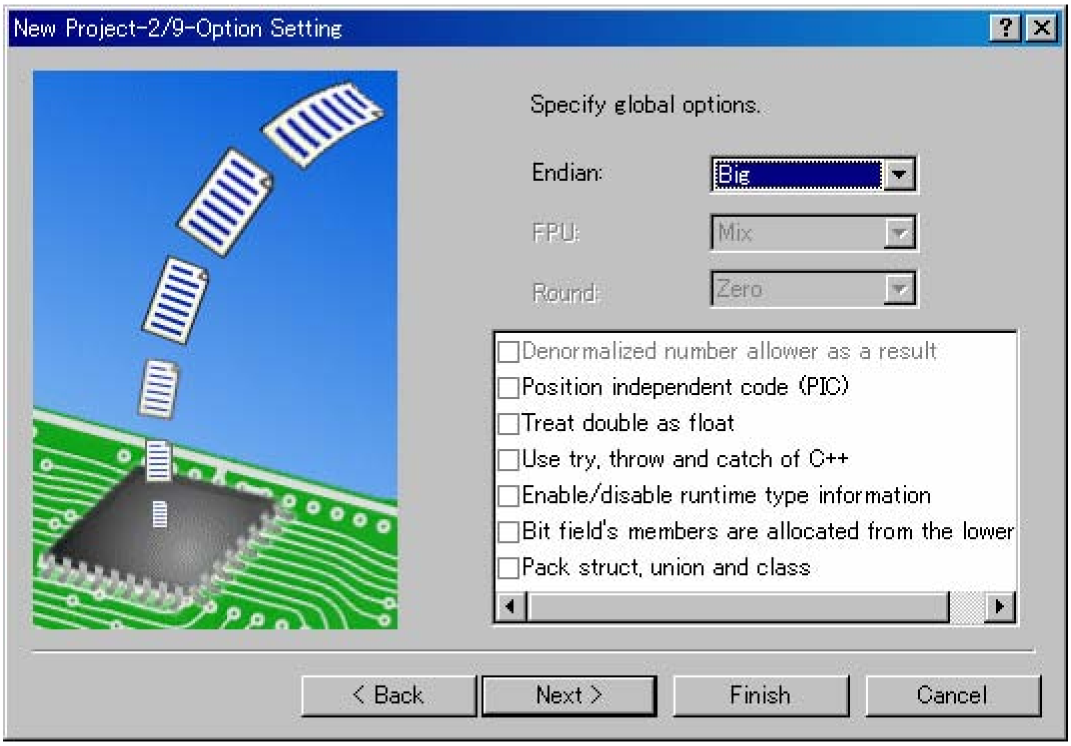
Figure 1-2

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.c`, `vecttbl.c`, `iodefine.h`, and `vect.h`, 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.



New Project-2/9-Option Setting

Specify global options.

Endian: Big

FPU: Mix

Round: Zero

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

Figure 1-3

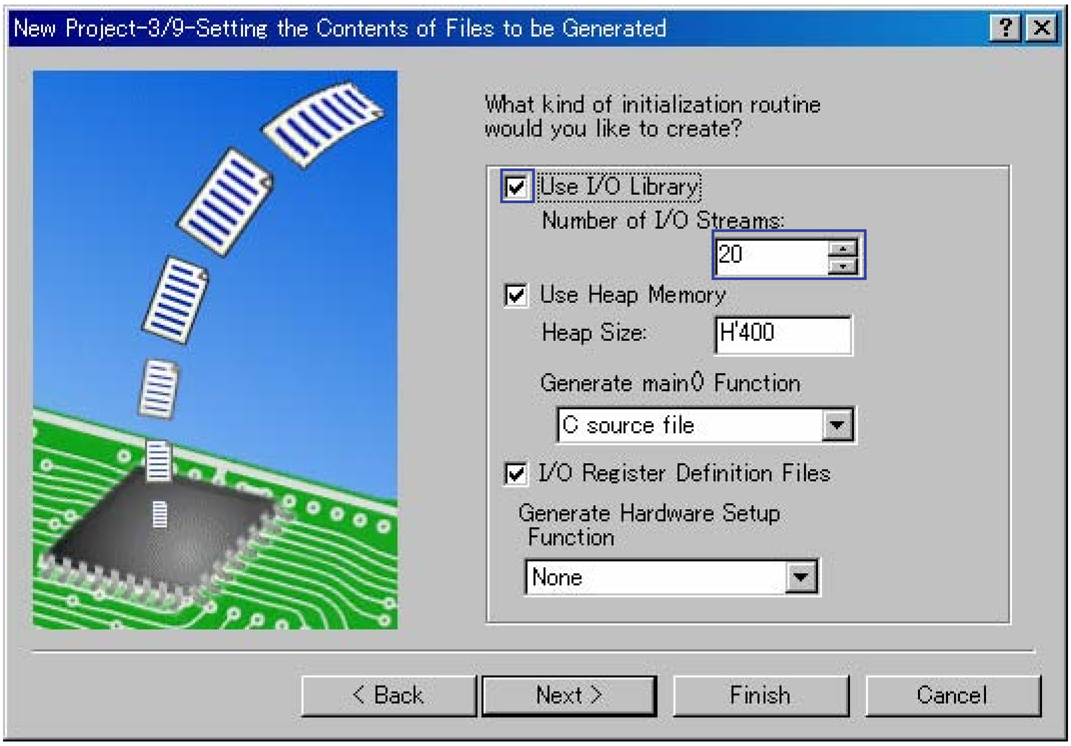
Note:

- 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**.



New Project-3/9-Setting the Contents of Files to be Generated

What kind of initialization routine would you like to create?

