Contents

Section 1 Overview

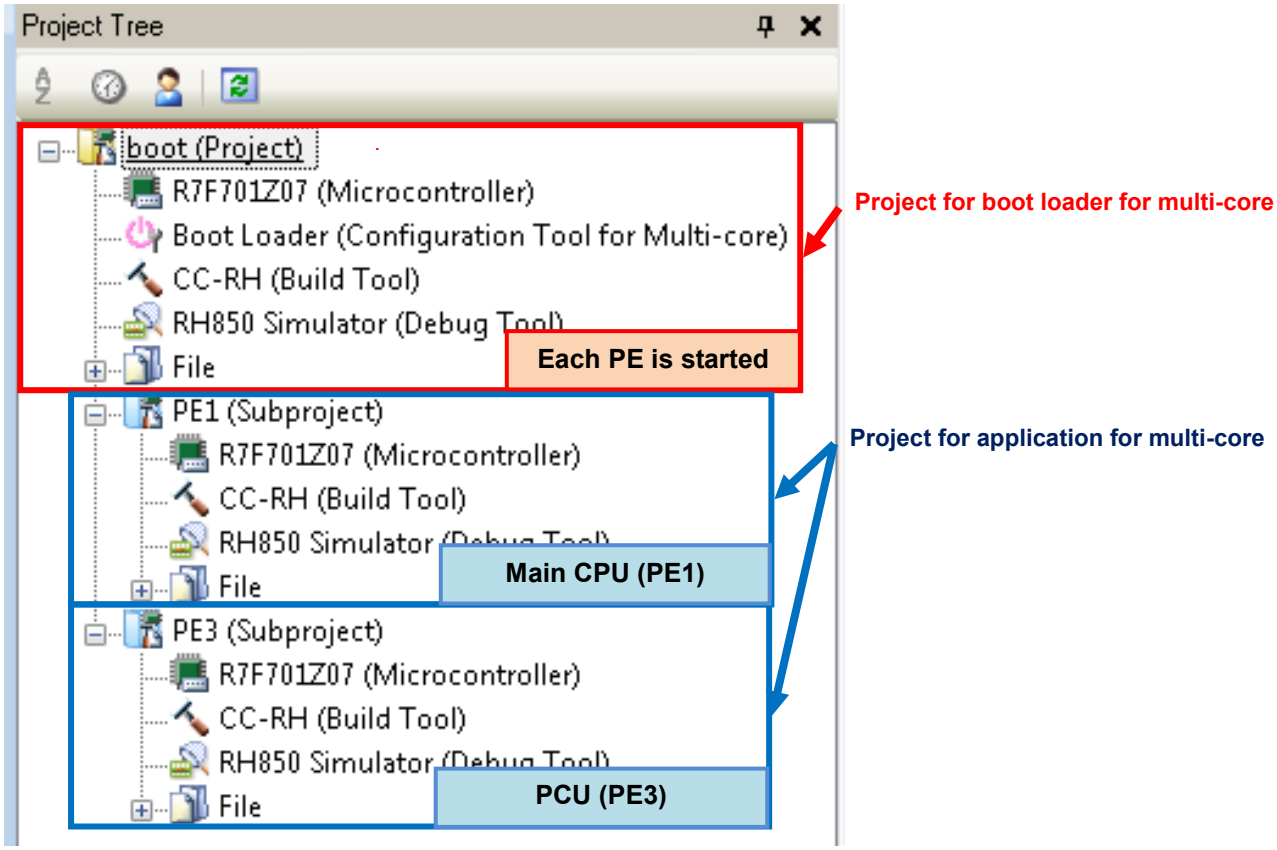
This tutorial explains the steps for using CubeSuite+ to newly create and build a multi-core project that targets the RH850/E1x-FCC1 (R7F701Z07) microcontroller.

This section describes the overview of a multi-core project. A multi-core project consists of a single project for the boot loader for a multi-core and projects for applications for a multi-core for the number of CPU cores mounted on the microcontroller.

1.1 Configuration of a Multi-core Product

When creating a multi-core project, create a project for the boot loader for a multi-core (hereafter referred to as boot loader project) and projects for applications for a multi-core (hereafter referred to as application projects). The boot loader project executes processes starting from a reset and until branching to each application project. An application project executes processes for each PE (processor element).

The Project Tree of CubeSuite+ has the following configuration. The boot loader project serves as the main project, and application projects for the number of PEs serve as subprojects. This kind of a project configuration allows not only one of the PEs to be debugged but both PEs to be debugged in synchronization.



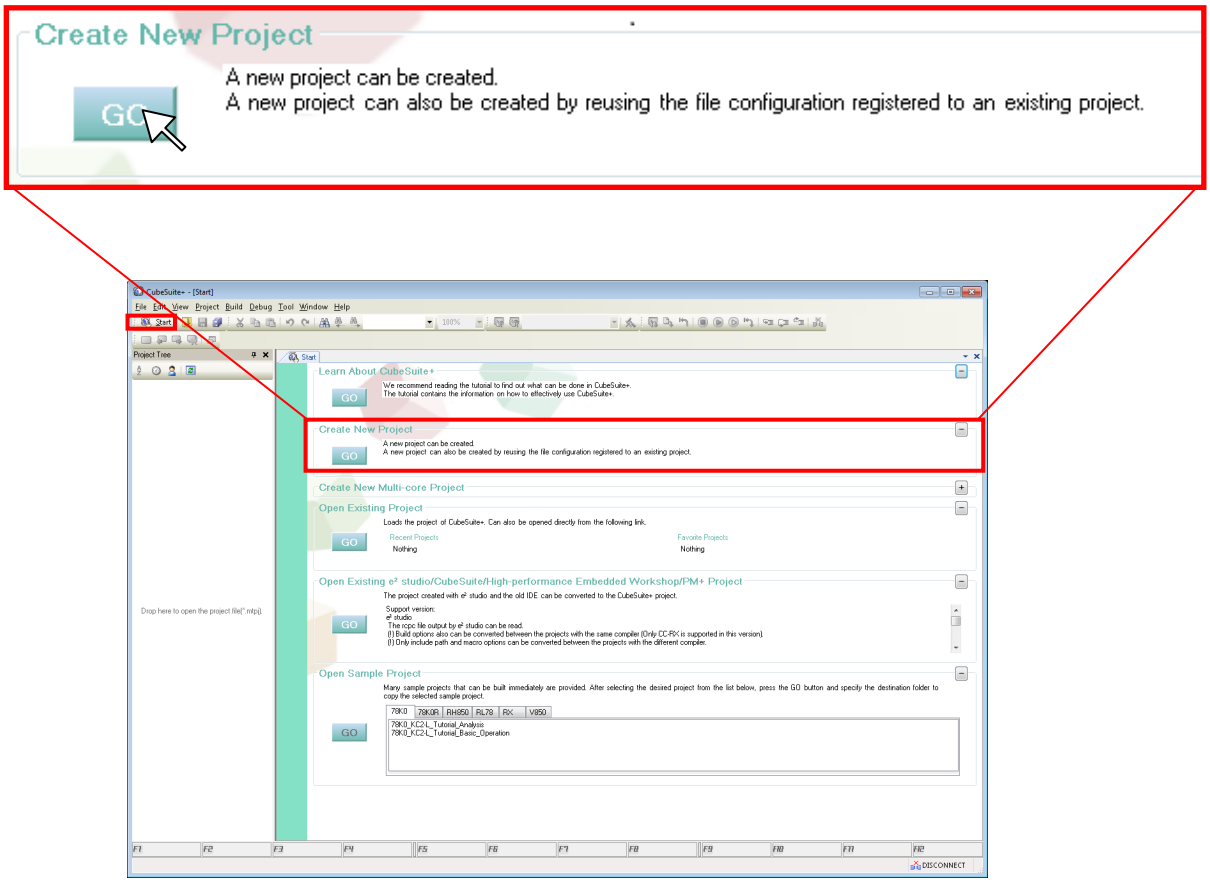
Section 2 Boot Loader Project

This section describes the method for creating a boot loader project. The boot loader project executes processes starting from a reset and until branching to each application project.

