Supplementary explanation for HEW generation file

（SH-1, SH-2, SH-2E, SH-DSP）

1. Introduction

1.1 Contents of This Document

This document describes the description method of programming in embedded assembly language and C/C++ language. Supplementary explanation for HEW generation file is also included.

1.2 Targeted Compilers

This document targets compilers, which are newer than SuperH RISC engine C/C++ compiler Ver.7, and HEW3.0. This document contains “pragma” etc., which can not be used in the older versions. In addition, an electronic instruction manual (PDF file) is included in the compiler packages. Please refer to it for information on use of HEW, compilers, assemblers and optimizing linkage editor.

1.3 Notes

Please refer to a hardware manual and a programming manual attached to your microcomputer for information on contains of memory allocation and vector tables or use of the microcomputer. This document does not describe C/C++ language specification.

2. Programming in Assembly Language

2.1 Description of Reset Function

The following assembly program is minimum necessary to have SH-1/SH-2/SH-2E/SH-DSP microcomputers worked. Please set a stack pointer (R15 register) at 0xFFFFFFF0 just after power-on-reset, and jump at 0x00000800 where _PowerON_Reset_PC is set at (It is automatically performed by hardware). The processing is terminated with a SLEEP instruction. You can name the section as you want. In this document however, the sections are named PResetPRG and DVECTTBL in accordance with HEW generation files. You can also name the function as you want. In this document however, the function is named _PowerON_Reset_PC in accordance with HEW generation files. DVECTTBL is reset vector table, and it must be set at 0. PResetPRG section can be set anywhere in ROM region as long as it is not in the vector table area. In this document however, it is set at 0x00000800 in accordance with HEW generation files.

```
    .SECTION PResetPRG,CODE,LOCATE=H'00000800
_PowerON_Reset_PC:
    SLEEP

    .SECTION DVECTTBL,DATA,LOCATE=H'00000000
    .DATA.L    _PowerON_Reset_PC
    .DATA.L    H'FFFFFFF0

    .END
```

List 2-1

When manual reset function is added to the above program, it becomes as follows.

```
    .SECTION PResetPRG,CODE,LOCATE=H'00000800
_PowerON_Reset_PC:
    SLEEP
_Manual_Reset_PC:
    SLEEP

    .SECTION DVECTTBL,DATA,LOCATE=H'00000000
    .DATA.L    _PowerON_Reset_PC
    .DATA.L    H'FFFFFFF0
    .DATA.L    _Manual_Reset_PC
    .DATA.L    H'FFFFFFF0
    .END
```

List 1-2

2.2 Function Call

In the following assembly program, _main function is called from _PowerON_Reset_PC function after power-on-reset, and the processing is terminated with a SLEEP instruction after returning from the function. Descriptions of a _Manual_Reset_PC function and a vector table are omitted.

```
    .SECTION PResetPRG,CODE,LOCATE=H'00000800
_PowerON_Reset_PC:
    MOV.L    #_main, R0
    JSR      @R0
    NOP
    SLEEP

    .SECTION P,CODE,LOCATE=H'00001000
_main:
    RTS
    NOP
```

List 2-3

In case of returning from functions, you can use RTS instruction. However, it was not described in _PowerON_Reset_PC function above. It is because it never happens to return from reset function. In addition, it is required to set SLEEP instruction or loop them infinitely to prevent the program from running away. The following program is an example of infinite loop.

```
_PowerON_Reset_PC:
    MOV.L    #_main, R0
    JSR      @R0
    NOP
_loop:
    BRA      _loop
    NOP
```

List 2-4

Additionally, when calling function with JSR instruction, the returning address from the called function is stored in PR register (It is automatically performed by hardware). In case of returning from the

function call with RTS instruction, the processing which had been done before the function call is continued by restoring a value of PR register to PC register (It is also automatically performed by hardware). Because SH microcomputer has only one PR register, PR register needs to be saved temporarily and restored in case that the function call is nested with JSR instructions. In the following program, PR register is saved before a function call (a), and restored after the function call (b). Moreover, not only PR register, but also registers for general purpose need to be saved and restored as well.

```
    .SECTION PResetPRG,CODE,LOCATE=H'00000800
_PowerON_Reset_PC:
    MOV.L    #_main, R0
    JSR      @R0
    NOP
    SLEEP

    .SECTION P,CODE,LOCATE=H'00001000
_main:
    STS.L    PR, @-R15    ...(a)
    MOV.L    #_foo, R0
    JSR      @R0
    NOP
    LDS.L    @R15+, PR    ...(b)
    RTS
    NOP

_foo:
    RTS
    NOP
```

List 2-5

Because it never happens to return from _PowerON_Reset_PC function, PR register needs to be neither saved nor restored in _PowerON_Reset_PC function.

2.3 Description of Interrupt Function

You can describe interrupt function in assembly language as follows. In the following program, if an interruption by general illegal instruction exception occurs, _INT_Illegal_code function is called and the processing is terminated with a SLEEP instruction. Also, if an interruption by illegal slot instruction exception occurs, _INT_Illegal_slot function is called and a processing is terminated with a SLEEP instruction.

```
    .SECTION PIntPRG,CODE,LOCATE=H'00000400
_INT_Illegal_code:
    SLEEP

_INT_Illegal_slot:
    SLEEP

    .SECTION DINTTBL,DATA,LOCATE=H'00000010
    .DATA.L    _INT_Illegal_code
    .DATA.L    0
    .DATA.L    _INT_Illegal_slot
```

List 2-6

You can name the functions and the sections as you want here also. In this document however, the functions are named _INT_Illegal_code and _INT_Illegal_slot, and the sections are named PIntPRG and DINTTBL in accordance with HEW generation files. You can set PIntPRG section at anywhere in ROM region as long as it is not in the vector table area. In this document however, it is set at 0x00000400 in accordance with HEW generation files. DINTTBL is vector tables other than reset vector table, and set at 0x00000010. You can set the vector tables at an address other than 0x00000010 by setting an address of VBR register ([2.4 VBR Register]).

In the above program [List 2-6], if _main function is defined as follows, general illegal instruction exception occurs by (a) instruction, and _INT_Illegal_code function is called.

```
_main:
    .DATA.L 0    ...(a)
    RTS
    NOP
```

List 2-7

Also, if _main function is defined as follows, illegal slot instruction exception occurs by (b) instruction, and _INT_Illegal_slot function is called.

```
_main:
    RTS        ...(b)
    .DATA.W 0
```

List 2-8

[List 2-6] shows cases of terminating processing with SLEEP instructions after occurrence of interruptions. In some cases such as timer interruption, it may return from an interrupt function, and continue the processing, which had been done before the interruption. When returning from interrupt function, it must be returned with RTE instruction ([List 2-9]). Interrupt function is different from general functions, which can be returned with RTS instruction. When an interruption occurs, PC register and SR register are stored on the stack, and an interrupt function is called (It is automatically performed by hardware). In case of returning from interrupt function with RTE instruction, processing which had been done before the interruption is continued by taking the PC register and

the SR register back from the stack (It is also automatically performed by hardware).

```
_INT_xxx:
    RTE
    NOP
```

List 2-9

Registers other than PC register and SR register are neither saved nor restored at occurrence of an interruption. Therefore, values of registers, which have possibilities to be rewritten in interrupt function must be temporally saved on the stack before it is used in interrupt function. The values of registers also must be restored again before returning from the interrupt function with RTE instruction. In the following program, a value of R0 register is modified by (c) processing. If an interruption occurs between (a) and (b), it returns to (b) from interrupt function with the value of R0 register, which is modified in _INIT_xxx function. Therefore, the microcomputers do not work properly from that time.

```
_main:
    STS.L    PR, @-R15
    MOV.L    A_foo, R0      ...(a)
    JSR      @R0            ...(b)
    NOP
    LDS.L    @R15+, PR
    RTS
    NOP

_INT_xxx:    ; 任意の例外処理
    MOV.L    #0, R0         ...(c)
    RTE
    NOP
```

List 2-10

In order to avoid it, R0 register must be saved temporarily before（C）processing, and restored after (C) processing ((d),(e) of [List 2-11]).

```
_main:
    STS.L    PR, @-R15
    MOV.L    A_foo, R0
    JSR      @R0
    NOP
    LDS.L    @R15+, PR
    RTS
    NOP

_INT_xxx:    ; 任意の例外処理
    MOV.L    R0, @-R15      ...(d)
    MOV.L    #0, R0
    MOV.L    @R15+, R0      ...(e)
    RTE
    NOP
```

List 2-11

2.4 VBR register

In hardware specification on SH microcomputers, vector tables other than reset vector table are referenced using the offset from the value of VBR register. Therefore, you can also set the vector tables other than reset vector table at any address you want by setting VBR register at an address you want. For example, you will set VBR register as follows. However, an address of DINTTBL section needs to be set at xxxxxxxx.

```
_PowerON_Reset_PC:
    MOV.L    # H'xxxxxxxx, R0
    LDC      R0, VBR
    ...

    .SECTION DINTTBL,DATA,LOCATE=H'xxxxxxxx
    .DATA.L    0
    .DATA.L    0
    .DATA.L    0
    .DATA.L    0
    .DATA.L    _INT_Illegal_code
    .DATA.L    0
    .DATA.L    _INT_Illegal_slot
```

List 2-12

You can also set VBR register as follows.

```
_PowerON_Reset_PC:
    MOV.L    # H'xxxxxxxx - H'00000010, R0
    LDC      R0, VBR
    ...

    .SECTION DINTTBL,DATA,LOCATE=H'xxxxxxxx
    .DATA.L    _INT_Illegal_code
    .DATA.L    0
    .DATA.L    _INT_Illegal_slot
```

Figure 2-1

In this case, an address of vector tables other than reset vector table needs to be set at xxxxxxxx. Because VBR register needs to be set at a start address of the vector tables including reset vector table, 0x00000010 is subtracted from xxxxxxxx in the above program. You can also set VBR register as follows.

```
_PowerON_Reset_PC:
    MOV.L    #_INT_Vectors - H'00000010, R0
    LDC      R0, VBR
    ...

    .SECTION DINTTBL,DATA,LOCATE=H'xxxxxxxx
_INT_Vectors:
    .DATA.L    _INT_Illegal_code
    .DATA.L    0
    .DATA.L    _INT_Illegal_slot
```

List 2-13

By setting an address at xxxxxxxx, you can set VBR register at an address, which is subtracted 0x00000010 (reset vector table size) from it. In this case, when you change the vector table, you only need to modify one address (an address set for LOCATE) in the source file, so you can avoid mistakes at modifying.