☑ Use I/O Library

Number of I/O Streams: 20

☑ Use Heap Memory

Heap Size: H'400

Generate main() Function: C source file

☑ I/O Register Definition Files

Generate Hardware Setup Function: None

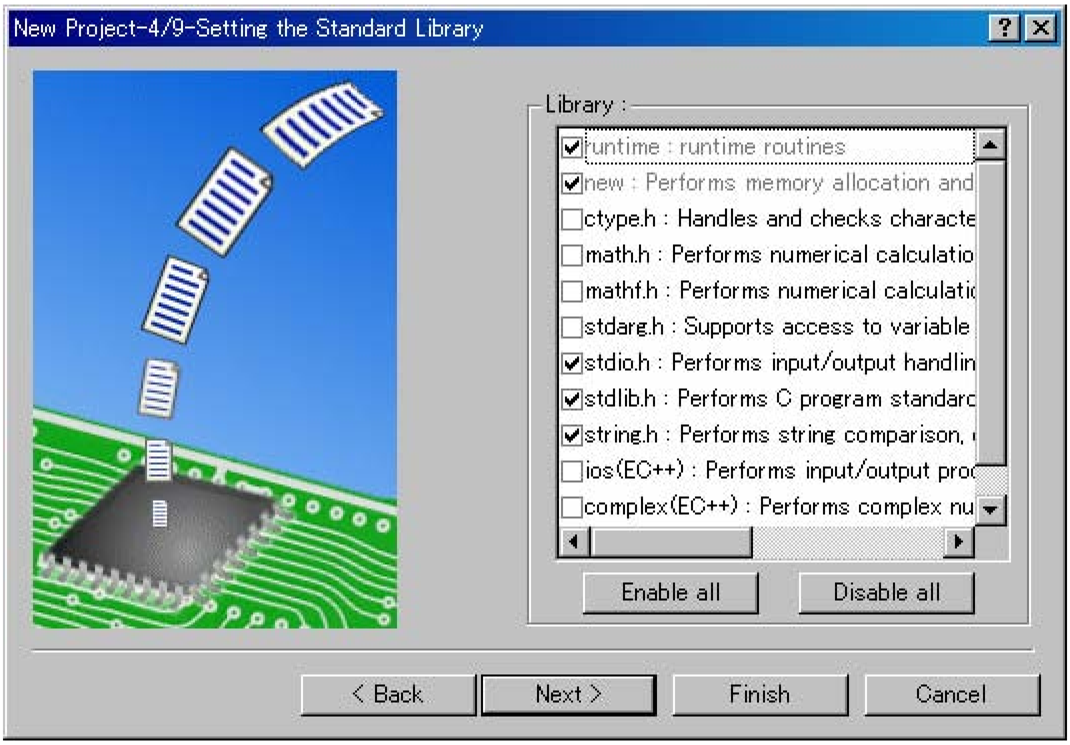
Figure 1-4

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.



- ☑ runtime : runtime routines
- ☑ new : Performs memory allocation and
- ☐ ctype.h : Handles and checks characte
- ☐ math.h : Performs numerical calculatio
- ☐ mathf.h : Performs numerical calculati
- ☐ stdarg.h : Supports access to variable
- ☑ stdio.h : Performs input/output handlin
- ☑ stdlib.h : Performs C program standarc
- ☑ string.h : Performs string comparison,
- ☐ ios(EC++) : Performs input/output proc
- ☐ complex(EC++) : Performs complex nu

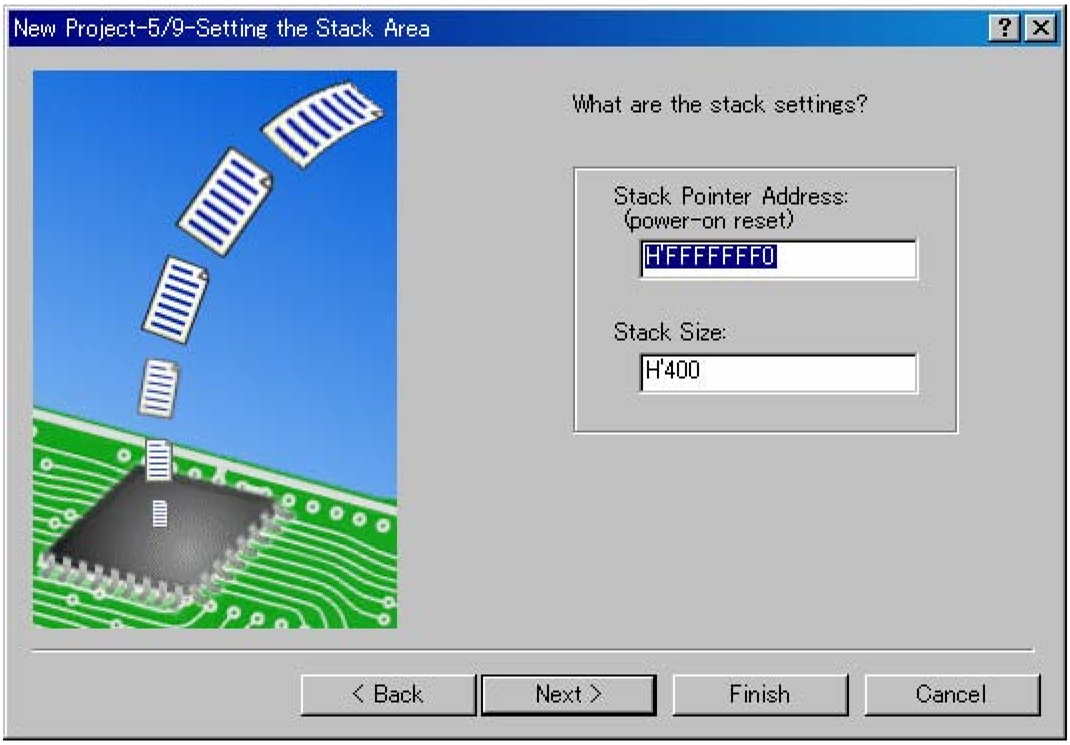
Figure 1-5

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.



New Project-5/9-Setting the Stack Area

What are the stack settings?

Stack Pointer Address:
(power-on reset)
H'FFFFFFF0

Stack Size:
H'400

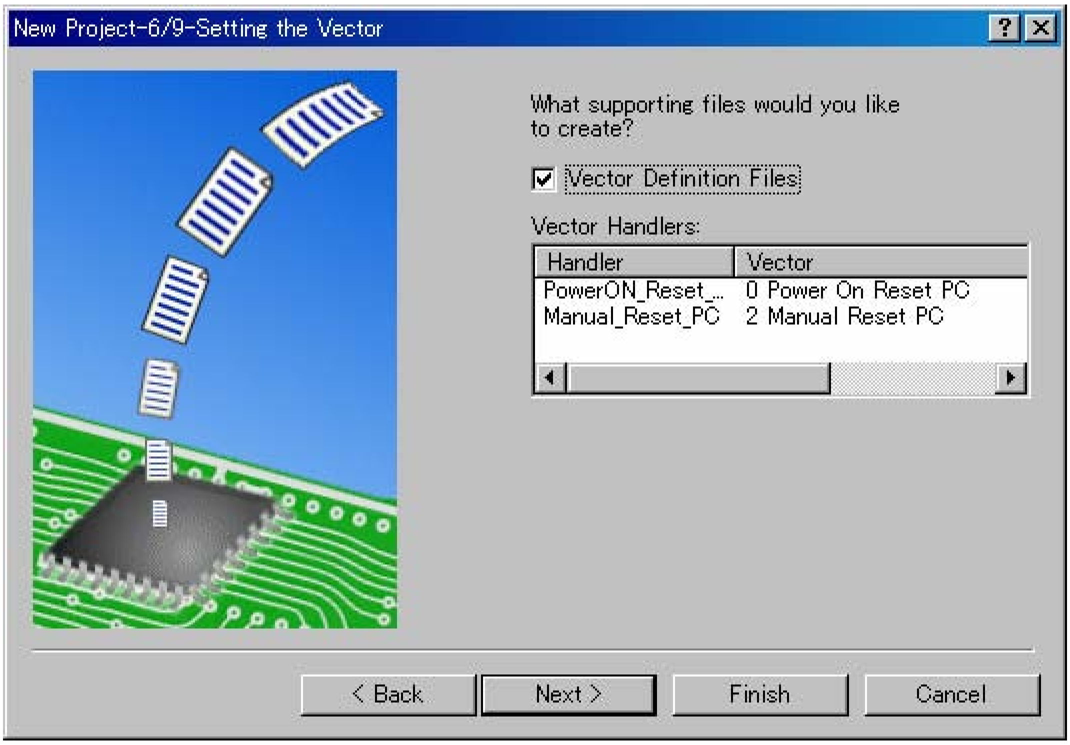
Figure 1-6

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.