2.1 Creating a New Boot Loader Project

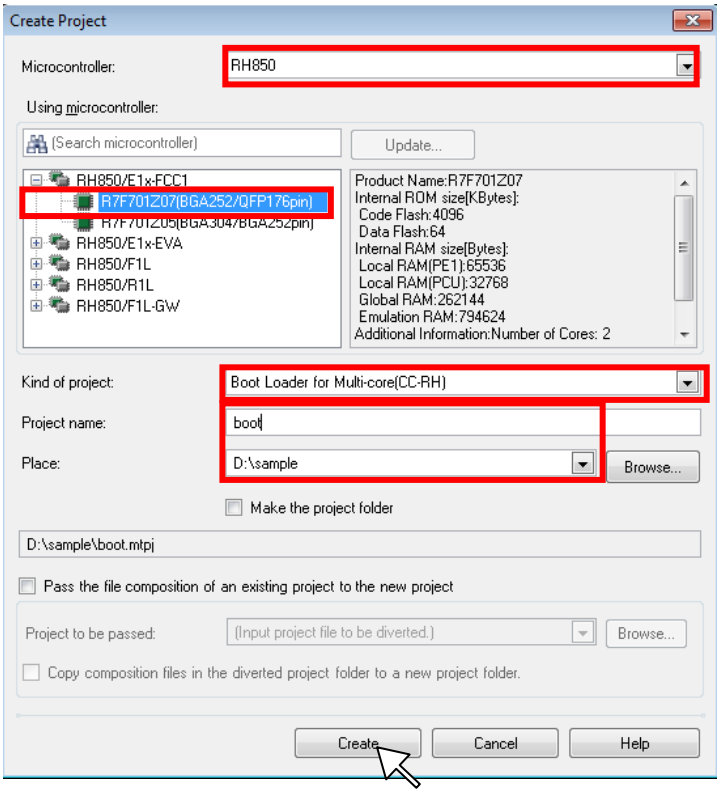
Create a boot loader project following the steps listed below.

1. Create a new project.
   Start CubeSuite+ and click the [Start] button. Then, click the the [GO] button of [Create New Project] on the Start panel.

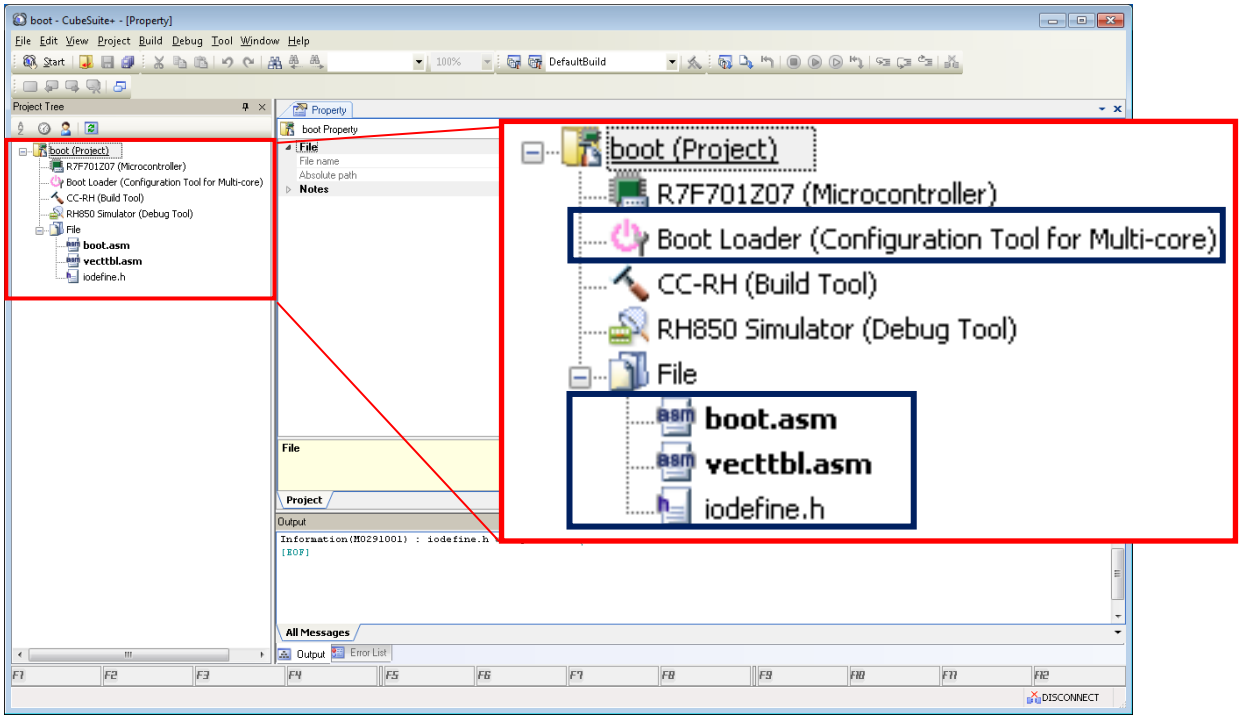


Remarks A project can also be created from [Create New Multi-core Project] on the Start panel.
In this case, if [Create an application project with a boot loader project] is selected in the Create Project dialog box, a boot loader project and a single application project are created. The application project name becomes "*Boot loader project name*_App1". The project name can be changed.

2. Set the project.
   The Create Project dialog box opens. In this dialog box, specify "RH850" in [Microcontroller]. Also select the target microcontroller in [Using microcontroller]. Next, specify "Boot Loader for Multi-core (CC-RH)" in [Kind of project]. Finally, specify Project name and Place, and then click the Create button.



3. Start the boot loader project.
   The boot loader project is started. The Boot Loader node is displayed in the Project Tree. boot.asm, vecttbl.asm, and iodefine.h are automatically registered in the File node.



2.2 Registering the Source

When a boot loader project for a multi-core is newly created in CubeSuite+, the following three files are automatically registered in the File node of the Project Tree.

- Start-up routine for boot loader (boot.asm)
- Exception/interrupt vector table (vecttbl.asm)
- I/O header file (iodefine.h)

The source files for a boot loader project for a multi-core consist of only boot.asm, cstartm.asm, and iodefine.h. boot.asm, cstartm.asm, and iodefine.h are described in the following subsections.

2.2.1 Start-up routine for boot loader

In the start-up routine (boot.asm) for the boot loader, the following processes starting from a reset and until branching to each application project are executed. The processes should be customized if required.

(1) Common entry routine for PEs

The PEID (processor element number) is acquired to identify which PE is executed from among the multiple PEs. Execution branches to the entry routine of each PE according to the acquired PEID. When the PEID is 1, branching is performed to PE1's entry routine (__start_PE1). When the PEID is 2, branching is performed to PE2's entry routine (__start_PE2). When the PEID is 3, branching is performed to PE3's entry routine (__start_PE3).

```
        ; jump to entry point of each PE
        stsr    0, r10, 2           ; get HTCFG0
        shr     16, r10             ; get PEID

        cmp     1, r10
        bz      __start_PE1
        cmp     2, r10
        bz      __start_PE2
        cmp     3, r10
        bz      __start_PE3
```

(2) Entry routine for PE1 (__start_PE1)

Execution branches to a routine (_hdwinit_PE1) to clear RAM and a routine (_init_eiint) to change the mode for EI-level interrupts to table reference mode, and after that, jumps to the application project for PE1.

```
__start_PE1:
        jarl    _hdwinit_PE1, lp            ; initialize hardware
$ifdef USE_TABLE_REFERENCE_METHOD
        jarl    _init_eiint, lp             ; initialize exception
$endif

        mov     #_pm1_setting_table, r13
        ld.w    .OFFSET_ENTRY[r13], r10 ; r10 <- #__start
        jmp     [r10]                       ; jump to #__start
```