3. Programming in C/C++ Language

3.1 Description of Reset Function

The preceding chapter described programming methods in assembly language. This chapter describes programming methods in C/C++ language. The following program written in C/C++ language is equivalent to [List-2-1].

```
#include <machine.h>                    ...(a)

#pragma entry PowerON_Reset_PC          ...(b)

#pragma section ResetPRG                ...(c)
void PowerON_Reset_PC(void)
{
    sleep();                            ...(d)
}
#pragma section VECTTBL                 ...(e)
void* RESET_Vectors[] = {
     (void*)PowerON_Reset_PC,
     (void*)0xFFFFFFF0
};
```

List 3-1

PowerON_Reset_PC is a reset function. It is generated in PResetPRG section using #pragma section ResetPRG statement (c). RESET_Vectors is a reset vector table. It is generated in DVECTTBL section using #pragma section VECTTBL statement (e). At memory allocation of optimizing linkage editor, DVECTTBL section needs to be set at 0 ([4.5 Memory allocation of optimizing linkage editor]). PResetPRG section is set at 0x00000800 in HEW generation setting. You can use built-in functions to describe SLEEP instruction in C/C++ language (d). When using the built-in functions, <machine.h> need to be included in a source file, which is in use (a). There are some built-in functions other than sleep function such as setting/getting of status register (set_cr / get_cr), setting/getting of interrupt mask (set_imask / get_imask), setting/getting of VBR register (set_vbr / get_vbr), nop function and so on. In addition, because it never happens to return from _PowerON_Reset_PC function, registers need to be neither saved before function calls nor restored after function calls. The save / restore codes of the registers can be suppressed by using #pragma entry statement (b). You can also describe [List-2-1] in C/C++ language as follows.

```
#include <machine.h>

#pragma entry PowerON_Reset_PC
#pragma stacksize 0x400              ...(f)

#pragma section ResetPRG
void PowerON_Reset_PC(void)
{
    sleep();
}
#pragma section VECTTBL
void* RESET_Vectors[] = {
    (void*)PowerON_Reset_PC,
    __secend("S")                   ...(g)
};
```

List 3-2

S section with 0x400 bytes area is generated by compilers using #pragma stacksize statement (f). S section is called a stack section. __second (g) is called a section address operator (it is not a function), and can be set at an address, which is added 1 to the end address of a section surrounded by quotes (“) . Then you can set S section in RAM region. At memory allocation of optimizing linkage editor, you need to set a top address of the section. Therefore, if you set the end address of the stack at 0xFFFFFFF0, you need to set the top address of the section at 0xFFFFFBF0 （0xFFFFFFF0 – 0x400）. When you add descriptions of manual reset function to it, the program becomes as follows.

```
#include <machine.h>

#pragma entry PowerON_Reset_PC
#pragma entry Manual_Reset_PC

#pragma section ResetPRG
#pragma stacksize 0x400

void PowerON_Reset_PC(void)
{
    sleep();
}

void Manual_Reset_PC(void)
{
    sleep();
}

#pragma section VECTTBL
void* RESET_Vectors[] = {
    (void*)PowerON_Reset_PC,
    __secend("S")
    (void*)Manual_Reset_PC,
    __secend("S")
};
```

List 3-3

3.2 Function Call

The following written in program in C/C++ language is equivalent to [List2-5]. Descriptions of a manual reset function and a reset vector table are omitted. Compilers automatically generate RTS instruction ([List 2-3]) and instructions for save / restore operations of the PR and the other registers ([List 2-5]). Therefore, you do not need to describe them by yourself.

```
#include <machine.h>

#pragma entry PowerON_Reset_PC
#pragma stacksize 0x400

void foo(void)
{
}

void main(void)
{
    foo();
}

#pragma section ResetPRG
void PowerON_Reset_PC(void)
{
    main();
    sleep();
}
```

List 3-4

foo function and main function are generated in P section. Here upon HEW generation setting, P section is set at 0x00001000.

3.3 Description of Interrupt Function

The following program written in C/C++ language is equivalent to [List 2-6].

```
#include <machine.h>

#pragma section IntPRG                ...(a)

#pragma interrupt INT_Illegal_code    ...(b)
void INT_Illegal_code(void)
{
    sleep();
}

#pragma interrupt INT_Illegal_slot    ...(c)
void INT_Illegal_slot(void)
{
    sleep();
}

#pragma interrupt INT_xxx             ...(d)
void INT_xxx(void)
{
}

#pragma section INTTBL                ...(e)
void *INT_Vectors[] =
{
    (void*) INT_Illegal_code,
    (void*) 0,
    (void*) INT_Illegal_slot
    ...
    (void*) INT_xxx
}
```

List 3-5

INT_Illegal_code function, INT_Illegal_slot function and INT_xxx function are generated in PIntPRG section by using #pragma section IntPRG statement (a). INT_Vectors is vector tables other than reset vector table. It is generated in DINTTBL section by using #pragma section INTTBL statement (e). Here upon HEW generation setting, PIntPRG section is set at 0x00000400, and DINTTBL section is set at 0x00000010. However, you can set the DINTTBL section at any address you want in accordance with setting of VBR address ([3.4 VBR Register]). INT_xxx function returns from the function by RTE instruction by using #pragma interrupt statement (The code is automatically generated by the compilers). Additionally, there are registers, which the values have possibilities to be rewritten in interrupt function. In such case, save / restore operation codes of the registers are generated by compilers. The operation codes of INT_Illegal_code function and INT_Illegal_slot function are generated to return from the functions with RTE instruction by using #pragma interrupt statement as (b) and (c) indicate above. In this case however, the processing is terminated with a SLEEP instruction because of a built-in function sleep.

3.4 VBR Register

VBR register can be set using built-in functions.

```
#include <machine.h>

extern void* INT_Vectors[];
void PowerON_Reset_PC(void)
{
    set_vbr((void*)((char*)INT_Vectors-0x00000010));
    ...
}
```

List 3-6

VBR register needs to be set at a top address of the reset vector table. Therefore, it needs to be set at an address, which is subtracted reset vector table size（0x00000010）from the top address (INT_Vectors) of vector tables other than reset vector table. By setting in this way, VBR register will automatically set at a proper address just by specifying an address of DINITTBL at memory allocation of optimizing linkage editor.

3.5 Non-initialized data area

Definition of external variable is generated in B section as non-initialized data area at default in case that the external variable has no initial value such as variable a as the following (a) indicates.

```
int a;                          ...(a)
void main(void)
{
    int b = a;                  ...(b)
    a = 1;                      ...(c)
}
```

List 3-7

Because the non-initialized data area has possibilities to be rewritten at executing programs as (c) indicates, it needs to be set in RAM region. In addition, external variables, which have no initial values must be set 0 in according to ANSI C/C++ language specification. For example, the value of variable a must be 0 when referring to (b). Therefore, B section area is required to clear with 0 at executing applications as follows. However, in case non-initialized data was properly initialized in the applications, It is not necessary to clear with 0.

```
void PowerON_Reset_PC
{
    char* p;

    p=(char*)__sectop("B");
    for(; p<(char*)__secend("B"); p++ )
        *p = 0;
    main();
    sleep();
}
```

List 3-8

Additionally, __sectop is a section address operator (it is not a function) as well as __second. It can be set at the top address of the section.

3.6 Constant Area

Definition of external variables is generated in C section as constant data area at default, in case that the external variable has the initial value, which is designated as constant, not as volatile such as variable b as the following (a) indicates. Character strings such as (b) is also generated in C section as constant data area at default.

```
int a;
const int b = 1;                ...(a)
void main(void)
{
    char* str = “string”;       ...(b)
    a = b;
}
```

List 3-9

Because constant area cannot be rewritten at executing programs, it is set in ROM region. Additionally, strings can be treated in the same way as [3.7 Initialization Data Area] by setting the (-string=data) compiler option.

3.7 Initialized Data Area

Definition of external variable is generated in D section as initialized data area at default, in case that the external variable has the initial value such as variable a as the following (a) indicates.

```
int a = 1;                          ...(a)
void main(void)
{
    a = 2;                          ...(b)
}
```

List 3-10

Because the initialized data area has a initial value, the initial data need to be set in ROM region. However, because the initialized data area has possibilities to be rewritten at executing programs as (b) indicates, the data, which are accessed to storage and take the value needs to be set at RAM region. What you can use here is ROM support function ([4.4 ROM support function]) included in optimizing linkage editor. By using this function, R section can be generated as a copy of D section. Where you are actually accessing to is R section hereat. D section is set in ROM region, and R section is set in RAM region at memory allocation of the optimizing linkage editor. R section can be named as you want. Additionally, variables with the initial values need to have the value by the time of execution of programs. Therefore, the data of D section needs to be copied to R section as follows.

```
void PowerON_Reset_PC(void)
{
    char *p, *q;

    p=(char*)__sectop("D");
    q=(char*)__sectop("R");
    for( ; p<(char*)__secend("D"); p++, q++ )
        *q = *p;

    main();
    sleep();
}
```

List 3-11

3.8 _INITSCT Function

[List 3-8] and [List 3-11] described about the initialization of B section and copying of D section to R section by user programs. However, you do not need to describe the initialized routines by using _INITSCT function included in standard libraries. You can use _INITSCT function as follows.

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

void PowerON_Reset_PC(void)
{
    _INITSCT();                                          ...(b)

    main();

    sleep();
}

#pragma section $DSEC                                    ...(c)
static const struct {                                    ...(d)
    char *rom_s;
    char *rom_e;
    char *ram_s;
}DTBL[]= {
    {__sectop("D"),__secend("D"), __sectop("R")},
};

#pragma section $BSEC                                    ...(e)
static const struct {                                    ...(f)
    char *b_s;
    char *b_e;
}BTBL[]= {
    {__sectop("B"), __secend("B")},
};
```

List 3-12

By just calling _INITSCT function, initialization of B section and copying of D section to R section are performed. In case of using _INITSCT function, please include <_h_c_lib.h> as (a) indicates, and link the standard library([4.6 Standard library]). Structure variable (d) is generated in C$DSEC section, and (f) is generated in C$BSEC section by using #pragma section statement as(c) and (e) indicate. They need to be set in ROM region. In case that there are any initialization data areas other than D section, please add them in DTBL as follows. Moreover, R1 section needs to be generated as a copy area of the initialization data area by ROM conversion feature. D1 section and R1 section must be set in ROM region and in RAM region each.

```
#pragma section 1
int a = 0;                                  ...D1 セクション

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

List 3-13

In case that there are any non-initialized data areas other than B section, please add them in BTBL as follows.　Additionally, B1 section must be set in RAM region.

```
#pragma section 1
int a;                                      ...B1 セクション

#pragma section $BSEC
static const struct {
    char *b_s;
    char *b_e;
}BTBL[]= {
    {__sectop("B"), __secend("B")},
    {__sectop("B1"), __secend("B1")},
};
```

List 3-14

3.9 Global class object (at programming in C++ language)

In case of developing application software in C++ language, globally declared class objects (global class objects) may exist. (a) and (b) in the following source program are the 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 3-15

In case that this class has a constructor, it needs to be called before accessing to any member of the class. Please look at the following program in C++ language. In this case, (c) needs to be performed, and member variable a as (d) indicates needs to be initialized to 1 before an execution of (e). Briefly, the constructor(c) must be called by the time of an execution of (e).

```
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 3-16

_CALL_INIT function is prepared as a standard library for the constructor call. _CALL_END function is also prepared to call destructor of the global class object. _CALL_INIT function and _CALL_END function are declared in <_h_c_lib.h>. Please include <_h_c_lib.h> in the source file which is in use (f). You can call _CALL_INIT function before initiation of an application (g), and call _CALL_END function after termination of the application (h).

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

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

    main();

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

List 3-17

Additionally, information for calling constructor and destructor is generated in C$INIT section. This section is automatically generated by compilers. Please set C$INIT section in ROM region at memory allocation of optimizing linkage editor.

3.10 Runtime Function and Standard Library

Registers and CPU instructions vary depending on CPU. For example, SH-2E microcomputer has a floating-point register and a floating-point unit. Therefore, it is possible carrying out floating-point operations with CPU instructions relatively small number of times by using the floating-point instruction. On the contrary, SH-1, SH-2, SH-DSP microcomputers have neither registers nor instructions for the floating-point operations. Therefore, in order to carry out the floating-point operations, dozens to hundreds of CPU instructions need to be executed. It can be said for not only in case of floating-point operation, but also in case of shift operation and four rules. Runtime function is useful in a time like this. Runtime function is usually used at developing applications in C/C++ language. It is included in the standard library. In order to use the standard library functions, the standard library must be linked([4.6 Standard Library]).

The following programs show codes generation at carrying out the floating-point operations.