New Project-6/9-Setting the Vector

What supporting files would you like to create?

☑ Vector Definition Files

Vector Handlers:

Handler	Vector
PowerON_Reset_...	0 Power On Reset PC
Manual_Reset_PC	2 Manual Reset PC

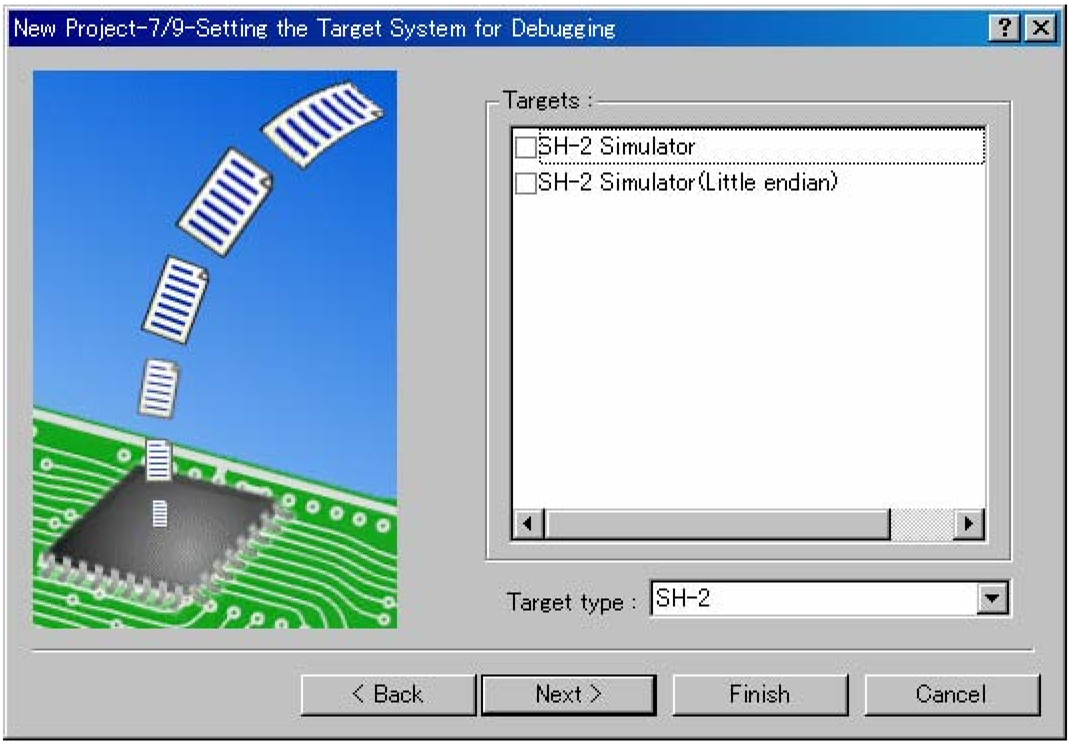
Figure 1-7

Note:

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

(8) Set the debugger target

Proceed with the default settings.



New Project-7/9-Setting the Target System for Debugging

Targets :

☐ SH-2 Simulator

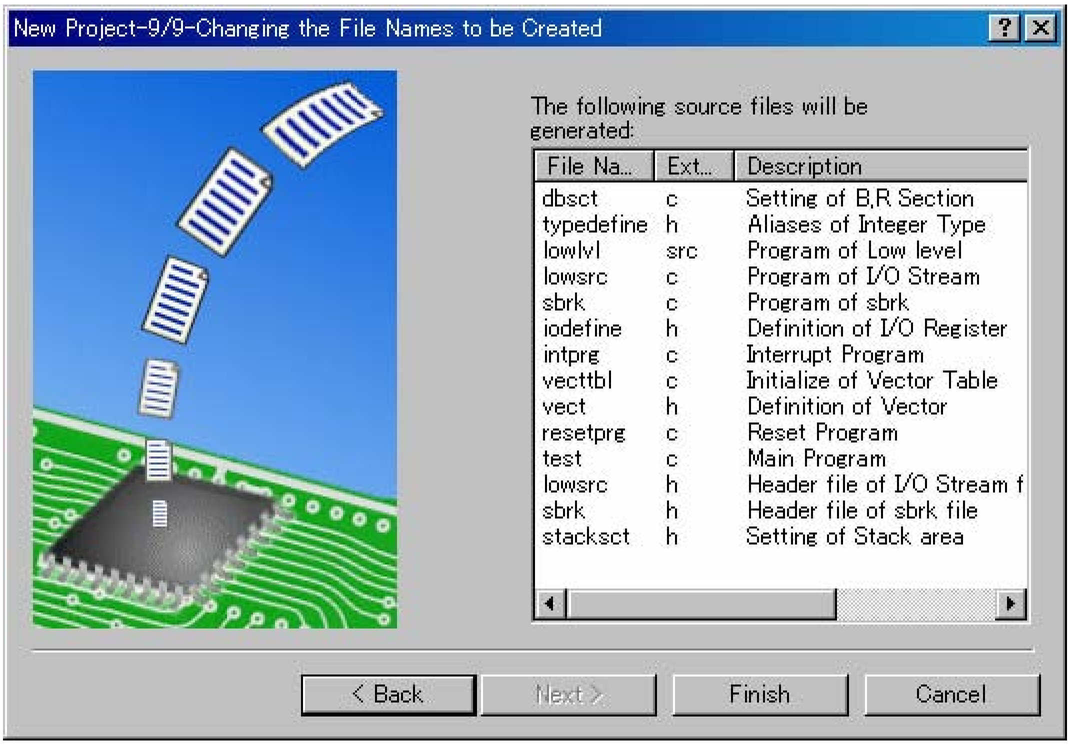
☐ SH-2 Simulator(Little endian)

Target type : SH-2

Figure 1-8

(9) Change the name of the generation file

Select **Finish**.