Calling _hdwinit_PE1

The global RAM and local RAM (PE1) are initialized to zero for the ECC function. The symbols (GLOBAL_RAM_ADDR, GLOBAL_RAM_END, LOCAL_RAM_PE1_ADDR, and LOCAL_RAM_PE1_END) which are used in initialization are defined at the beginning of the file. These values should be changed to the addresses of the target microcontroller if needed.

```
        ; The following is the addresses in R7F701Z07.
        ; Specify values suitable to your system if needed.
        GLOBAL_RAM_ADDR     .set    0xfeee0000
        GLOBAL_RAM_END      .set    0xfef1ffff

        LOCAL_RAM_PE1_ADDR  .set    0xfedf0000
        LOCAL_RAM_PE1_END   .set    0xfedfffff
```

```
;---------------------------------------------------------------------------
;       hdwinit_PE1
;---------------------------------------------------------------------------
        .section    ".text", text
        .align      2
_hdwinit_PE1:
        mov     lp, r14             ; save return address

        ; clear Global RAM
        mov     GLOBAL_RAM_ADDR, r6
        mov     GLOBAL_RAM_END, r7
        jarl    _zeroclr4, lp

        ; clear Local RAM PE1
        mov     LOCAL_RAM_PE1_ADDR, r6
        mov     LOCAL_RAM_PE1_END, r7
        jarl    _zeroclr4, lp

        mov     r14, lp
        jmp     [lp]
```

```
;---------------------------------------------------------------------------
;       zeroclr4
;---------------------------------------------------------------------------
        .align      2
_zeroclr4:
        br      .L.zeroclr4.2
.L.zeroclr4.1:
        st.w    r0, [r6]
        add     4, r6
.L.zeroclr4.2:
        cmp     r6, r7
        bh      .L.zeroclr4.1
        jmp     [lp]
```

Calling _init_eiint

The mode for EI-level interrupts with interrupt priority levels of 0 to 2 is changed from direct branch mode to table reference mode. Since the change is to be made to table reference mode, the interrupt vector mode select bits of EI-level interrupt control registers EIC0, EIC1, and EIC2 are to be set. As the ICBASE symbol whose value is the EIC0 address is defined and the address is set using the offset from this ICBASE value, the address should be changed to an address for the target microcontroller if required. Note that when the mode for an EI-level interrupt with a different priority is changed to table reference mode, the interrupt vector mode select bit of each EI-level interrupt control register is to be set.
Set the start address of the EIINTTBL section in the INTBP register. The EIINTTBL section is defined in vecttbl.asm. However, since this is commented out by default, the macro "USE_TABLE_REFERENCE_METHOD" that is defined at the beginning of the file needs to be validated to enable this processing.

[When disabled]

```
        ; example of using eiint as table reference method
        ;USE_TABLE_REFERENCE_METHOD .set 1
```

[When enabled] ";" is deleted.

```
        ; example of using eiint as table reference method
        USE_TABLE_REFERENCE_METHOD .set 1
```

```
$ifdef USE_TABLE_REFERENCE_METHOD
;-----------------------------------------------------------------------
;       init_eiint
;-----------------------------------------------------------------------
        ; interrupt control register address
        ICBASE  .set    0xfffeea00

        .align  2
_init_eiint:
        mov     #__sEIINTTBL, r10
        ldsr    r10, 4, 1           ; set INTBP

        ; Some interrupt channels use the table reference method.
        mov     ICBASE, r10         ; get interrupt control register address
        set1    6, 0[r10]           ; set INT0 as table reference
        set1    6, 2[r10]           ; set INT1 as table reference
        set1    6, 4[r10]           ; set INT2 as table reference

        jmp     [lp]
$endif
```

Branching to entry routine of PE1's application project

Read the address of the entry routine (__start_pm) of the application project for PE1 from the application information table (_pm1_setting_table of cstartm.asm) that is created in the application project for PE1. Then, branch to this entry routine.

(3) Entry routine for PE2 (__start_PE2)

Since the RH850/E1x-FCC1 (R7F701Z07) is a microcontroller without PE2, processing is ended by branching to the __exit routine. The __exit routine is a routine that repeatedly branches to itself to keep an unused PE waiting.

```
__start_PE2:
        br      __exit              ; PE2 does not exist in R7F701Z07
```

```
__exit:
        br      __exit
```

(4) Entry routine for PE3 (__start_PE3)

Execution branches to a routine (_hdwinit_PE3) to clear RAM and a routine (_init_eiint) to change the mode for EI-level interrupts to table reference mode, and after that, jumps to the application project for PE3. However, since these processes are commented out, processing is ended by branching to the __exit routine by default, assuming that PE3 is not used. When using PE3, cancel the comment out and make the comment out of "br __exit".

```
__start_PE3:
;       jarl    _hdwinit_PE3, lp    ; initialize hardware
;$ifdef USE_TABLE_REFERENCE_METHOD
;       jarl    _init_eiint, lp             ; initialize exception
;$endif
;       mov     #_pm3_setting_table, r13
;       ld.w    .OFFSET_ENTRY[r13], r10 ; r10 <- #__start
;       jmp     [r10]                       ; jump to #__start

        br      __exit
```

Calling _hdwinit_PE3

The local RAM (PE3) is initialized to zero for the ECC function. The symbols (LOCAL_RAM_PE3_ADDR and LOCAL_RAM_PE3_END) which are used in initialization are defined at the beginning of the file. These values should be changed to the addresses of the target microcontroller if needed.

```
        LOCAL_RAM_PE3_ADDR  .set    0xfedf8000
        LOCAL_RAM_PE3_END   .set    0xfedfffff
```

Calling _init_eiint

Like for PE1, call _init_eiint. The macro "USE_TABLE_REFERENCE_METHOD" that is defined at the beginning of the file needs to be validated to enable this processing.

Branching to entry routine of PE3's application project

Like for PE1, read the address of the entry routine (__start_pm) of the application project for PE3 from the application information table (_pm3_setting_table of cstartm.asm) that is created in the application project for PE3. Then, branch to this entry routine.
Note however that cstartm.asm is created for PE1 and so the application information table name needs to be changed from _pm1_setting_table to _pm3_setting_table to suit PE3.

2.2.2 Exception/Interrupt vector table

The reset process or exception/interrupt process is executed in vector mode. The handler address can be specified in two modes: direct vector mode and table reference mode. For details on setting the mode to select the exception handler address for each interrupt channel used, refer to the specifications for the interrupt controller built in each product.

The exception/interrupt vector table (vecttbl.asm) that is automatically registered in the File node of the Project Tree when creating a new project in CubeSuite+ is described below. The table should be customized if required.

(1) RESET

The address of RESET is obtained by adding the exception source offset (exception source offset of RESET is 0) to the base address indicated by the RBASE register. The RESET handler address of the RH850/E1x-FCC1 (R7F701Z07) is address 0x00000000 when started from the user area and is address 0x01000000 when started from the user boot area.

```
        .section "RESET", text
        .align    512
        jr32      __start ; RESET
```

"jr32 __start" is embedded at the start of the RESET section due to the above definition.
When creating a new project in CubeSuite+, the linker option "-start" specifies the RESET section to be allocated at address 0x01000000.

(2) Exception/Interrupt of direct vector mode

In the direct vector mode, execution branches to a fixed handler address in accordance with the interrupt priority. For the reference location of a handler address, a value obtained by adding the offset of the exception source to the base address indicated by the RBASE register or EBASE register is used. The PSW.EBV bit is used to select which base address is to be used.
When creating a new project in CubeSuite+, an interrupt/exception handler is allocated immediately after the RESET section on assumption that the RBASE register value is used as the base address.

```
        .section "RESET", text
        .align    512
        jr32      __start ; RESET
                                  Allocated immediately after RESET
        .align    16
        jr32      _Dummy ; SYSERR

        .align    16
        jr32      _Dummy ; HVTRAP
        ...
```

By default, an instruction specifying to branch to the dummy function "_Dummy" is located at the offset location corresponding to SYSERR, HVTRAP, FETRAP, etc. "_Dummy" is a routine that repeatedly branches to itself, and it is defined in vecttbl.asm. The routine should be customized if required.