```
float a,b,c;
void main()
{
    c = a + b;
}
```

List 3-18

The following program shows a code generation in case of using SH-2E.

```
_main:
        MOV.L       #_a,R2
        FMOV.S      @R2,FR8
        MOV.L       #_b,R2
        FMOV.S      @R2,FR9
        MOV.L       #_c,R2
        FADD        FR9,FR8
        RTS
        FMOV.S      FR8,@R2
```

List 3-19

The following program shows a code generation in case of using SH-2. __adds is called. It is a runtime function in the standard libraries.

```
_main:
        STS.L       PR,@-R15
        MOV.L       #_a,R6
        MOV.L       #_b,R5
        MOV.L       @R6,R1
        MOV.L       #__adds,R6
        JSR         @R6
        MOV.L       @R5,R0
        MOV.L       #_c,R6
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R6
```

List 3-20

3.11 Low-level Interface Routine

At developing application software in C/C++ language, you might use functions such as standard input / output libraries (fopen, printf, scanf etc.) and memory management libraries (malloc, free, new, delete). However, the compilers do not offer all these features. For example, there are many possible output directions such as LCD, HDD, printers and CD-R/RW etc. There also are many possible input directions such as DIP switches, keyboards, mouse devices, buttons on cellular phone, touchscreens etc. Needless to say, those devices require to be operated all differently. Therefore, it is impossible offering all the features of the standard input / output libraries and the memory management libraries by compilers. Consequently, a function constellation is called from the standard input / output libraries and the memory management libraries. This is called low-level interface routine. It absorbs the difference in operation among those devices. It needs to be developed by user. There are open, close, read, write, lseek, sbrk, errno_addr, wait_sem, signal_sem as the low-level interface routines. Please refer to the compiler user's manual for information on how to develop those routines. Additionally, the functions such as errno_addr, wait_sem, siglan_sem are needed in case of using reentrant standard libraries (in case of specifying –reent at constructing of the standard libraries). Moreover, libraries, which perform termination of programs such as exit, atexit, abort are also required to be developed by user as well.

4. Exposition of HEW Generation Files

4.1 Setting of Project Generator

This document describes generation files, which are generated by project generator according to the following procedures. The project generator is started from HEW menu [File → New Workspace]. Please also refer to a tutorial of the online manual (PDF file), which is included in the SH C/C++ compiler package for information on use of the project generator.

(1) Creation of new workspace

Here is in case of selecting [Application] as project .

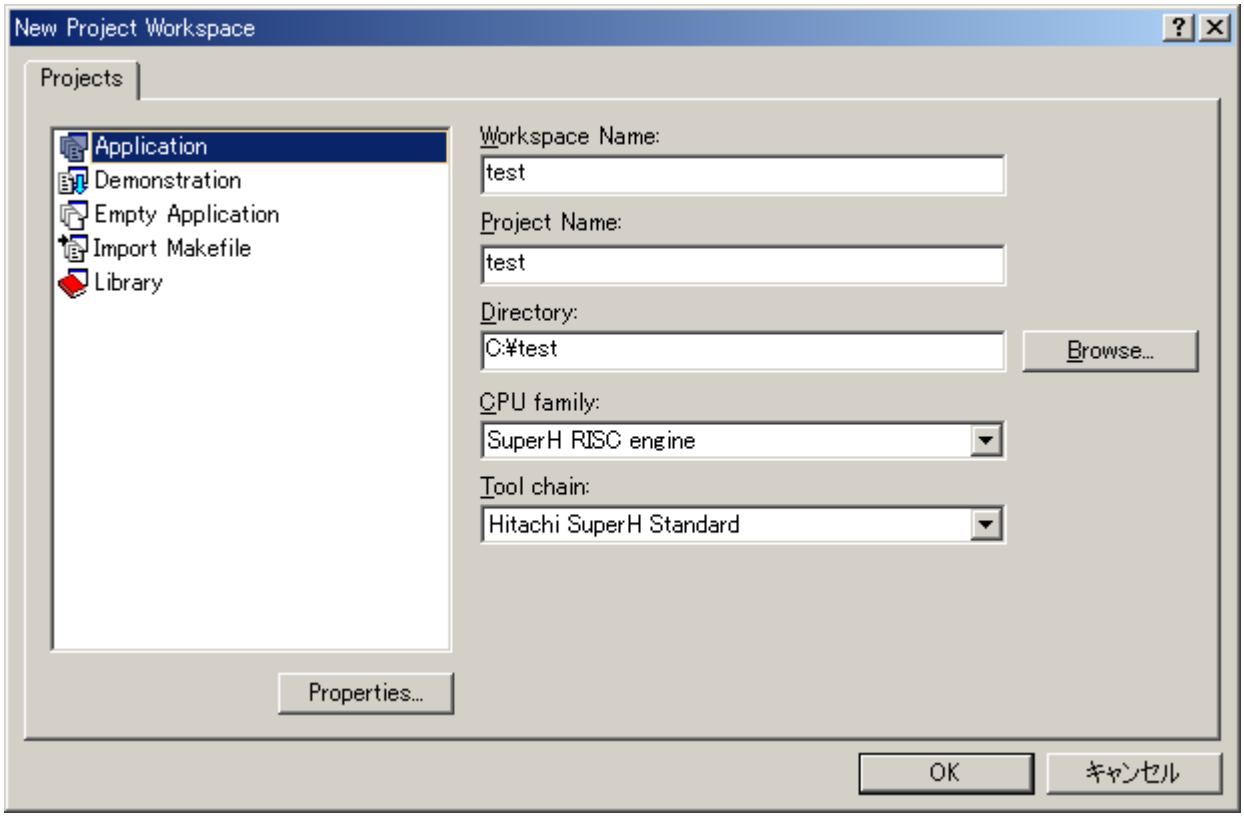

Figure 4-1

In case of selecting any other projects on this dialog box, it may be impossible choosing options such as checkboxes, which are described in later chapters.