New Project-9/9-Changing the File Names to be Created

The following source files will be generated:

File Na...	Ext...	Description
dbsct	c	Setting of B,R Section
typedefine	h	Aliases of Integer Type
lowlvl	src	Program of Low level
lowsrc	c	Program of I/O Stream
sbrk	c	Program of sbrk
iodefine	h	Definition of I/O Register
intprg	c	Interrupt Program
vecttbl	c	Initialize of Vector Table
vect	h	Definition of Vector
resetprg	c	Reset Program
test	c	Main Program
lowsrc	h	Header file of I/O Stream f
sbrk	h	Header file of sbrk file
stacksct	h	Setting of Stack area

Figure 1-9

1.2 List of Generation Files

The sample files automatically generated in the project generator are as follows.

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

`lowlvl.src`	I/O low-level interface routine • Called from low-level interface routines (`write` and `read`), in which `_charput` and `_charget` are defined. • This program only runs on the simulator. • Generated according to the *(4) Use I/O library* specification. For details, see *5.2 I/O (lowlvl.src, lowsrc.c,* lowsrc.h).
`dbsct.c`	Specification of memory initialization target • Targets for RAM initialization and transfer processing from ROM to RAM areas is defined. For details, see *4.1 Memory Initialization Function _INTSCT (dbsct.c)*.
`intprg.c`	Interrupt function • Defines the interrupt function (dummy). • Generated according to the *(7) Vector Definition Files* specification. For details, see *3.3 Exception Processing Function (intprg.c, vect.h)*.
`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 *(4) Use I/O library* specification. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)*.
`resetprg.c`	Reset function • Defines the reset function (`PowerON_Reset_PC`). • Generated according to the *(7) Vector Definition Files* specification. For details, see *2.3 Reset Function (resetprg.c)*.
`sbrk.c`	Low-level interface routine for memory management • Defines the low-level interface routine for memory management (`sbrk`). • Generated according to the *(4) Use Heap Memory usage* specification. For details, see *5.1 Memory Management (sbrk.c, sbrk.h)*.
`test.c` `(test.cpp)`	Main routine • Defines the main function. Also defines the abort function when C++ is used. • The file name is that specified in *(1) Project Name*.
`vecttbl.c`	Vector table • Defines the exception processing vector table. • Generated according to the *(7) Vector Definition Files*. For details, see *2.1 Reset Vector Table (vecttbl.c)*.
`lowsrc.h`	I/O low-level function header • Defines the `IOSTREAM` macro that specifies the file handler count (number of files that can be used concurrently). • Generated according to the *(4) Use I/O library* specification. • Reflected in the value set for *(4) Number of I/O Streams*. For details, see *5.2 I/O (lowlvl.src, lowsrc.c, lowsrc.h)*.

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

`sbrk.h`	Low-level used header for memory management • Defines the `HEAPSIZE` macro that specifies the overall size of the heap area. • Generated according to the *(4) Use Heap Memory* specification. • Reflected in the value set for *(4) Heap Size*. 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 *(7) Vector Definition Files* specification. • Reflected in the value set for *(6) Stack Size*. For details, see *2.2 Setting Stack Size (stacksct.h)*.
`typedefine.h`	Type alias declaration header • Defines the type alias declaration.
`vect.h`	Vector table header • Defines the prototype declaraction for the reset function and interrupt function. • Specifies `#pragma interrupt` for the interrupt function. • Generated according to the *(7) Vector Definition Files* specification. For details, see *3.3 Exception Processing Function (intprg.c, vect.h)*.

2. Reset Processing

The following explains the operations once power-on reset is performed for the sample program generated by HEW.

2.1 Reset Vector Table (vecttbl.c)

When power-on reset is performed, the following is performed on the CPU.

1. The initial value (execution start address) of the program counter (PC) is taken from the exception processing vector table.
2. The initial value of the stack pointer (SP) is taken from the exception processing vector table.
3. The vector base register (VBR) is cleared to `H'00000000`, and the interrupt mask bit (I3 to I0) of the status register (SR) is set to `H'F` (`B'1111`).
4. The values taken from each exception processing vector table are set with each PC and SP, and program execution starts.

The exception processing vector table is a data table from which the CPU obtains address information for the jump destination for a given exception cause, when exception processing occurs. During reset exception processing, the initial values of the program counter (PC) and stack pointer (SP) are obtained from the addresses in *Table 2-1*. As such, these setting values need to be set before reset.

Table 2-1 Exception processing vector table (reset cause)

Exception cause		Vector number	Vector table address
Power-on reset	PC	0	`H'00000000 ~ H'00000003`
	SP	1	`H'00000004 ~ H'00000007`
Manual reset	PC	2	`H'00000008 ~ H'0000000B`
	SP	3	`H'0000000C ~ H'0000000F`

In the sample program, the exception processing vector table for reset causes is separate from the exception processing vector table for other exception processing. The exception processing vector table for reset causes is defined in `vecttbl.c` as `RESET_Vectors` (List 2-1).

```
#pragma section VECTTBL          ...(a)

void *RESET_Vectors[] = {
//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 Power On Reset PC
PowerON_Reset_PC,                ...(b)
//;<<VECTOR DATA END (POWER ON RESET)>>
// 1 Power On Reset SP
    __secend("S"),               ...(c)
//;<<VECTOR DATA START (MANUAL RESET)>>
//;2 Manual Reset PC
Manual_Reset_PC                  ...(d)
//;<<VECTOR DATA END (MANUAL RESET)>>
// 3 Manual Reset SP
    __secend("S")                ...(e)
};
```

List 2-1

Explanation of List 2-1

The `#pragma section VECTTBL` specification places the `RESET_Vectors` array in the DVECTTBL section. (a)

The address of the power-on reset function (`PowerON_Reset_PC`) is set in the first element of the array. (b)

The end address of the S section is set in the second element of the array. (c)