Change the name of "__Dummy" at the offset location corresponding to the exception/interrupt that is to be customized to "_Interrupt function name". Also, define the interrupt function. If the interrupt function is to be defined in the C source file, use the "#pragma interrupt" directive to define it. For details on coding, refer to the manual on coding.

[Sample code] If interrupt function "func1" is executed when exception "SYSERR" occurs

```
.section "RESET", text
.align    512
jr32      __start ; RESET
                          "__Dummy" is changed to "_Interrupt
                          function name".
.align    16
jr32      _func1 ; SYSERR

.align    16
jr32      _Dummy ; HVTRAP

.align    16
jr32      _Dummy ; FETRAP

...
```

```
#pragma interrupt func1(priority=SYSERR, callt=true, fpu=true)
void func1(unsigned long feic)
{
        ...
}
```

(3) Exception/Interrupt of table reference mode

In the RH850, an interrupt in table reference mode can be specified as an extended specification of interrupts. In a direct reference mode, the handler address of an EI-level interrupt is one for each interrupt priority, and multiple interrupt channels with the same priority branch to the same interrupt handler address. However, there are cases where it is preferable for each interrupt handler to use a different code area, due to the application. The table reference mode is defined in the RH850 to handle interrupts which may be used in such kind of manner.

When creating a new project in CubeSuite+, if the exception/interrupt table is in the EIINTTBL section, the allocated address of the dummy function "_Dummy_EI" is embedded in an area which is a multiple of 4 from the start of the EIINTTBL section. According to this, upon occurrence of an exception/interrupt of table reference mode with an interrupt priority of n (n = 0 to 512), execution branches to "_Dummy_EI". "_Dummy_EI" is a routine that repeatedly branches to itself, and it is defined in vecttbl.asm. The routine should be customized if required.

```
.section "EIINTTBL", const
.align    512
.dw       #_Dummy_EI ; INT0
.dw       #_Dummy_EI ; INT1
.dw       #_Dummy_EI ; INT2
.rept     512 - 3
.dw       #_Dummy_EI ; INTn
.endm
```

When creating a new project in CubeSuite+, the EIINTTBL section is specified to be allocated at address 0x00 by the linker option "-start". So if necessary, specify the allocated address.

Change the name of "#_Dummy" at the offset location corresponding to the exception/interrupt that is to be customized to "#_Interrupt function name". Also, define the interrupt function. If the interrupt function is to be defined in the C source file, use the "#pragma interrupt" directive to define it. For details on coding, refer to the manual on coding.

[Sample code] If interrupt function "func2" is executed when EIINT interrupt channel 9 "EIINT9" occurs

```
        .section "EIINTTBL", const
        .align   512
        .dw      #_Dummy_EI ; INT0
        .dw      #_Dummy_EI ; INT1
        .dw      #_Dummy_EI ; INT2
        .dw      #_Dummy_EI ; INT3
        .dw      #_Dummy_EI ; INT4
        .dw      #_Dummy_EI ; INT5
        .dw      #_Dummy_EI ; INT6
        .dw      #_Dummy_EI ; INT7
        .dw      #_Dummy_EI ; INT8
        .dw      #_func2    ; INT9
        .rept    512 - 9
        .dw      #_Dummy_EI ; INTn
```

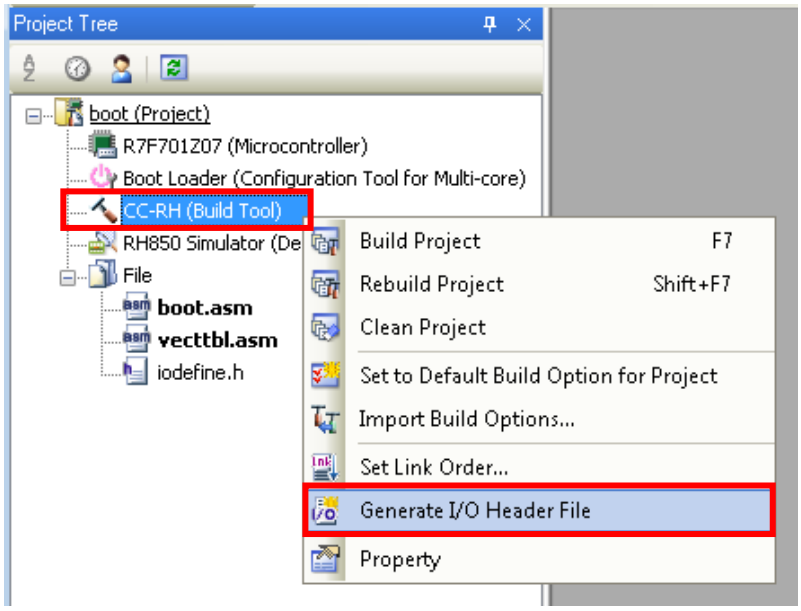
"#_Dummy_EI" is changed to "#_Interrupt function name".

```
#pragma interrupt func2(channel=9, enable=true, callt=true, fpu=true)

void func2(unsigned long eiic)
{
        ...
}
```

2.2.3 I/O header file

When creating a new boot loader project, generate the I/O header file (iodefine.h) for the relevant microcontroller specified in the project and automatically register it in the project. In the I/O header file, register names of the microcontroller and their addresses are defined. If this file is not used in the boot loader project, remove it from the project.

The I/O header file can also be generated by right-clicking the [CC-RH (Build Tool)] node of the CubeSuite+ Project Tree and then selecting [Generate I/O Header File].