(2) CPU choice

Here is in case of selecting [SH-2] as [CPU Series], and [SH7046] as [CPU Type]. Even if you select any other CPU Series（SH-1, SH2-DSP, SH-2E）or CPU Types, the basic generation files are the same.

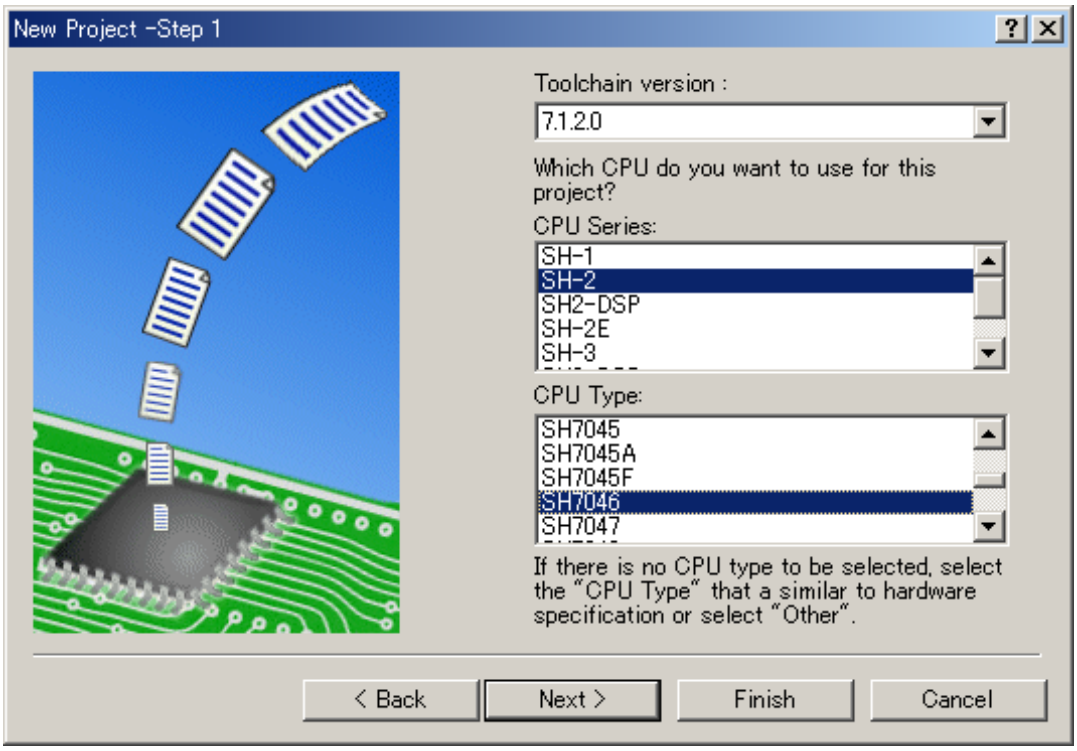

Figure 4-2

- [CPU Series] choice is reflected in CPU option ([4.3 CPU Option]).
- [CPU Type] choice is reflected in the contents of description in iodefine.h file and memory allocation of the optimizing linkage editor. If there is no CPU type to be selected, select the “CPU type” that a similar to hardware specification or select “other”.

(3) Option setting

Here is in case of modifying nothing of options on the dialog box, and just go to the next.

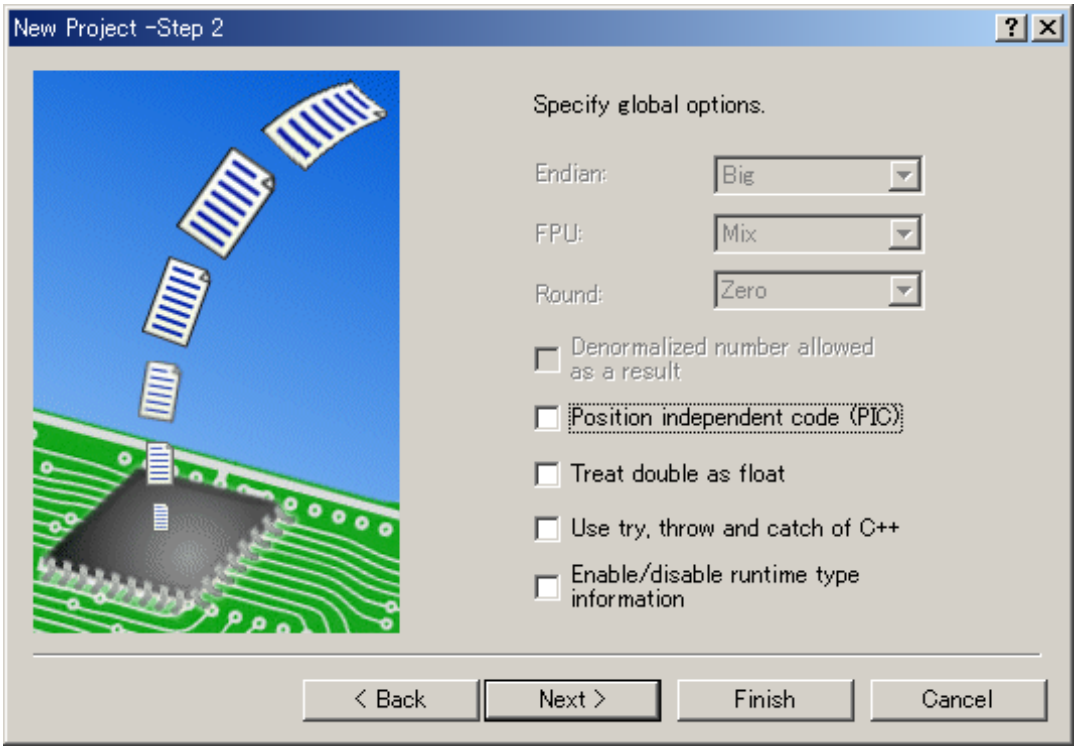

Figure 4-3

- CPU choices are all reflected in options on this dialog box. They vary depending on CPUs, which is selected at setting of [(2) CPU choice].

(4) Setting of generation files

Here is in case of checking [Use I/O Library] checkbox, setting “20” as [Number of I/O Streams], and selecting [C/C++ source file] as [Generate Hardware Setup Function].

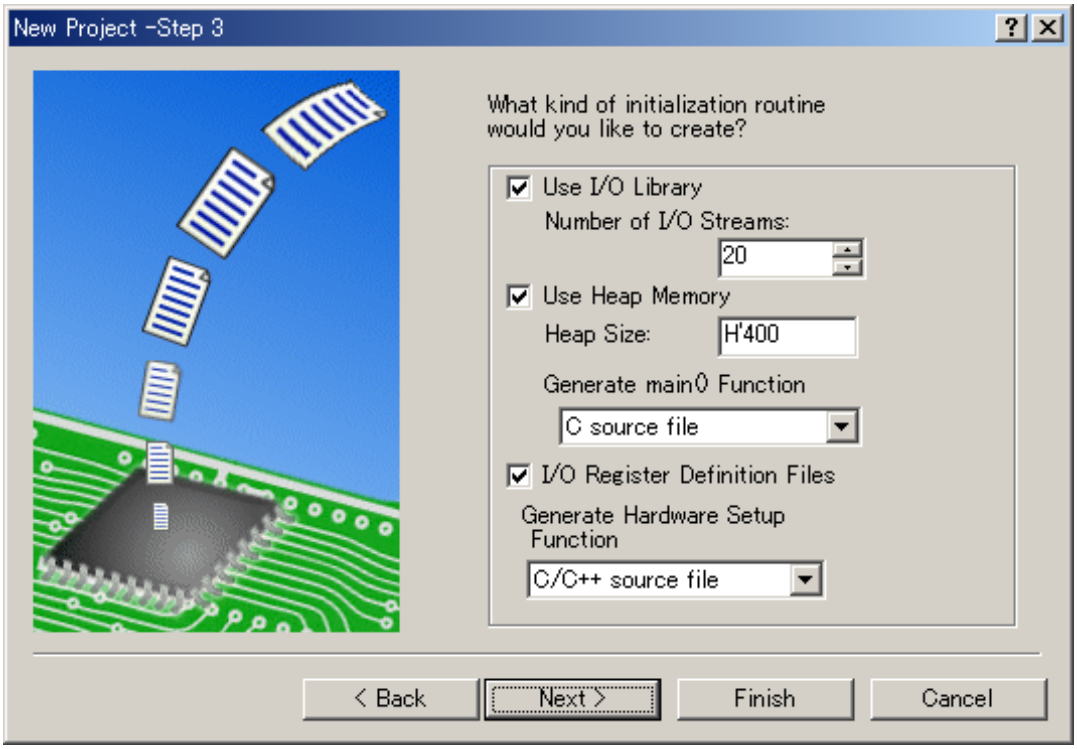

Figure 4-4

- By checking [Use I/O Library] checkbox, lowlvl.src, lowsrc.c, lowsrc.h are generated. A sample program of input / output-related low-level interface routines（open, close, write, read, lseek）and an initialized program of standard libraries（_INIT_IOLIB, _CLOSEALL）are described in them. If you do not use input / output–related standard libraries, please clear the checkbox, or delete lowlvl.src, lowsrc.c and lowsrc.h from HEW project after the workspace construction. The setting of [Number of I/O Stream] is reflected in lowsrc.h.
- By checking [Use Heap Memory] checkbox, sbrk.h and sbrk.c are generated. A sample program of memory management-related low-level interface routine (sbrk) is described in them. If you do not use memory management-related standard libraries, please clear the checkbox, or delete sbrk.h and sbrk.c from HEW project after the workspace construction. The setting of [Heap Size] is reflected in sbrk.h.
- By specifying [Generate main() Function], main function (C source file or C++ source source file) and low-level interface routine (abort) are generated. By checking [I/O Register Definition Files] checkbox, iodefine.h is generated.
- By specifying [Generate Hardware Setup Function], hwsetup.c, hwsetup.cpp or hwsetup.src are generated.

(5) Setting of standard library

Here is in case of modifying nothing of options on the dialog, and just go to the next.

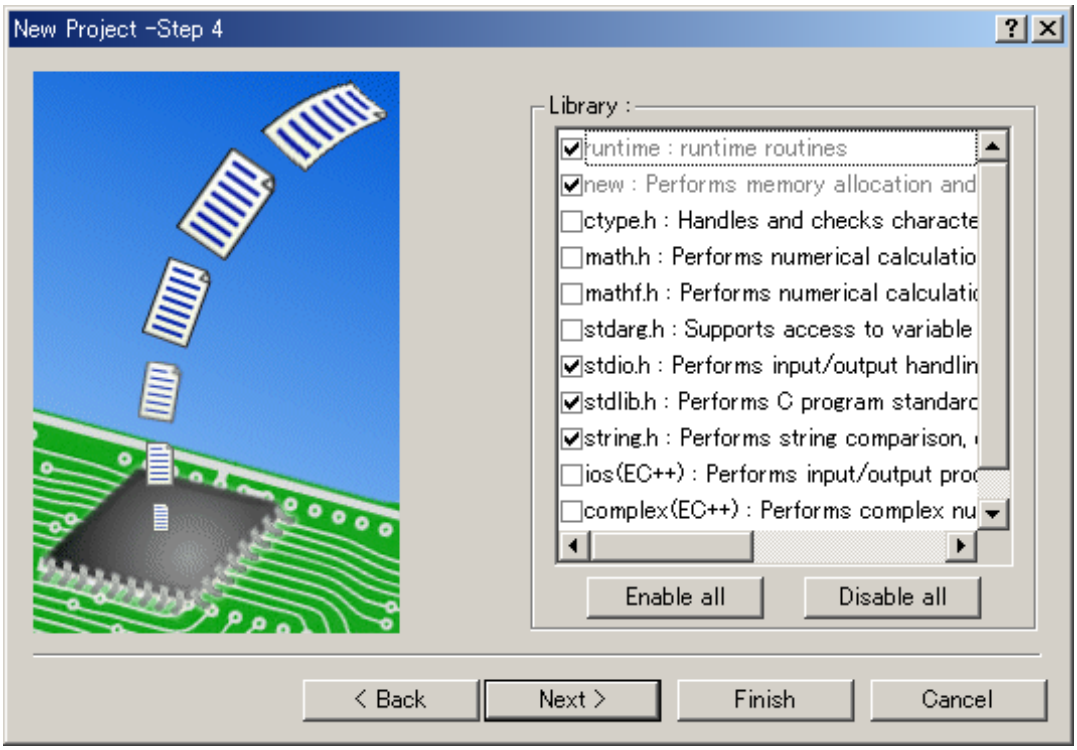

Figure 4-5

- The options you select on the dialog are reflected in setting of standard library.

(6) Setting of stack area

Here is in case of modifying nothing of options on the dialog, and just go to the next.

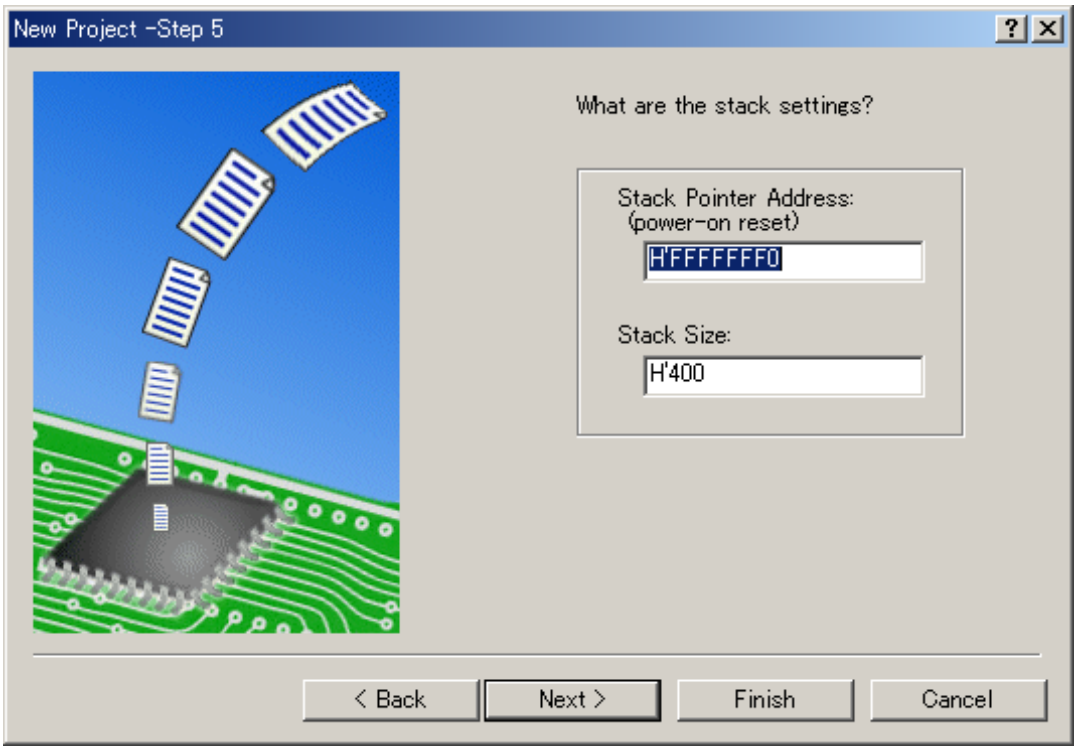

Figure 4-6

- Setting of [Stack Pointer Address] is reflected in memory allocation of optimizing linkage editor. Setting of [Stack Size] is reflected in stacksct.h.
- However, in case of clearing the checkbox for [Vector Definition Files] at [(7) Vector setting], stacksct.h is not generated.