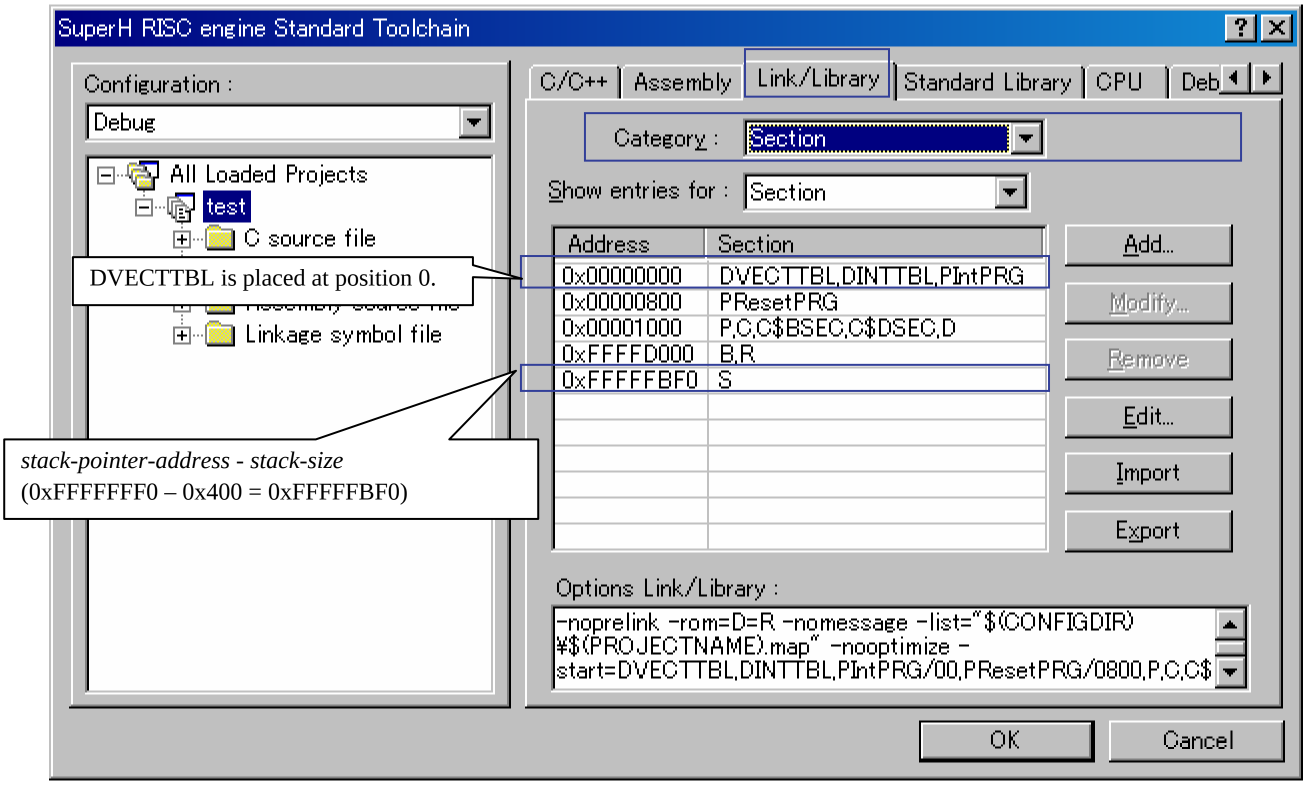
The address of the manual reset function (`Manual_Reset_PC`) is set in the third element of the array. (d)

The end address of the S section is set in the fourth element of the array. (e)

`__secend` is a section address operator (not a function), which obtains the end address + 1 for the section enclosed in double quotation marks (").

The `RESET_Vectors` array can be placed in the 0 position to become the exception processing vector table for reset causes.

In order to place this array in the 0 position, it needs to be placed in the 0 position for the `DVECTTBL` section in the linker section setting. In the sample project, this is set as in Figure 2-1.



Configuration : Debug

Category : Section

Show entries for : Section

Address	Section
0x00000000	DVECTTBL,DINTTBL,PIntPRG
0x00000800	PResetPRG
0x00001000	P,C,C$BSEC,C$DSEC,D
0xFFFFD000	B,R
0xFFFFFBF0	S

Options Link/Library :

```
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -nooptimize -
start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/0800,P,C,C$
```

DVECTTBL is placed at position 0.

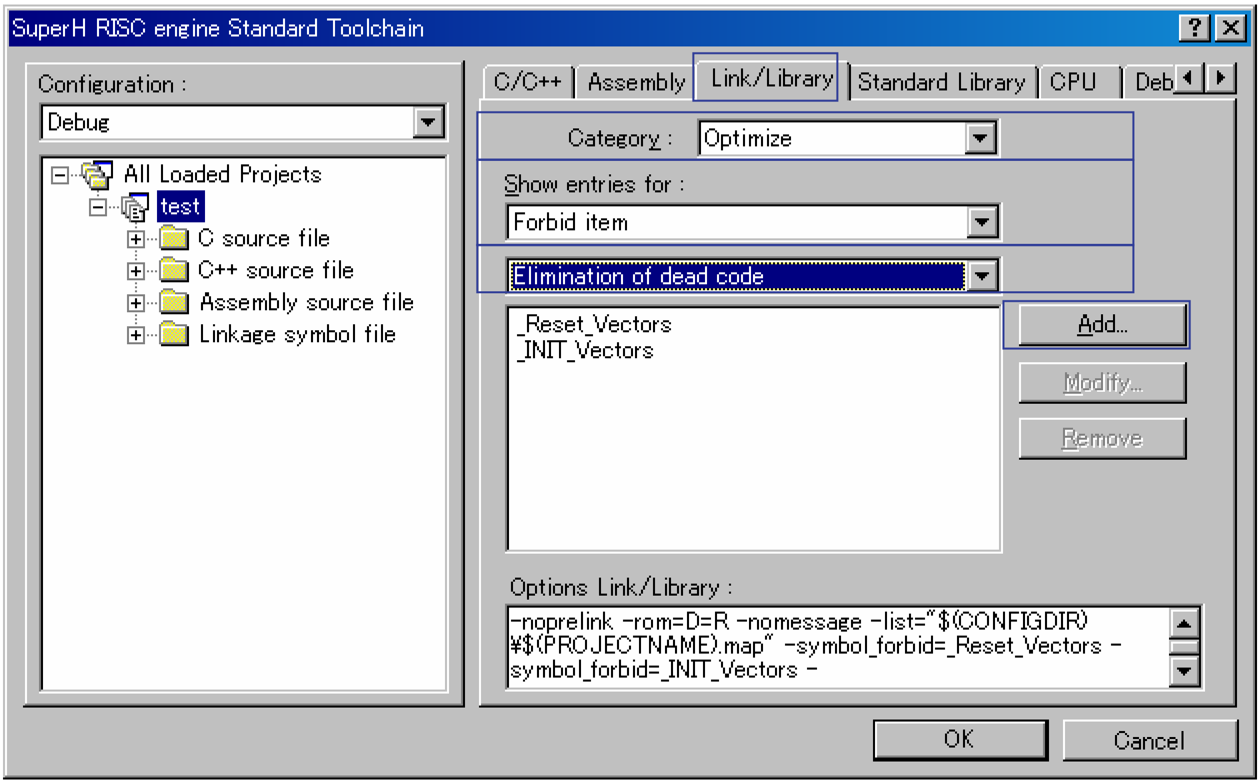
stack-pointer-address - stack-size
(0xFFFFFFF0 – 0x400 = 0xFFFFFBF0)

Figure 2-1

Precautions regarding linker optimization

When using optimization to delete unreferenced symbols for the linker, the vector tables (`RESET_Vectors` and `INT_Vectors`) may also end up being deleted by the optimization. To avoid vector table deletion, the vector table symbols need to be specified for `Elimination of dead code` in the Link/Library, as shown in Figure 2-2.