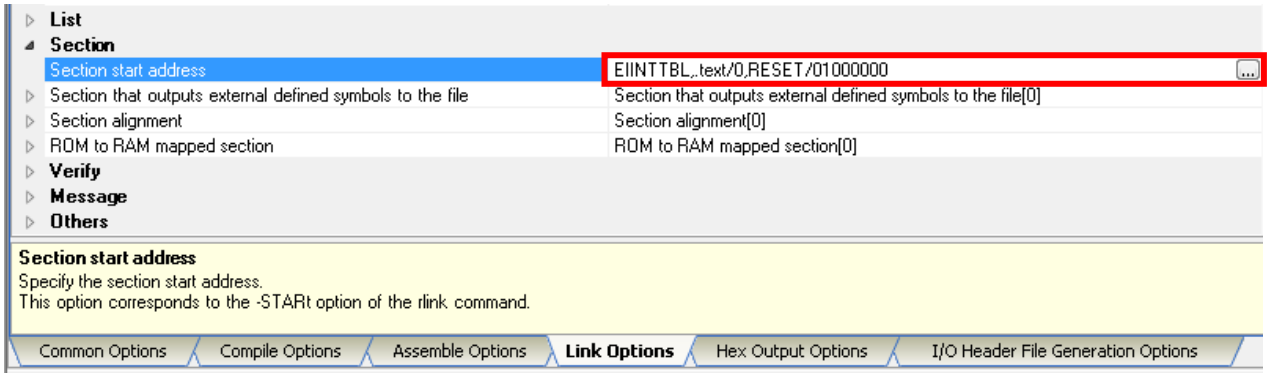


2.3 Setting the Options

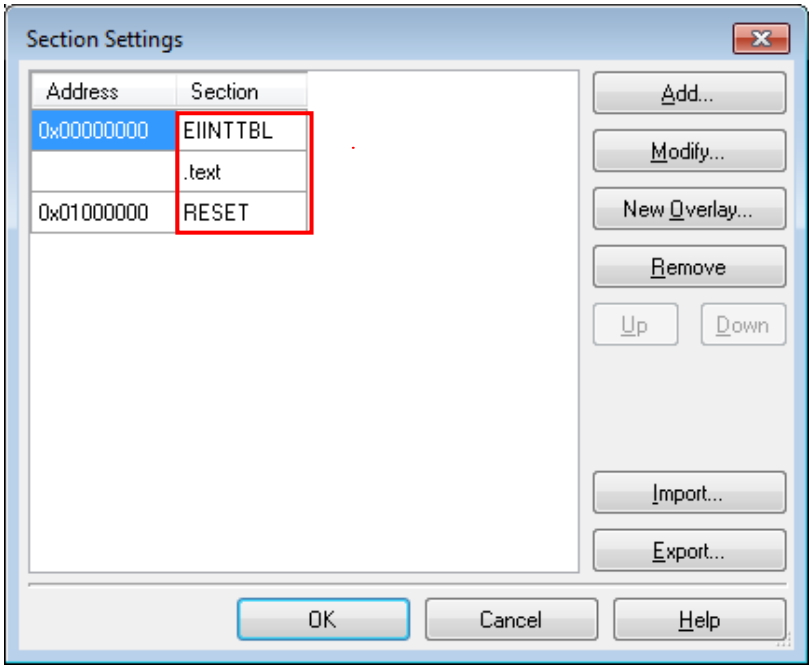
The options used in particular to create the boot loader project are described here.

2.3.1 Link options

Specify the start address of the section by selecting [Link Options] tab -> [Section] category -> [Section start address]. The following specification is made by default. This string is passed to the linker as a parameter of the link option "-start".



Clicking the [...] button at the right edge of the [Section start address] property opens the Section Settings dialog box as shown below. The start address of the section can also be specified from this dialog box.



Based on this section setting, the "EIINTTBL=>.text" section is allocated from address 0x00000000 in the address ascending order and the RESET section is allocated from address 0x01000000. Customize the section settings in this dialog box so that the sections are allocated to the desired addresses.

Section 3 Application Project

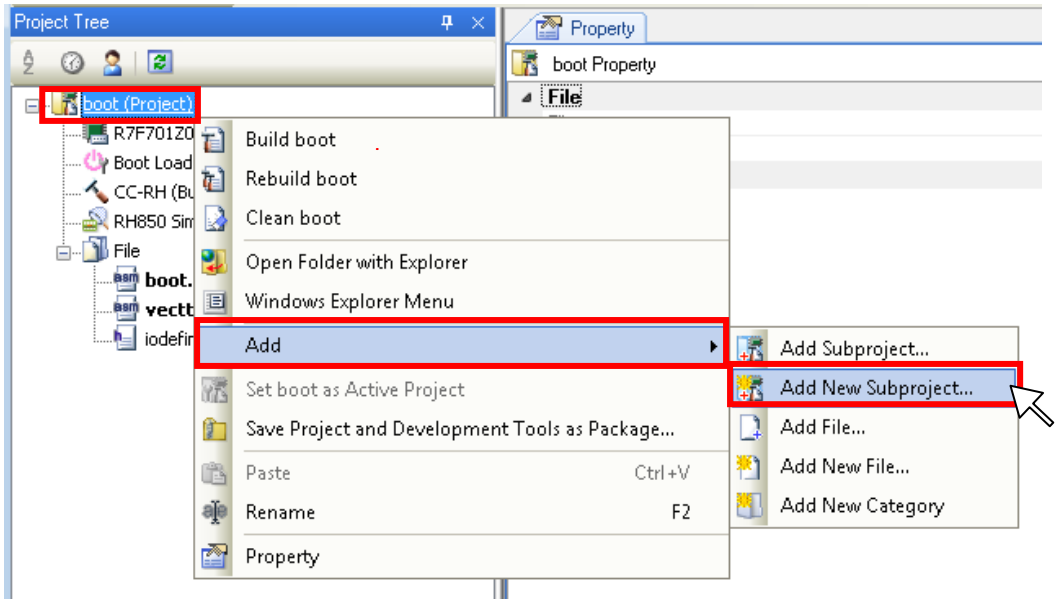
This section describes the method for creating application projects for a multi-core product. An application project executes processes for each PE.

Two application projects are created with PE1 as the project name for the main CPU and PE3 as the project name for PCU. All steps, from creation of a new project, option setting, and up to the build method are described in this section.

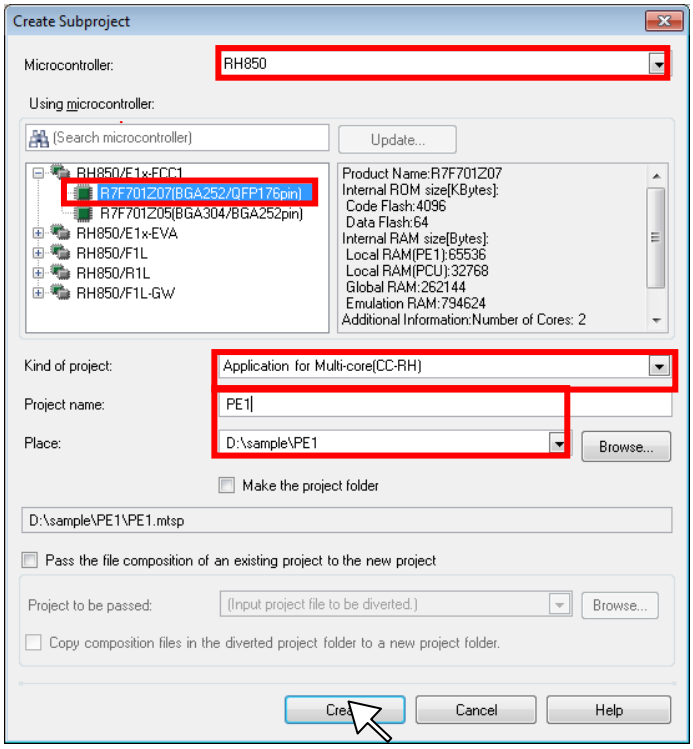
3.1 Creating a New Application Project

An explanation of the procedure for creating an application project as a subproject with the boot loader project already created is given here. Since the RH850/E1x-FCC1 is a dual-core product, two application projects are to be made.

1. Add a subproject.
   Add subprojects to the boot loader project which is the main project. Right-click the [Project] node of the Project Tree -> [Add] -> [Add New Subproject] to add a subproject. When adding a subproject that has already been created, add it from [Add Subproject].