(7) Vector setting

Here is in case of modifying nothing of options on the dialog, and just go to the next.

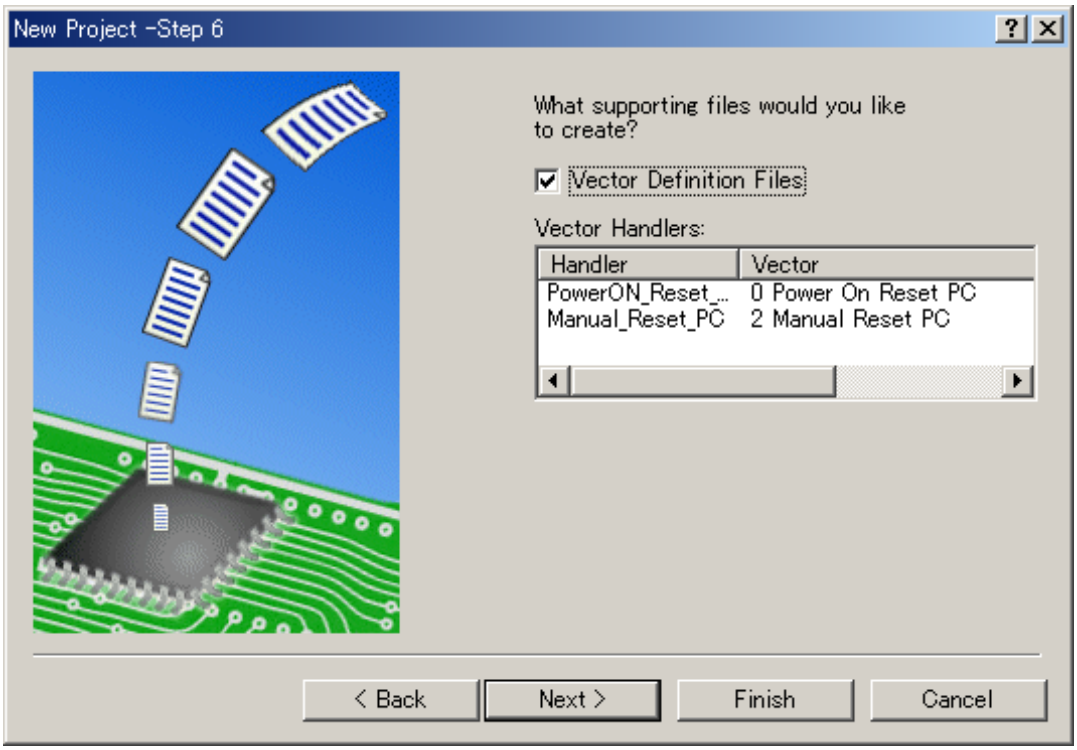

Figure 4-7

- By checking [Vector Definition Files] checkbox, intprg.c, resetprg.c, stacksct.h, vect.c and vect.h are generated.

(8) Specification of debugger targets

Here is in case of modifying nothing of options on the dialog, and just go to the next.

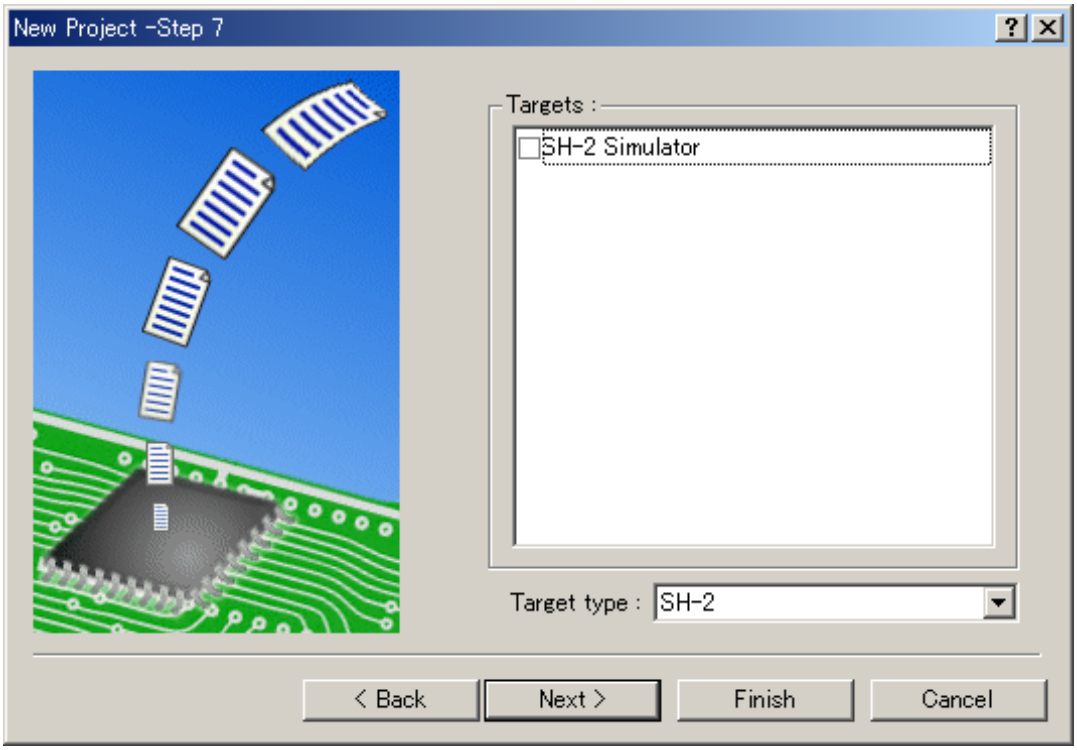

Figure 4-8

(9) Modification of generated file names

Here is in case of modifying nothing of options on the dialog, and just go to the next.

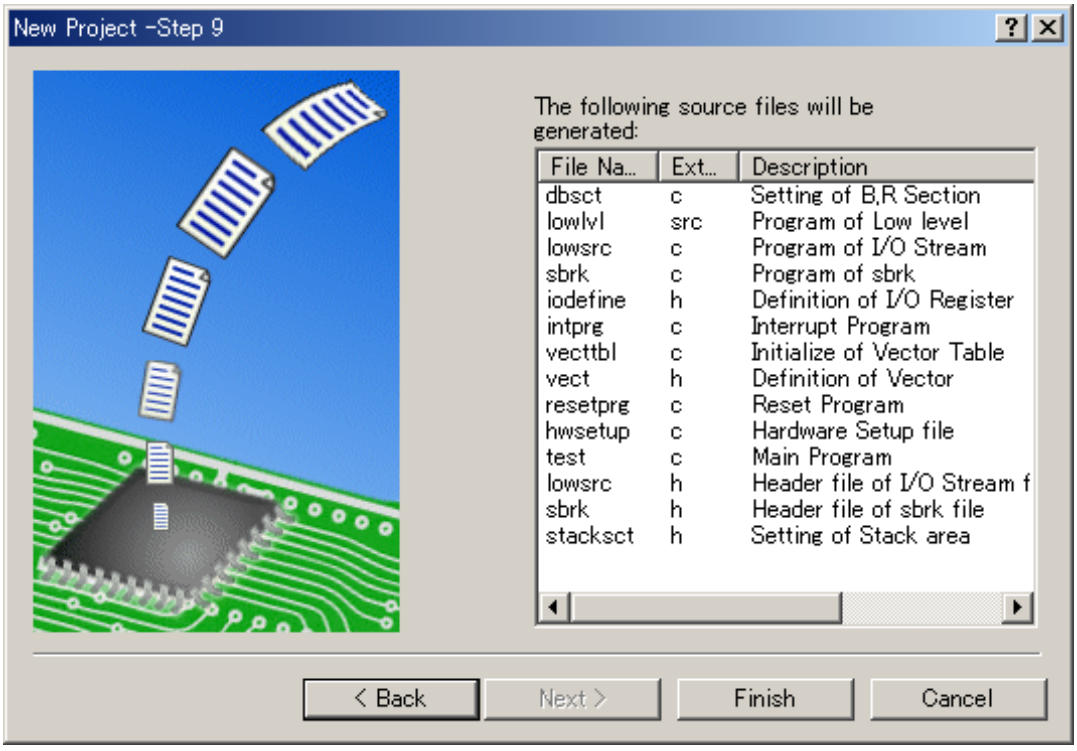

Figure 4-9

4.2 List of Generated Folders and Files

The following list shows the folder constructions and generated files, which are generated according to [4.1 Setting of Project Generator].

C:¥test		A workspace folder The folder name is which specified at (1) [Directory].
	test.hws	A workspace file The file name is which specified at (1) [Workspace Name].
	¥test	A project folder The folder name is which specified at (1) [Project Name].
	dbsct.c	This file is inevitably generated.
	hwsetup.c (hwsetup.src,hwsetup.cpp)	A file generated by specifying (4)[Generate Hardware Setup Function]
	intprg.c	A file generated by checking (7)[Vector Definition Files] checkbox.
	iodefine.h	A file generated by checking (4)[I/O Register Definition Files] checkbox.
	lowlvl.src	A file generated by checking (4)[Use I/O Library] checkbox.
	lowsrc.c	A file generated by checking (4)[Use I/O Library] checkbox.
	lowsrc.h	A file generated by checking (4)[Use I/O Library] checkbox. Setting of (4)[Number of I/O Stream] is reflected in this file.
	resetprg.c	A file generated by checking (7)[Vector Definition Files] checkbox.
	sbrk.c	A file generated by checking (4)[Use Heap Memory] checkbox.
	sbrk.h	A file generated by checking (4)[Use Heap Memory] checkbox. Reflects (4)[Heap Size].
	stacksct.h	A file generated by checking (7)[Vector Definition Files] checkbox. Reflects (6)[Stack Size]
	test.c (test.cpp)	A file generated by specifying (4)[Generate main() Function] The file name is one, which is specified at (1) [Project Name].
	test.hwp	A HEW project file The file name is which specified at (1) [Project Name].
	vect.h	A file generated by checking (7)[Vector Definition Files] checkbox.
	vecttbl.c	A file generated by checking (7)[Vector Definition Files] checkbox.
	¥Debug	Intermediate files and absolute file are generated in this folder at specifying Debug configuration.
	¥Release	Intermediate files and absolute file are generated in this folder at specifying Release configuration.

Figure 4-1

4.3 CPU Option

From HEW menu：[Options → SuperH RISC engine Standard Toolchain]

→select [CPU] tab

The following dialog box is displayed.

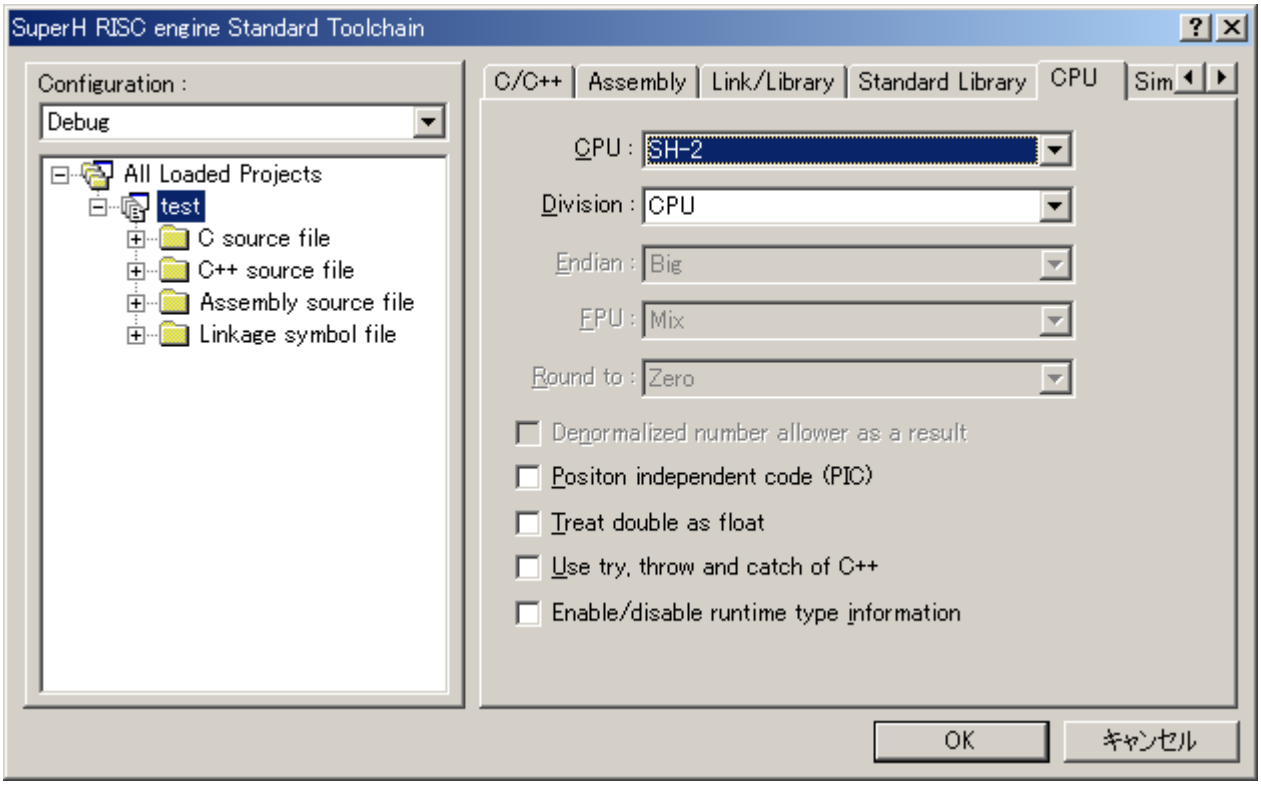

Figure 4-10

This dialog box reflect [(2) CPU choice] and [(3) Option setting].

4.4 ROM support function

From HEW menu：[Options → SuperH RISC engine Standard Toolchain]

→select [Link/Library] tab

→ select ”Output” as [Category]

→select ”ROM to RAM mapped sections” as [Show entries for]