Note that when a symbol is specified, an underscore (_) needs to be appended to the beginning of the name defined in the program, for the C/C++ variable name or C function name. Likewise, for C++ functions, names defined in programs including argument arrays need to be enclosed in double quotation marks ("), except when the argument is void, in which case function-name() is to be specified.



Configuration : Debug

Category : Optimize

Show entries for : Forbid item

Elimination of dead code

_Reset_Vectors
_INIT_Vectors

Options Link/Library :

```
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -symbol_forbid=_Reset_Vectors -
symbol_forbid=_INIT_Vectors -
```

Figure 2-2

2.2 Setting Stack Size (stacksct.h)

`#pragma stacksize` can be specified in `stacksct.h` (List 2-2) to reserve a 0x400-byte stack area (S section) in the compiler.

When a stack is used from higher addresses down to lower addresses, the start address of the S section needs to be (*stack-pointer-address* - *stack-size*). In the sample project, since `0xFFFFFFF0` is set for the stack pointer address (Figure 1-6), the start address of the S section during section placement for the optimization linkage editor is `0xFFFFFBF0` (`0xFFFFFFF0` - `0x400`) (Figure 2-1).

```
#pragma stacksize 0x400
```

List 2-2

2.3 Reset Function (resetprg.c)

The following shows the processing contents for the `PowerON_Reset_PC` 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 0x10`	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 _UINT INT_Vectors;`	`INT_Vectors` is referenced.

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

`void PowerON_Reset_PC(void);`	A `PowerON_Reset_PC` prototype declaration is performed.
`void Manual_Reset_PC(void);`	A `Manual_Reset_PC` prototype declaration is performed.
`void main(void);`	A `main` prototype declaration is performed.

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

`#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);`	A prototype declaration is performed for the I/O-related standard library end function.

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

`//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);`	When `HardwareSetup` is called, a prototype declaration is performed.

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

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_PC
```

The entry function for the reset function is specified. When specification is performed in the entry function, save and restore codes for registers can be suppressed.

```
void PowerON_Reset_PC(void)
{
    set_vbr((void *)((_UBYTE *)&
INT_Vectors - INT_OFFSET));
```

Setting processing is performed for the vector base register (VBR). This register can be used to set a non-reset exception processing vector table in any address.
For details, see *3.2. Vector Base Register (VBR) Settings (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_PC` cannot be avoided.

```
//#pragma entry Manual_Reset_PC
void Manual_Reset_PC(void)
{
}
```

This is the manual reset function (dummy).

3. Non-reset Exception Processing

Non-reset exception causes include exceptions due to address errors, interrupts, and instructions. When an exception occurs, the start address for exception processing is taken from the (VBR + *vector-table-address-offset*) address, and exception processing is performed.

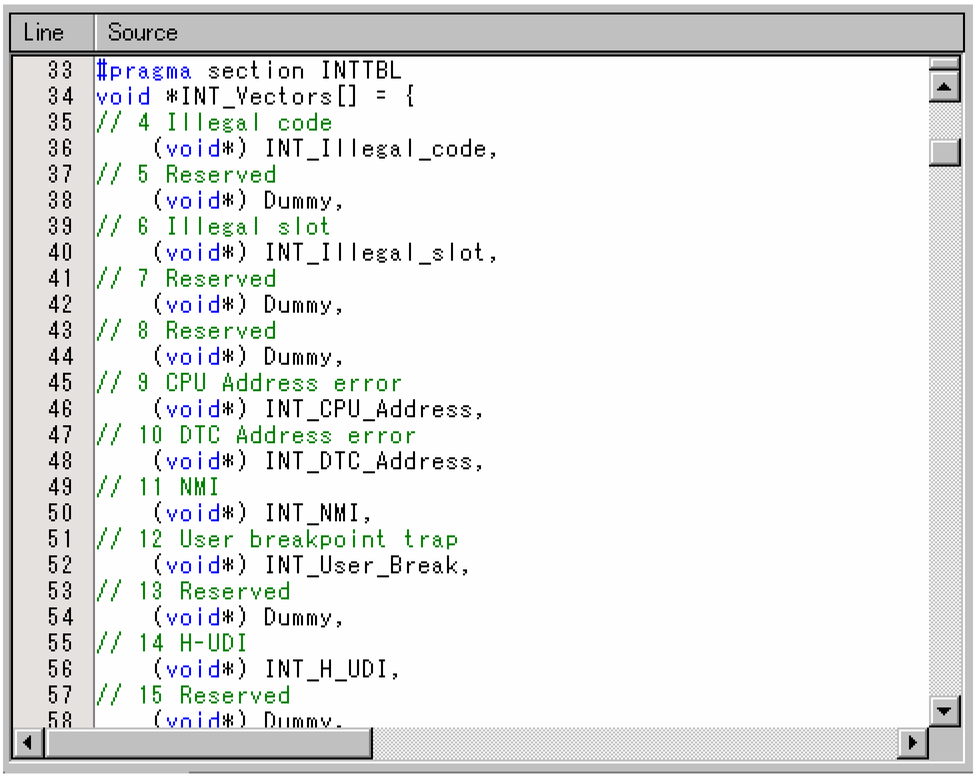
3.1 Non-reset Exception Processing Vector Table (vecttbl.c)

The start address for exception processing needs to be set in the exception processing vector table. Each different vector number and vector table address offset (offset value from VBR) is allocated to each exception cause. Table 3-1 lists the vector numbers and vector table address offsets.

Table 3-1 Exception processing vector table

Exception cause		Vector number	Vector table address offset
Power-on reset	PC	0	H'00000000 ~ H'00000003
	SP	1	H'00000004 ~ H'00000007
Manual reset	PC	2	H'00000008 ~H'0000000B
	SP	3	H'0000000C ~H'0000000F
General invalid instruction		4	H'00000010 ~H'00000013
(reserved by the system)		5	H'00000014 ~H'00000017
Slot invalid instruction		6	H'00000018 ~H'0000001B
(reserved by the system)		7	H'0000001C ~H'0000001F
		8	H'00000020 ~H'00000023
CPU address error		9	H'00000024 ~H'00000027
DTC address error		10	H'00000028 ~H'0000002B
Interrupt	NMI	11	H'0000002C ~H'0000002F
	User break	12	H'00000030 ~H'00000033
(Reserved by the system)		13	H'00000034 ~H'00000037
		14	H'00000038 ~H'0000003B
		15	H'0000003C ~H'0000003F
		⋮	⋮
		31	H'0000007C ~H'0000007F
Trap instruction (user vector)		32	H'00000080 ~H'00000083
		⋮	⋮
		63	H'000000FC ~H'000000FF
Interrupt	RQ0	64	H'00000100 ~H'00000103
	IRQ1	65	H'00000104 ~H'00000107
	IRQ2	66	H'00000108 ~H'0000010B
	IRQ3	67	H'0000010C ~H'0000010F
	Reserved by the system	68	H'00000110 ~H'00000113
	Reserved by the system	69	H'00000114 ~H'00000117
	Reserved by the system	70	H'00000118 ~H'0000011B
	Reserved by the system	71	H'0000011C ~H'0000011F
Built-in peripheral modules		72	H'00000120 ~H'00000123
		⋮	⋮
		255	H'000003FC ~H'000003FF

In the sample program, since the exception processing vector table for non-reset exception cause is allocated to an arbitrary address, the exception processing vector table for reset causes and the exception processing vector table for other exception processing are defined separately. The exception processing vector table for other exception processing is defined in `vecttbl.c` as `INT_Vectors` (Figure 3-1).