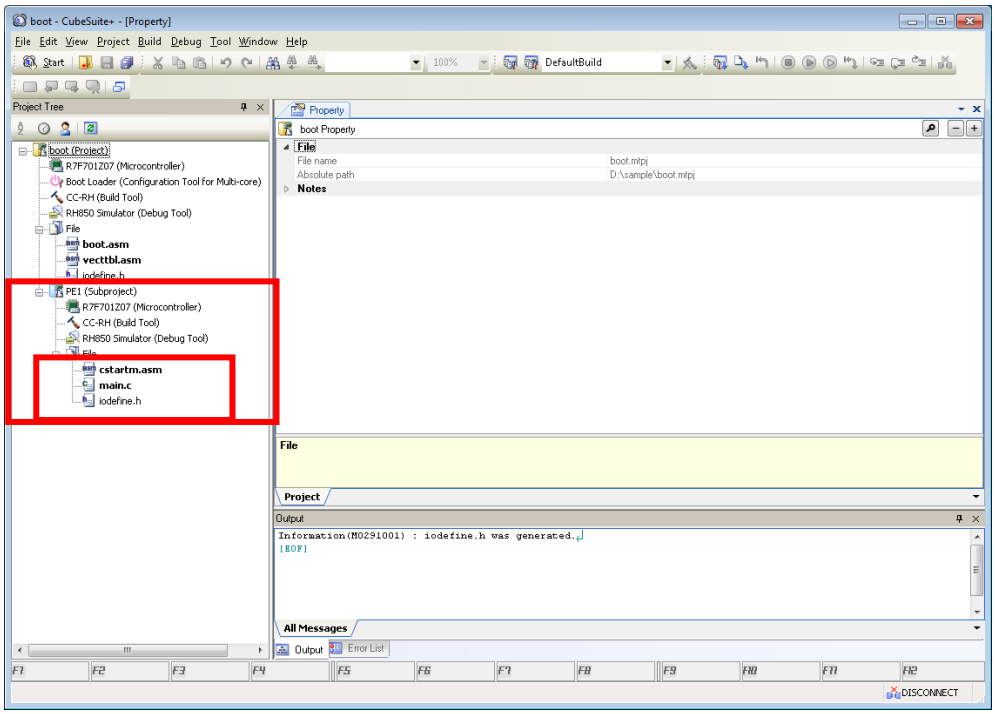


2. Set the subproject.
   The Create Subproject dialog box opens.
   In [Using microcontroller], specify the same microcontroller as that for the boot loader project. Next, specify "Application for Multi-core (CC-RH)" in [Kind of project]. Finally, specify Project name and Place, and then click the Create button.
   Here, the project name is set as PE1 assuming that it is a project for the main CPU.



* When a project was created from [Create New Multi-core Project] on the Start panel, in addition to the boot loader project, one application project for a multi-core will be created as a subproject.

3. Register the application project.
   An application project is registered as a subproject of the boot loader project. cstartm.asm, main.c, and iodefine.h are automatically registered in the File node of the Project Tree for a subproject.



Following a similar procedure, add one more subproject as the project of PCU. That project name is set as PE3.

3.2 Registering the Source

When an application project for a multi-core is newly created in CubeSuite+, the following three files are automatically registered in the File node of the Project Tree.

- Start-up routine for application (cstartm.asm)
- Empty main function (main.c)
- I/O header file (iodefine.h)

Register the necessary source files in the File node of the Project Tree. Registration can be done by dragging and dropping the file into the File node or right-clicking the File node and selecting [Add]. The start-up routine (cstartm.asm) for an application and the I/O header file are described in the following subsections.

3.2.1 Start-up routine for application

In the start-up routine (cstartm.asm) for an application, the start-up process is performed for each PE. The process should be customized if required.

(1) Define the application information table

The application information table (_pm1_setting_table) for PE1 is defined. The entry routine (__start_pm) of PE1's application project which is branched from within __start_PE1 of the start-up routine (boot.asm) for the boot loader is set using this application information table.

```
;-----------------------------------------------------------------------
;           processing module setting table
;-----------------------------------------------------------------------
            .section  ".const.cmn", const
            .public   _pm1_setting_table
            .align    4
_pm1_setting_table:
            .dw       #__start_pm       ; ENTRY ADDRESS
```

[Reference] start PE1 of boot.asm

```
__start_PE1:
            …
            mov       #_pm1_setting_table, r13
            ld.w      .OFFSET_ENTRY[r13], r10  ; r10 <- #__start
            jmp       [r10]                    ; jump to #__start_pm
```

Since the application information table is located in the .const.cmn section, the address of _pm1_setting_table can be passed to the boot loader project by specifying ".const.cmn" as a parameter of the -fsymbol option. In the boot loader project, the address (#__start_pm) of the entry routine of PE1's application project, which is located in _pm1_setting_table is loaded and execution branches to this address.

For an application project for PE3, change the application information table name from _pm1_setting_table to _pm3_setting_table.

```
            .section  ".const.cmn", const
            .public   _pm3_setting_table
            .align    4
_pm3_setting_table:
            .dw       #__start_pm       ; ENTRY ADDRESS
```

(2) Allocating the stack area

0x200 bytes should be allocated as the stack area used by the compiler-generated code for each PE. The stack areas are allocated in the .stack.bss section.

```
;------------------------------------------------------------------------
;           system stack
;------------------------------------------------------------------------
STACKSIZE       .set        0x200
            .section    ".stack.bss", bss
            .align      4
            .ds         (STACKSIZE)
            .align      4
_stacktop:
```

(3) Defining the RAM section initialization table

The table to be specified as a parameter of "_INITSCT_RH" (function that copies the initial values of the RAM section and clears the contents to zero) is defined. The address value (#__s section name) of the starting label in the section and the address value (#__e section name) of the ending label are used in the table. As a default table, the section for variables with initial values is the .data section and the section for variables without initial values is the .bss section, and "-rom=.data=.data.R" has been specified.

```
;------------------------------------------------------------------------
;           section initialize table
;------------------------------------------------------------------------
            .section    ".INIT_DSEC.const", const
            .align      4
            .dw         #__s.data,          #__e.data,          #__s.data.R

            .section    ".INIT_BSEC.const", const
            .align      4
            .dw         #__s.bss,           #__e.bss
```

When a RAM section is newly added, the added section should be defined in the table. Sections added to the table are also subject to copying and zero-clearing by the "_INITSCT_RH" function.

[Sample code] When the .sdata section and .sbss section are added (-rom=.sdata=.sdata.R is specified)

```
;------------------------------------------------------------------------
;           section initialize table
;------------------------------------------------------------------------
            .section    ".INIT_DSEC.const", const
            .align      4
            .dw         #__s.data,          #__e.data,          #__s.data.R
            .dw         #__s.sdata,         #__e.sdata,         #__s.sdata.R


            .section    ".INIT_BSEC.const", const
            .align      4
            .dw         #__s.bss,           #__e.bss
            .dw         #__s.sbss,          #__e.sbss
```

(4) Entry routine for application project (__start_pm)

This is an entry routine for the application project that branches from within __start_PE1 of the start-up routine (boot.asm) for the boot loader. The following process until branching to the main function is performed. The process should be customized if required.

Setting registers

Set values to the SP, GP, and EP registers.

```
;------------------------------------------------------------------------------
;       startup
;------------------------------------------------------------------------------
        .section  ".text", text
        .public   __start_pm
        .align    2
__start_pm:
        mov       #_stacktop, sp            ;   set sp register
        mov       #__gp_data, gp            ;   set gp register
        mov       #__ep_data, ep            ;   set ep register
```

Calling _hdwinit

As necessary, define the hdwinit function and perform the initialization processing for the peripheral devices. If this definition does not exist, the empty hdwinit function in the standard library (libc.lib) is linked and called.

Calling __INITSCT_RH

The section specified by the RAM section initialization table is copied and cleared to zero.

```
        mov       #__s.INIT_DSEC.const, r6
        mov       #__e.INIT_DSEC.const, r7
        mov       #__s.INIT_BSEC.const, r8
        mov       #__e.INIT_BSEC.const, r9
        jarl32    __INITSCT_RH, lp          ;   initialize RAM area
```

Setting FPU

Set PSW.CU0 so the FPU usage is enabled. Also, initialize the FPU function registers (FPSR and FPEPC). Delete this process when it is used as the start-up routine of a CPU that does not incorporate FPU as a processor.

```
        ; set various flags to PSW via FEPSW

        stsr      5, r10, 0               ; r10 <- PSW

        movhi     0x0001, r0, r11
        or        r11, r10
        ldsr      r10, 5, 0               ; enable FPU

        movhi     0x0002, r0, r11
        ldsr      r11, 6, 0               ; initialize FPSR
        ldsr      r0, 7, 0                ; initialize FPEPC
```

Transiting to main function

The following two processes are commented out. Both of them are processes to set the FEPSW register value, and the value will be reflected in PSW upon execution of the feret instruction. Delete the comments to enable these processes if needed.

- Clear the PSW.ID bit and enable interrupts. * Because the PSW.ID value after a reset is 1.
- Set the PSW.UM bit to shift from SV (supervisor mode) to UM (user mode).