The following dialog box is displayed.

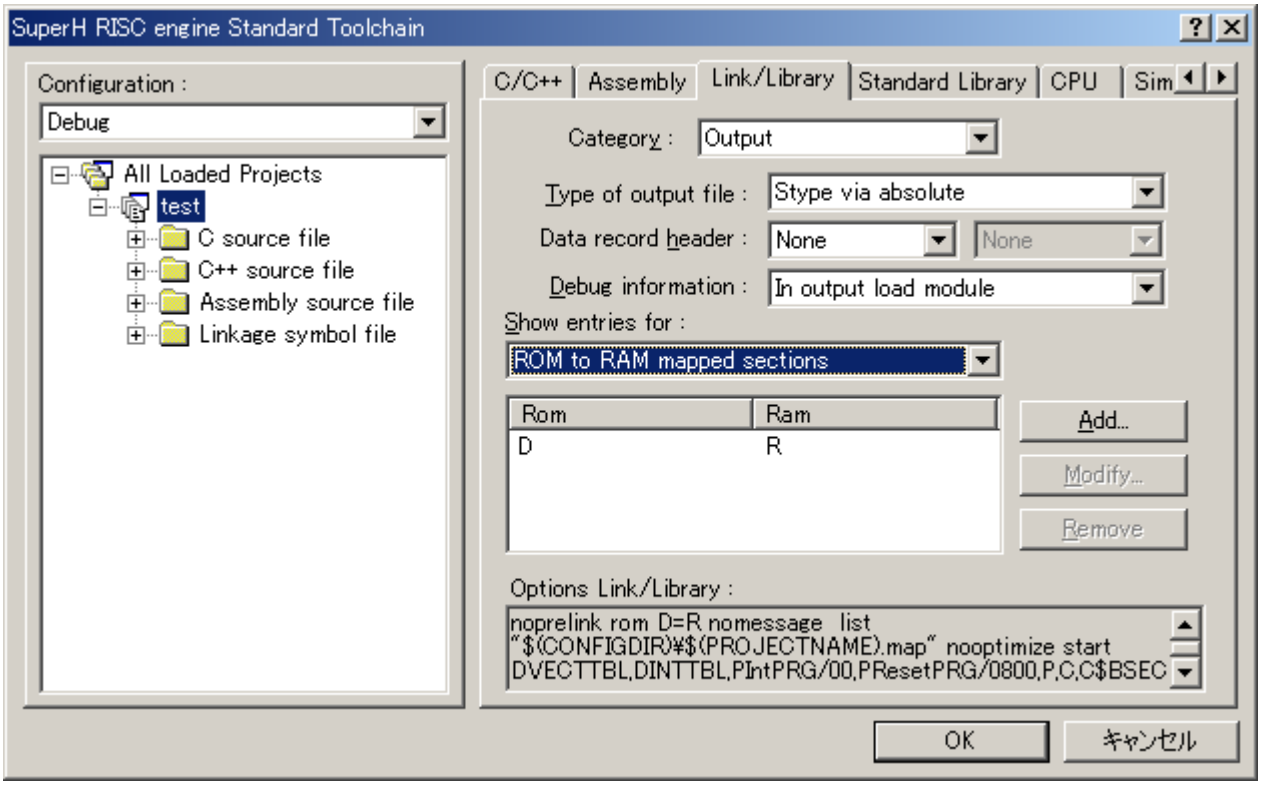

Figure 4-11

Please modify the setting on the dialog in case of using ROM support function of optimizing linkage editor.

4.5 Memory Allocation

From HEW menu：[Options → SuperH RISC engine Standard Toolchain]

→select [Link/Library] tab

→select "Section" as [Category]

→ select "Section" as [Show entries for]

The following dialog box is displayed.

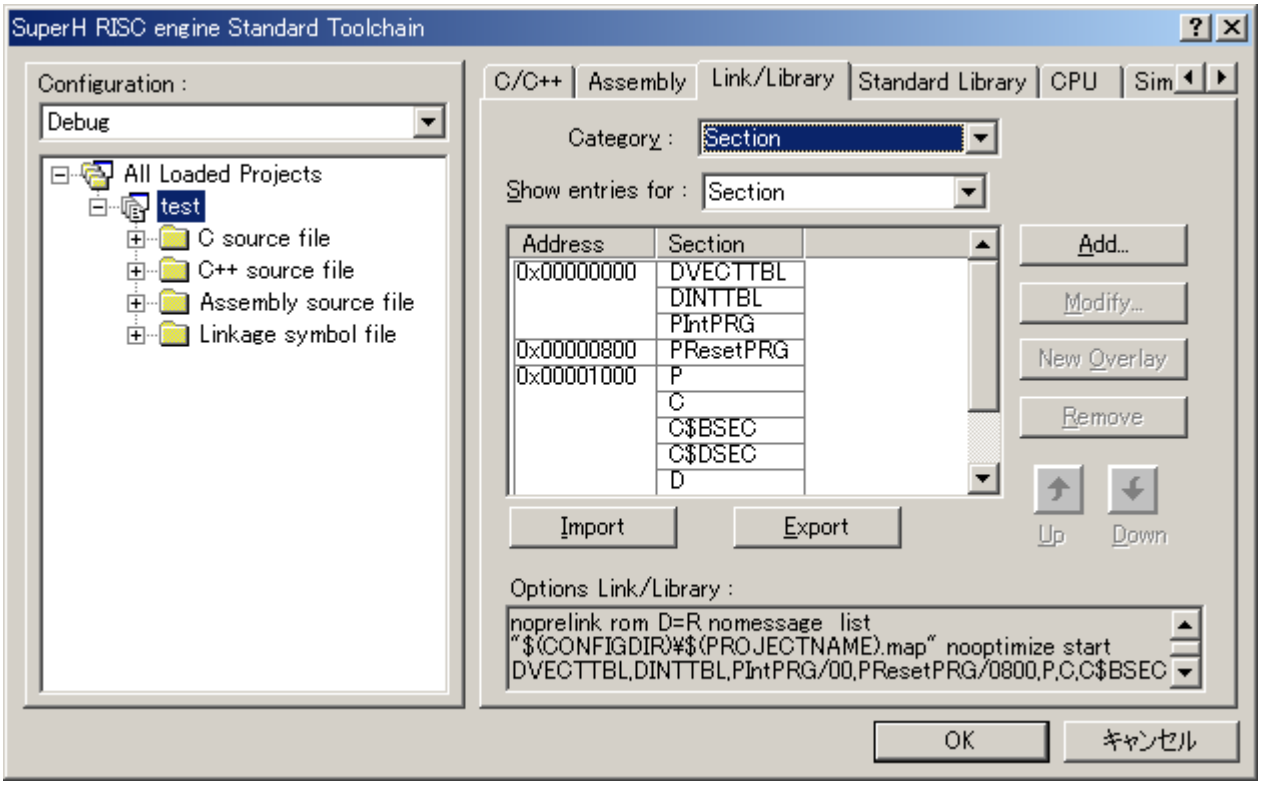

Figure 4-12

In case of selecting SH7046 as CPU Type, the sections are addressed as follows. However, C$INIT is addressed as follows in case that main function is generated in C++ language. Please modify the addresses properly on the above dialog box.

Address	Section name	Explanation
0x00000000	DVECTTBL	Reset vector table
	DINTTBL	Vector tables other than reset
	PIntPRG	Interrupt function
0x00000800	PResetPRG	Reset function
0x00001000	P	Program area
	C	Constant area
	C$BSEC	For _INITECT function
	C$DSEC	For _INITECT function
	C$INIT	For _CALL_INIT
	D	Initialized data area（ROM）
0xFFFFF000	B	Non-initialized data area
	R	Initialized data area（RAM）
0xFFFFFBF0	S	Stack section

List 4-2

- An error message [L1120 (W) Section address is not assigned to “xxx”] might be displayed at building. It shows you that you have omitted setting an address for xxx section. Please set “xxx” section at designated address on this dialog.
- An error message [L1100 (W) Cannot find “xxx” specified in option “start”] might be displayed at building. It shows you that you have specified xxx section unnecessarily on this dialog. Please drop xxx section out of the specification the dialog. C section is automatically addressed by HEW however, the section may not exist depending on the generated files. In this case, L1100 is output instead.

4.6 Standard Library

(1) Generation options

From HEW menu：[Options → SuperH RISC engine Standard Toolchain]

→select [Standard Library] tab

→ select ”Mode” as [Category]

The following dialog box is displayed.

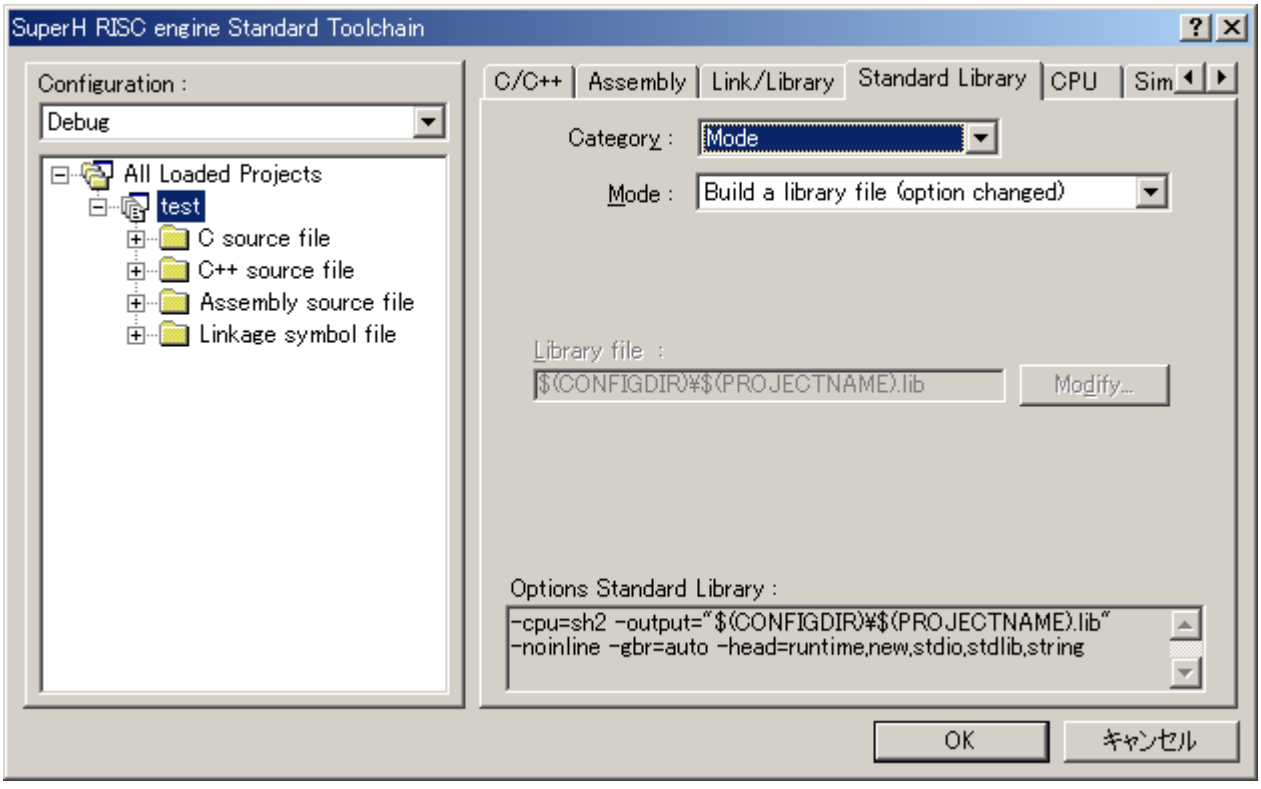

Figure 4-13

In order to link the standard library, please select either ”Build a library file (option changed)” or ”Build a library file (anytime)” as [Mode].

(2) Targeted header files

From HEW menu：[Options → SuperH RISC engine Standard Toolchain]

→select [Standard Library] tab

→select "Standard Library" as [Category]

The following dialog box is displayed.

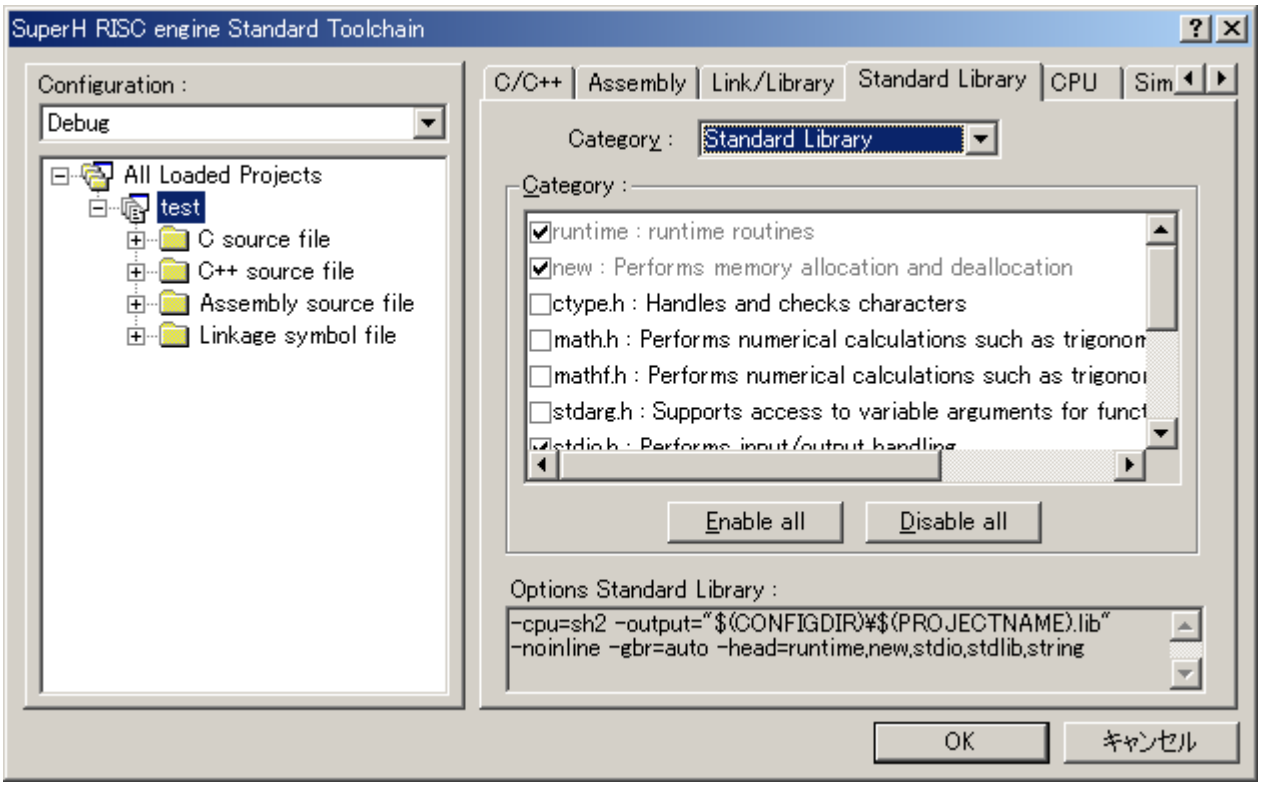

Figure 4-14

Please check the checkboxes of the standard library you want to use on a header file basis. However, a description to include them in the source file is still required. Please include the header files in the source file, which the standard library function is used.

4.7 Configuration

Options of compiler, assembler and optimized linkage editor are saved on projects on a unit basis called “configuration”. By switching the configurations, programs are built with these options.

Figure 4-15