```
#pragma section INTTBL
void *INT_Vectors[] = {
// 4 Illegal code
    (void*) INT_Illegal_code,
// 5 Reserved
    (void*) Dummy,
// 6 Illegal slot
    (void*) INT_Illegal_slot,
// 7 Reserved
    (void*) Dummy,
// 8 Reserved
    (void*) Dummy,
// 9 CPU Address error
    (void*) INT_CPU_Address,
// 10 DTC Address error
    (void*) INT_DTC_Address,
// 11 NMI
    (void*) INT_NMI,
// 12 User breakpoint trap
    (void*) INT_User_Break,
// 13 Reserved
    (void*) Dummy,
// 14 H-UDI
    (void*) INT_H_UDI,
// 15 Reserved
    (void*) Dummy,
```

Figure 3-1

3.2 Vector Base Register (VBR) Settings (set_vbr function)

By setting an arbitrary address in the VBR, the non-reset exception processing vector table can be placed in any address. The 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 `INT_Vectors` placement address. Since `INT_Vectors` is specified by using `#pragma section` in the `DINTTBL` section, `DINTTBL` can be placed in any section to place `INT_Vectors` in a given address.

```
resetprg.c
   #define INT_OFFSET 0x10
      …
   set_vbr((void *)((_UBYTE *)&INT_Vectors - INT_OFFSET));
```

List 3-1

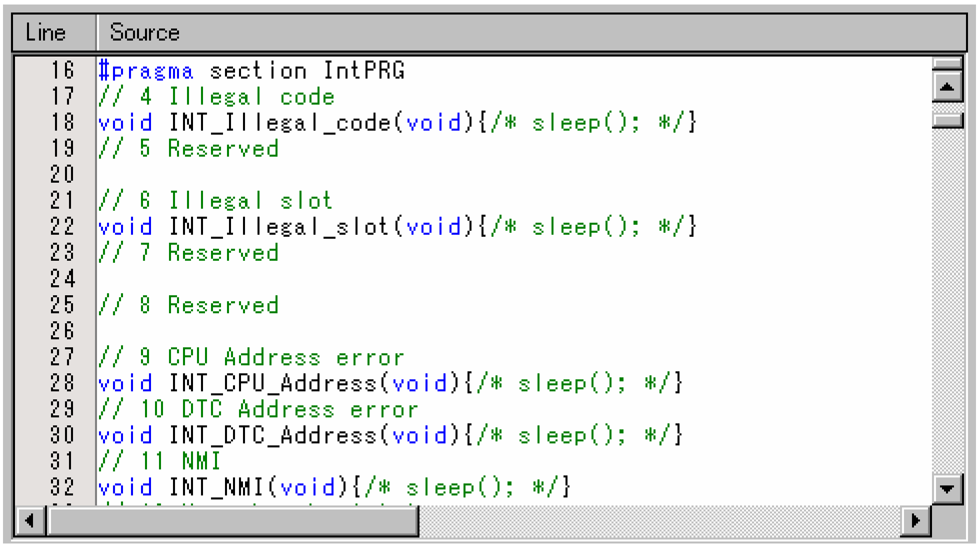
How the value set for the VBR is calculated

The non-reset vector table (`INT_Vectors`) starts from general invalid instructions.

As such, from the `INT_Vectors` address the offset value (`INT_OFFSET` (`0x00000010`)) of the general invalid instruction can be taken from the value set for the general invalid instruction to get the value set for the VBR.

3.3 Exception Processing Function (intprg.c, vect.h)

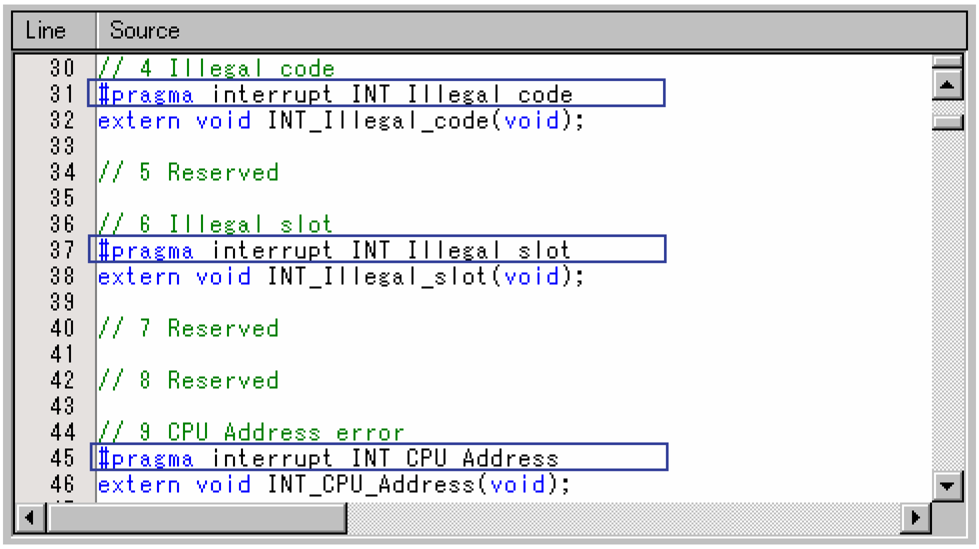
Non-reset exception processing (such as the `INT_Illegal_code` function and `INT_Illegal_slot` function) is defined as dummy functions in `intprg.c` (Figure 3-2).



```
#pragma section IntPRG
// 4 Illegal code
void INT_Illegal_code(void){/* sleep(); */}
// 5 Reserved

// 6 Illegal slot
void INT_Illegal_slot(void){/* sleep(); */}
// 7 Reserved

// 8 Reserved

// 9 CPU Address error
void INT_CPU_Address(void){/* sleep(); */}
// 10 DTC Address error
void INT_DTC_Address(void){/* sleep(); */}
// 11 NMI
void INT_NMI(void){/* sleep(); */}
```

Figure 3-2

These non-reset exception processing functions are specified in `vect.h` (Figure 3-3) using `#pragma interrupt`. Code for functions in which `#pragma interrupt` is specified are automatically generated as interrupt functions by the compiler. Return from interrupts by RTE instruction, required register save recovery, and other processing are performed in an interrupt function.