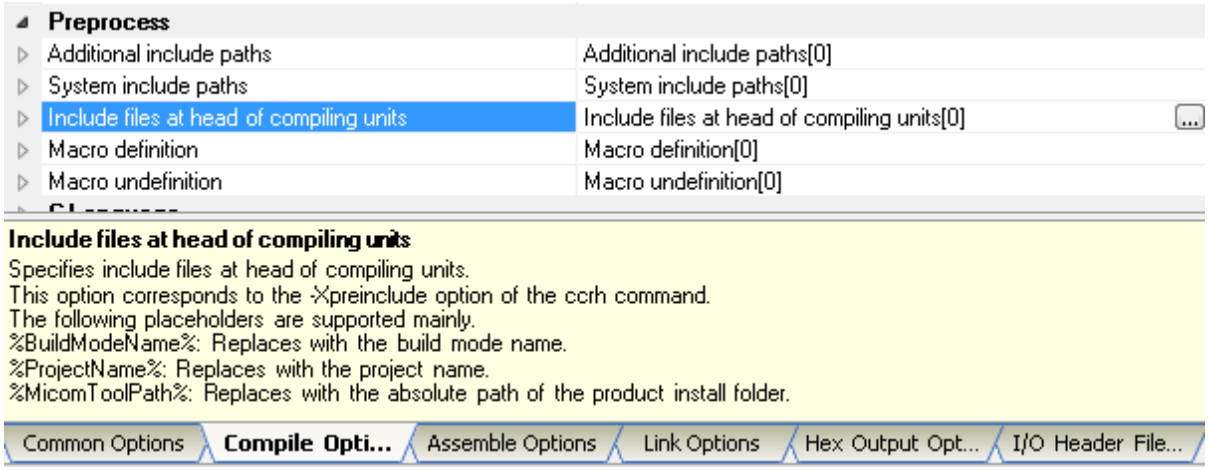
Set the address (#_exit) of _exit (routine that repeatedly branches to itself) to lp and the start address (#_main) of the main function for PE1 to the FEPC register. Later, when the feret instruction is executed, the FEPSW register value is reflected in PSW and the FEPC register value is reflected in PC, and execution shifts to the main function.

```
        ;xori    0x0020, r10, r10   ; enable interrupt

        ;movhi   0x4000, r0, r11
        ;or      r11, r10           ; supervisor mode -> user mode

        ldsr     r10, 3, 0          ; FEPSW <- r10

        mov      #_exit, lp         ; lp <- #_exit
        mov      #_main, r10
        ldsr     r10, 2, 0          ; FEPC <- #_main

        ; apply PSW and PC to start user mode
        ferret

_exit:
        br       _exit                      ;   end of program
```

3.2.2 I/O header file

When creating a new application project, generate the I/O header file (iodefine.h) for the relevant microcontroller specified in the project and automatically register it in the project. In the I/O header file, register names of the microcontroller and their addresses are defined.

When I/O registers are to be accessed in the program, include the I/O header file. Note that an #include specification does not have to be made in the source file when this file is specified as a parameter of the -Xpreinclude option. The -Xpreinclude option can be specified by [Compile Options] tab -> [Preprocess] category -> [Include files at head of compiling units]. In this property, specify the I/O header file for the relevant microcontroller.



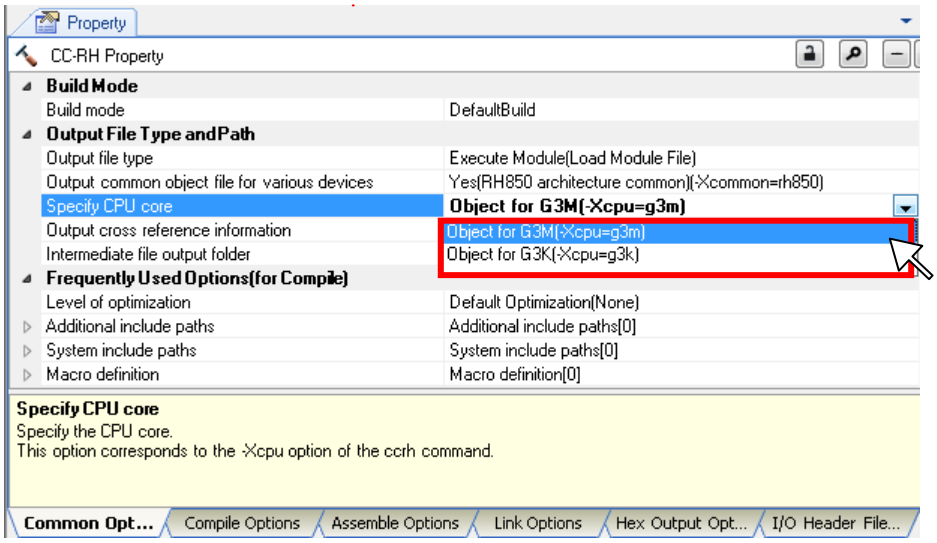
3.3 Setting the Options

The options used in particular to create application projects are described here.

3.3.1 Compile option

-Xcpu option

This option specifies the CPU core. An object for the specified core is generated. Specify either [Object for G3M] or [Object for G3K] at [Common Options] tab -> [Output File Type and Path] category -> [Specify CPU core]. [Object for G3K] is specified by default.

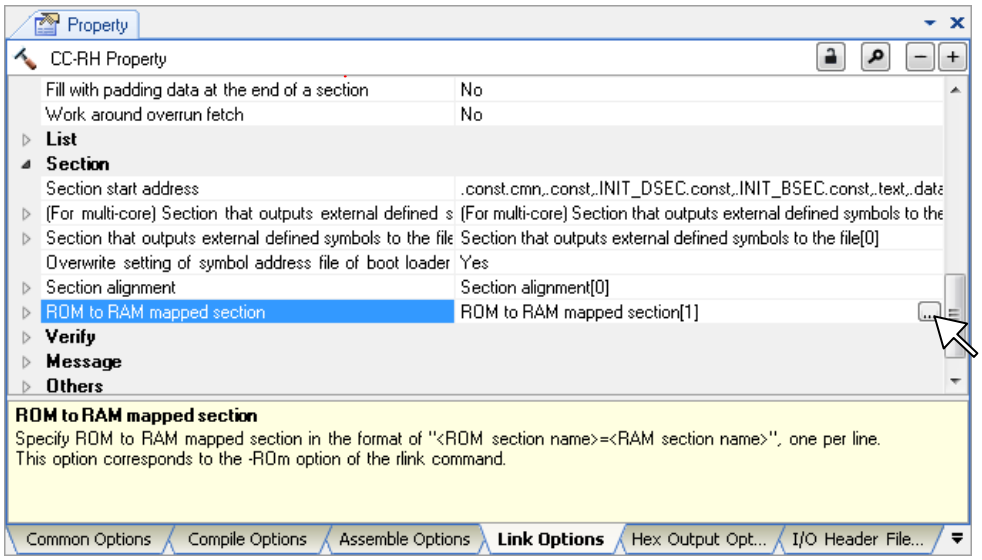


Select [Object for G3M] for a main CPU project and select [Object for G3K] for a PCU project. [Object for G3K] should be selected when compiling a file that includes functions that are executed from both the main CPU and PCU.

3.3.2 Link options

-rom option

The section containing variables with initial values needs to be located in ROM at a reset and in RAM at program execution. This process is called ROMization. The -rom option specifies the section which is mapped from ROM to RAM by ROMization. Click the [...] button at the right edge in [Link Options] tab -> [Section] category -> [ROM to RAM mapped section], and specify the section whose mapping is to be changed from ROM to RAM. Specify one section in one line in the format of <*ROM section name*>=<*RAM section name*>.