Configurations for Debug, Release and debug session are generated by the project generator. However, the configuration for debug session is generated in case of specifying the debugger targets at (8) Specification of debugger targets in [4.1 Project generator setting]. Please use the configuration for debug session if the setting of the compiler needs to be modified in accordance with the debugger targets.

Setting of these configurations generated by HEW are all the same, however, only Release configuration is set not to output the debug information. Because the debug information is additional information required at debugging by a debugger, the HEW output codes between Debug and Release configurations are the same. At modifying the setting of the compiler, a configuration in use is targeted. For example, if only the setting of the Debug configuration was modified, the codes between Debug and the other configurations would be different. It is possible modifying options of multiple configurations at the same time by specifying “All configurations” as the target at modifying the options of the compiler, the assembler etc. on the following dialog box.

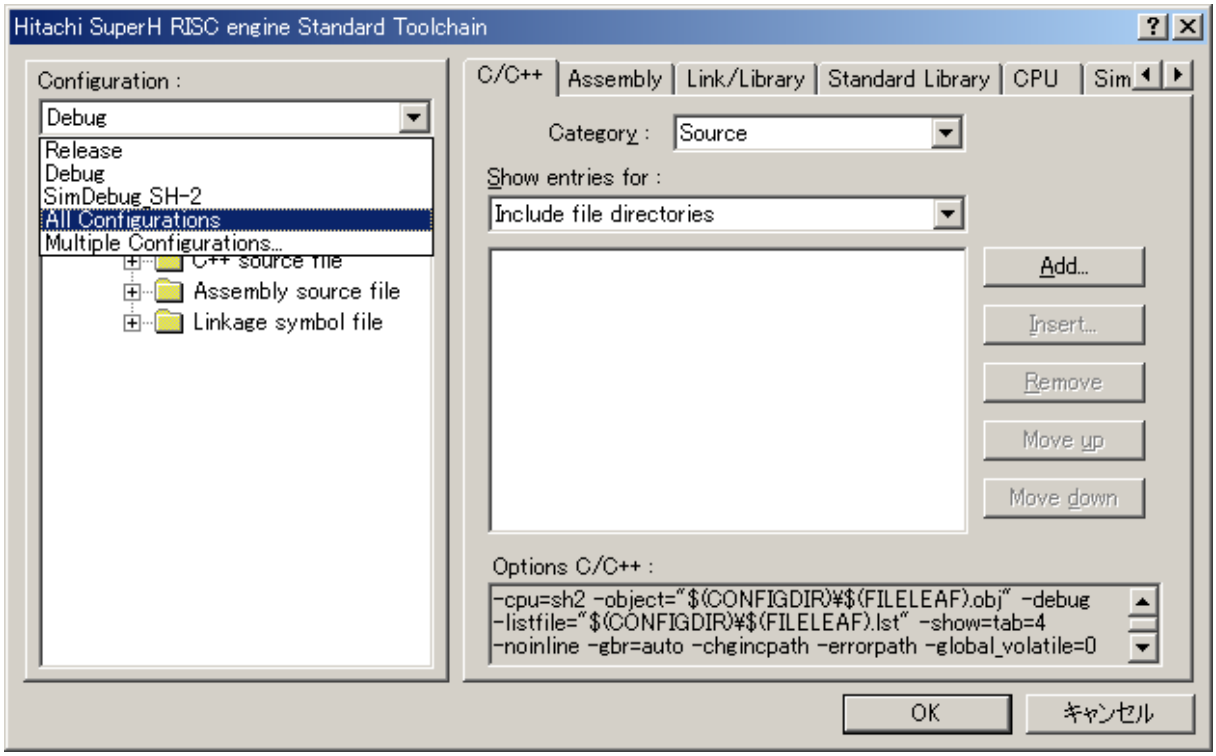

Figure 4-16

4.8 Project Configuration Editing

From HEW menu：[Project → Edit Project Configuration]

You can modify the options of files generated by HEW on the following dialog box.

(1) I/O Register

By checking [I/O Register Definition Files (Overwrite)] checkbox on [I/O Register] tab, and clicking [OK], iodefine.h is generated. You can use this feature in case that you have not checked [I/O Register Definition Files] checkbox at (4) Setting of generation files in [4.1 Setting of Project Generator], or in case that you want to restore iodefine.h.

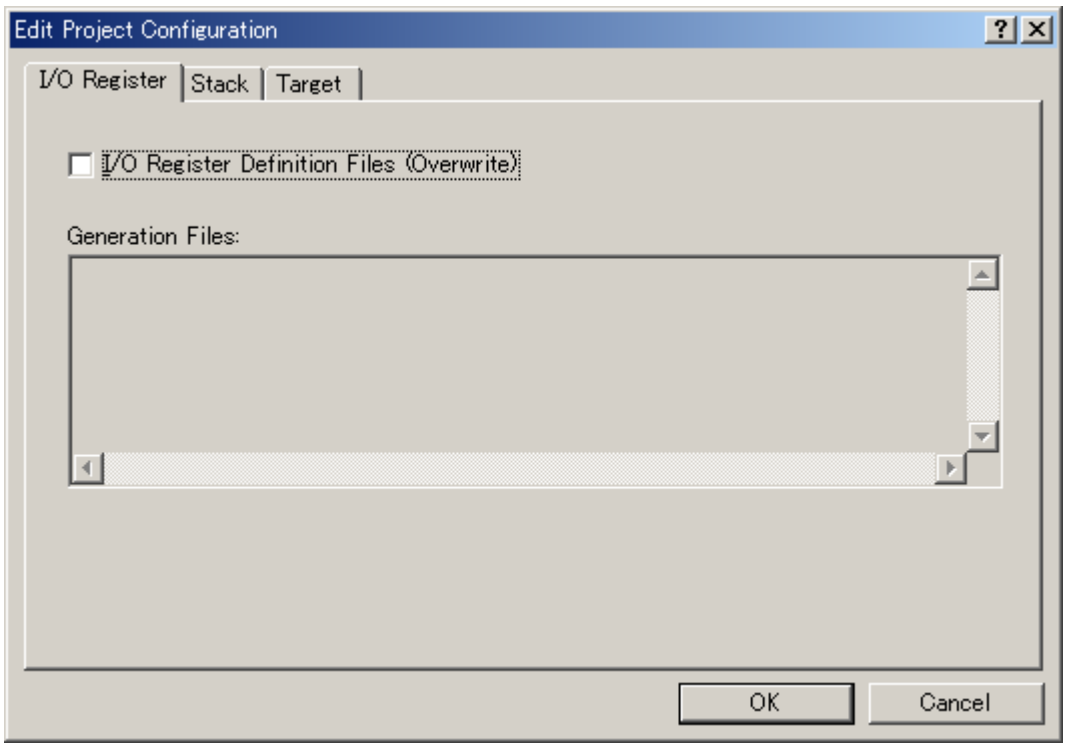

Figure 4-17

(2) Stack

You can set a stack area on the [Stack] tab. You can use this feature in case that you want to modify the setting of (6) Setting of stack area in [4.1 Setting of Project Generator]. These values are reflected in memory allocation of the optimized linkage editor and stacksct.h. However, this function can not be used in case that you have modified the address of S section at memory allocation of optimized linkage editor, or in case that you have modified the #pragma stacksize statement of stacksct.h.

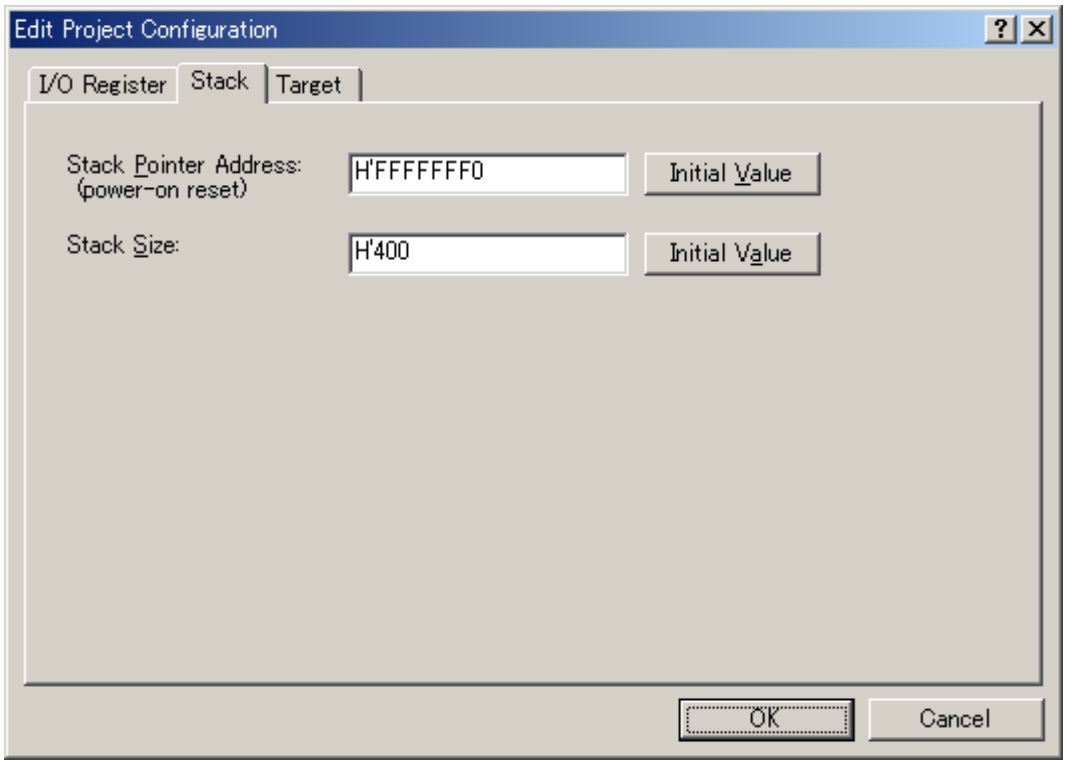

Figure 4-18

(3) Target

You can set debugger targets on [Target] tab. You can use this feature in case that you have not specified the targets at (8) Specification of debugger targets in [4.1 Setting of Project Generator].

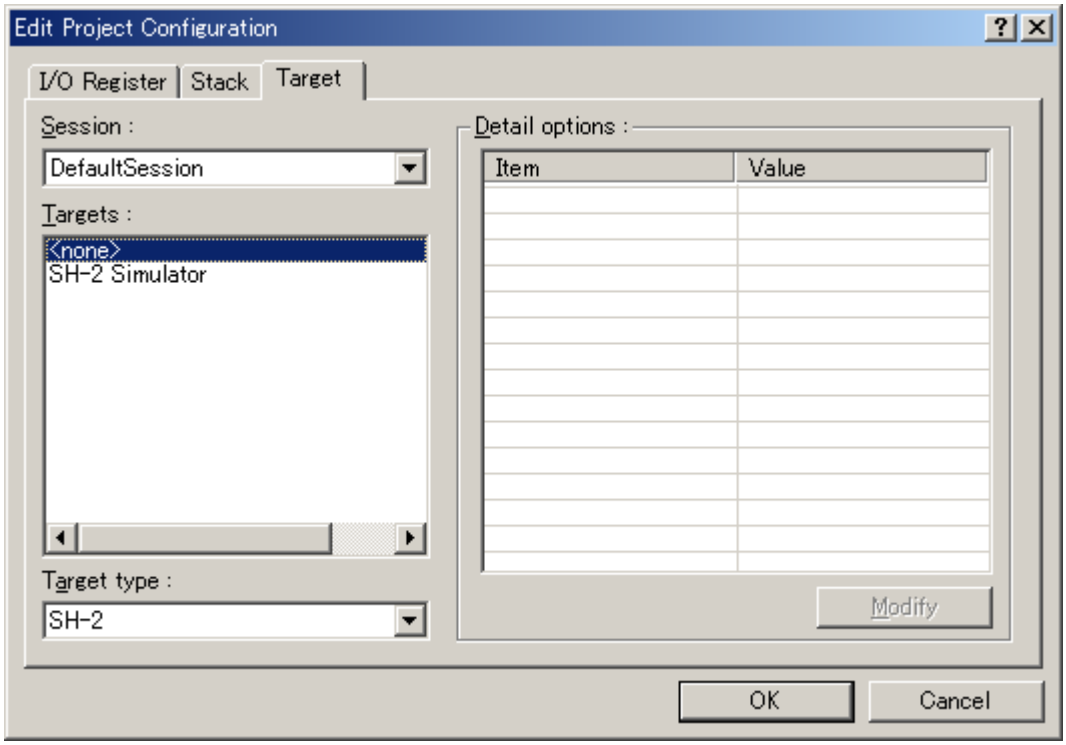

Figure 4-19

4.9 Low-level Interface Routines

(1) Memory management low-level interface routines

Sbrk function is described in sbrk.c as a low-level interface routine of memory management library. Please modify it properly according to the user system. Sbrk function is called when either malloc or new is called. You can delete the memory management library from the project, if you do not use this file. In addition, the heap size can be modified by setting the value of _sbrk_size. HEAPSIZE macro, which specifies the entire size of heap area is defined in sbrk.h. Please modify the HEAPSIZE macro if you want to modify the entire size of the heap area.