```
// 4 Illegal code
#pragma interrupt INT_Illegal_code
extern void INT_Illegal_code(void);

// 5 Reserved

// 6 Illegal slot
#pragma interrupt INT_Illegal_slot
extern void INT_Illegal_slot(void);

// 7 Reserved

// 8 Reserved

// 9 CPU Address error
#pragma interrupt INT_CPU_Address
extern void INT_CPU_Address(void);
```

Figure 3-3

4. Memory Initialization

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

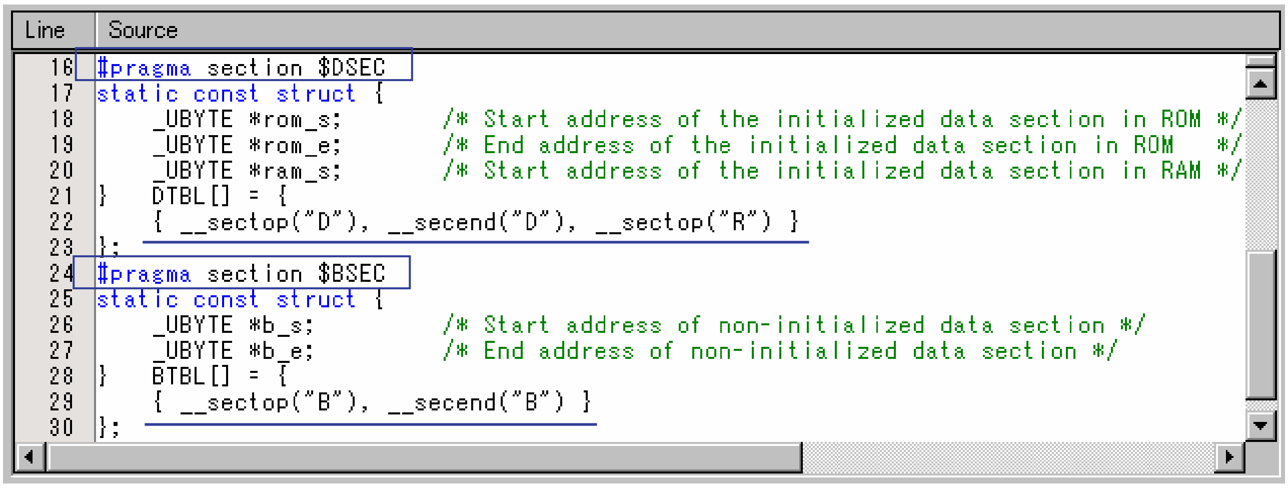
The `_INITSCT` function performs the following initialization processing.

- 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`.



```
#pragma section $DSEC
static const struct {
    _UBYTE *rom_s;       /* Start address of the initialized data section in ROM */
    _UBYTE *rom_e;       /* End address of the initialized data section in ROM   */
    _UBYTE *ram_s;       /* Start address of the initialized data section in RAM */
}   DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") }
};
#pragma section $BSEC
static const struct {
    _UBYTE *b_s;         /* Start address of non-initialized data section */
    _UBYTE *b_e;         /* End address of non-initialized data section */
}   BTBL[] = {
    { __sectop("B"), __secend("B") }
};
```

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

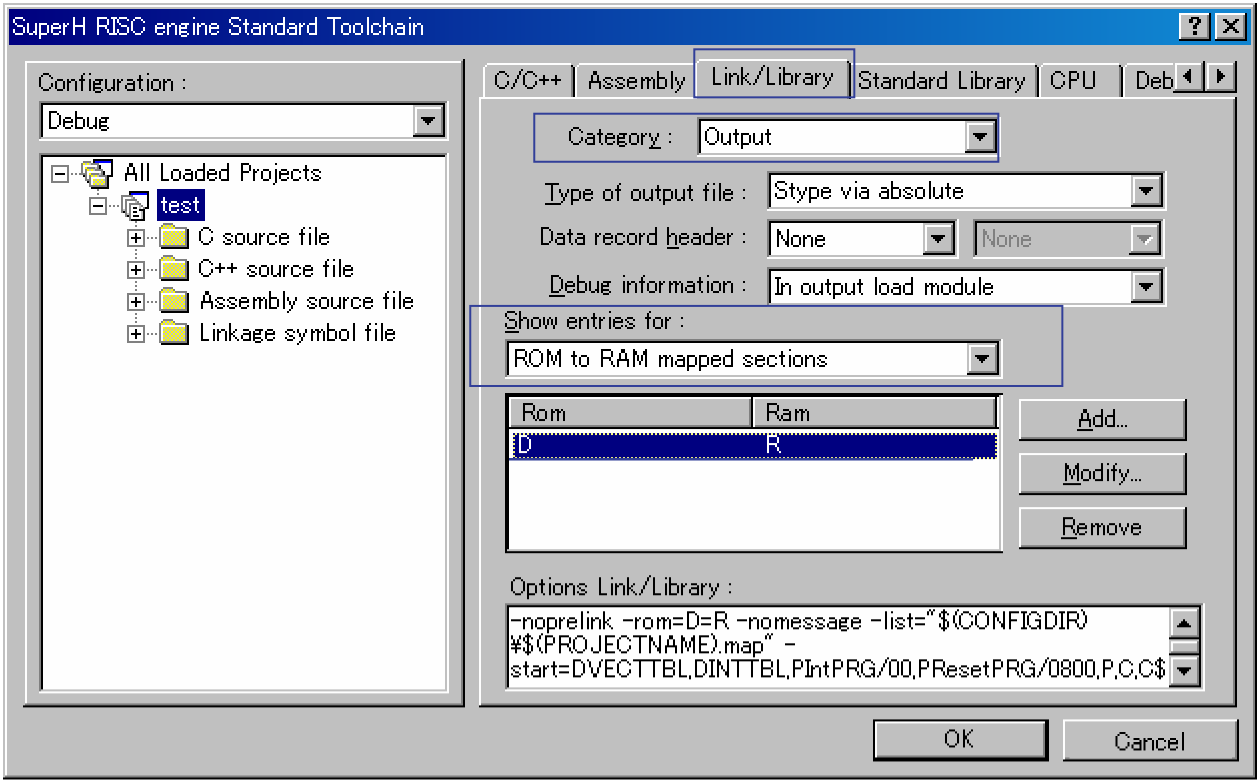
4.4 ROM Support Functionality

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**.



Configuration : Debug

Category : Output

Type of output file : Stype via absolute

Data record header : None None

Debug information : In output load module

Show entries for : ROM to RAM mapped sections

Rom	Ram
D	R

Options Link/Library :

```
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -
start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/0800,P,C,C$
```

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. Frequently Asked Questions

7.1 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.

7.2 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.

(c) 2007. Renesas Technology Corp., All rights reserved.