(2) Input / output low-level interface routines

Input / output low-level interface routine is described in lowlvl.src, lowsrc.c and lowsrc.h. Please modify it according to the user system. _INIT_IOLIB and _CLOSEALL functions are defined in lowsrc.c as an initialized routine of file handler (input / output stream). Initialization of the file handler, and opening files for standard input (stdin), standard output (stdout) and standard error output (stderr) of the file handler are performed in _INIT_IOLIB function. If none of those features: standard input / output and standard error output is necessary, please delete these opening handles. Please note that file handler can be operated only in _INIT_IOLIB function. You can configure member variables

of file handler such as _bufptr, _bufcnt, _bufbase and _buflen by using either setbuf function or setvbuf function after the file open. The processing to close all of the opened files is described in _CLOSE function.

IOSTREAM, which specifies the number of file handler is defined in lowsrc.h. In case of modifying the number of file handler, please modify the macro. Additionally, the three file handlers are open in _INIT_IOLIB function in lowsrc.c generated by HEW as described above. Therefore, when these open processing are enabled, the number of file handler, which can be used is (IOSTREAM – 3).

Open, close, read, write and lseek function are defined in lowsrc.c as low-level interface routines of Input / output libraries.

- File opening requests are distinguished which cause they have been made for between standard input, standard output and standard error output in open function. The file mode is also checked in this function. Any other inputs are treated as errors, and then return –1.
- Checking of a range of the file number and clearing of the file mode are performed in close function. In case that the file number is out of the range, it is treated as an error, and then return –1.
- Charget function, which actually gets a character is called as the same times as the number of the requested characters in read function after checking the file mode. In case that it is an error, return –1.
- Charput function, which actually outputs a character is called as the same times as the number of the requested characters in write function after checking the file mode. In case that it is an error, return –1.
- Nothing is performed in lseek function, which is generated by HEW.

Charget function and charput function, which are called from read function and write function are defined in lowlvl.src. These definitions are enabled only on simulator debuggers. Please note that these definitions are not enabled on target boards.

(3) Termination processing

Abort function is defined in <Project name>.c as a termination processing routine. Please modify it properly according to the user system. In case that the termination processing is not necessary, you can delete the abort function. In addition, please note that there are standard libraries to call the abort function.