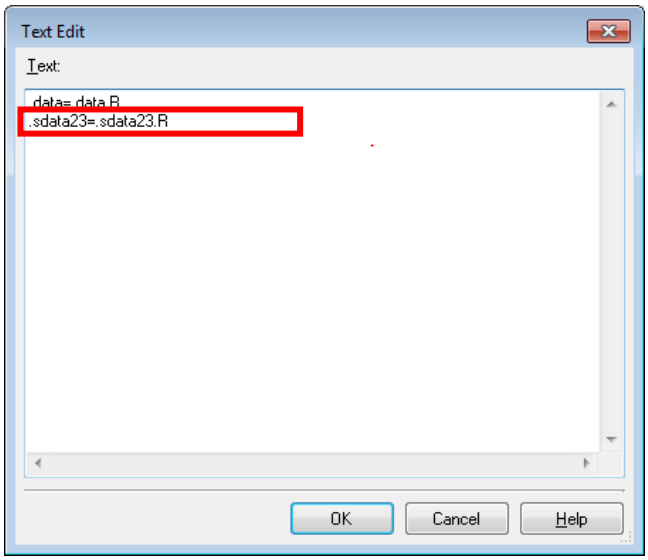
<*ROM section name*> is the section to which ROMization is to be performed. The -start option is used to specify the section set in <*ROM section name*> to be allocated to ROM and the section set in <*RAM section name*> to be allocated to RAM.

The following is specified by default.

```
.data=.data.R
```

When a section other than the .data section is added and it requires ROMization, this option must be additionally specified.

[Example] When .sdata23 is added and it is subject to ROMization (-rom=.sdata23=.sdata23.R is specified)



The initialization table of an added section also has to be added to the section initialization table (.INIT_DSEC.const) of the start-up routine (cstartm.asm). Prefixing the section name with "__s" makes it become a reserved symbol whose value is the start address of that section. Similarly, prefixing the section name with "__e" makes it become a reserved symbol whose value is the end address of that section. Usage of these reserved symbols is recommended for addition to the initialization table.

[Example] When .sdata23 is added (-rom=.sdata23=.sdata23.R is specified)

```
;------------------------------------------------------------------------
;               section initialize table
;------------------------------------------------------------------------
                .section    ".INIT_DSEC.const", const
                .align      4
                .dw         #__s.data,              #__e.data,              #__s.data.R
                .dw         #__s.sdata23,           #__e.sdata23,           #__s.sdata23.R
```

Start address of ROM section — End address of ROM section — Start address of RAM section

In a similar manner, when adding a section to which variables without initial values are located, it has to be added to the section initialization table (.INIT_BSEC.const).

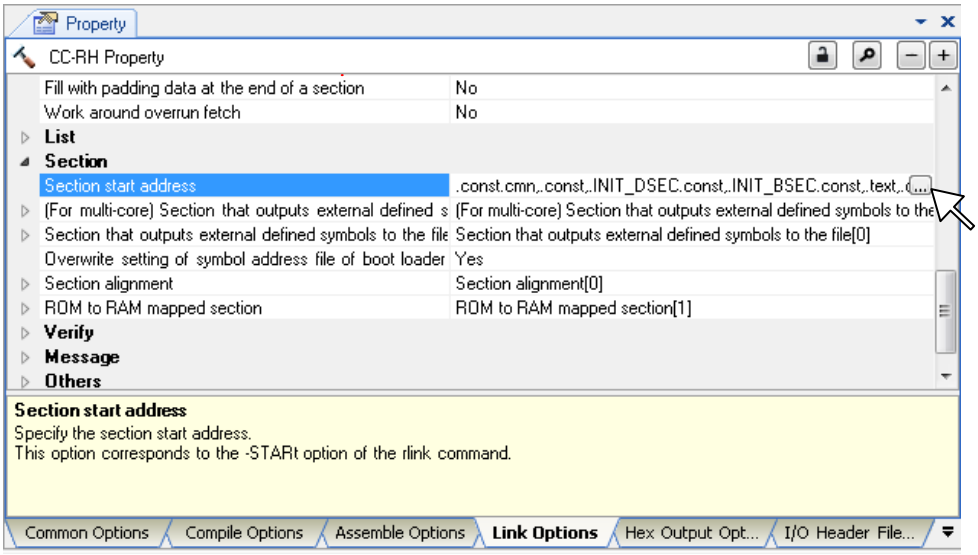
[Example] When .sbss23 is newly added

```
                .section    ".INIT_BSEC.const", const
                .align      4
                .dw         #__s.bss,               #__e.bss
                .dw         #__s.sbss23,            #__e.sbss23
```

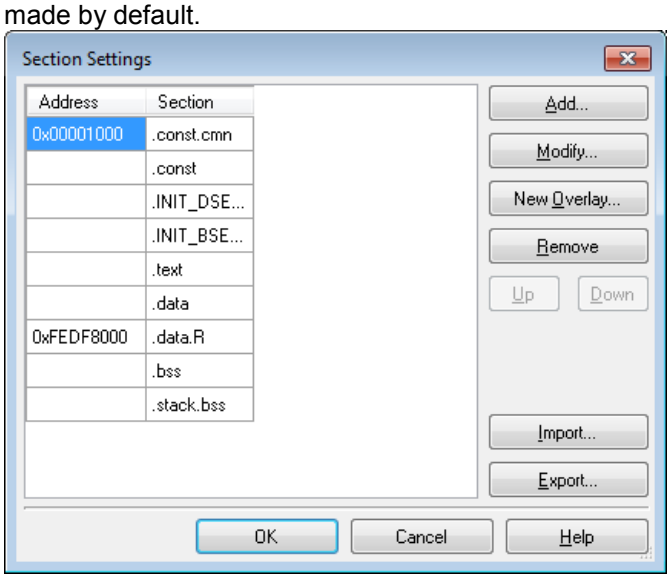
Start address of RAM section — End address of RAM section

-start option

This option specifies the start address of the section. Make the specification from [Link Options] tab -> [Section] category -> [Section start address].



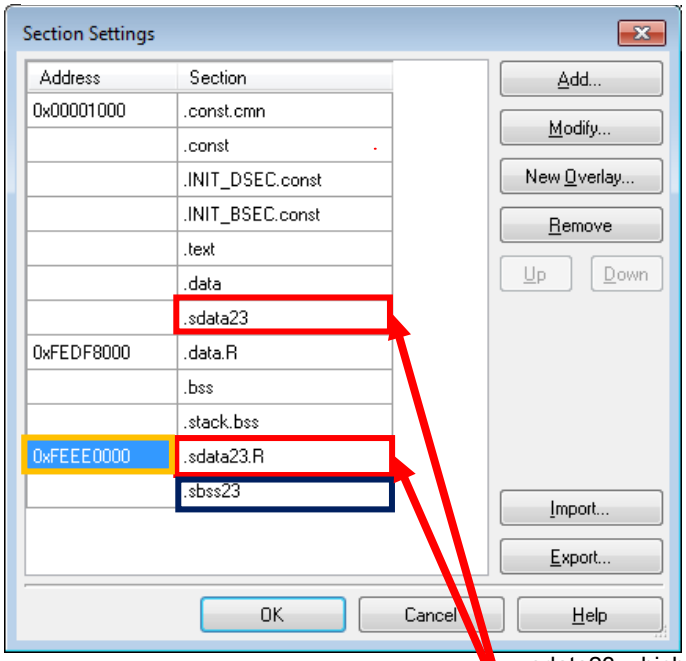
Clicking the [...] button at the right edge, opens the Section Settings dialog box. The following specification is made by default.



The above specification allocates the .const.cmn -> .const -> INIT_DSEC.const -> INIT_BSEC.const -> .text -> .data sections from address 0x1000 in the address ascending order and allocates the .data.R -> .bss -> .stack.bss sections from address 0xFEDF8000 in the address ascending order. Address 0xFEDF8000 is assumed to be the start address of the local RAM self area of PCU. Customize the specification in this dialog box to obtain the desired address allocation. For a project for the main CPU, for example, the local RAM self area starts from address 0xFEDF0000. Therefore, changing the specified address to address 0xFEDF0000 allows the RAM area to be used efficiently.

When a section other than the sections specified by default is added, this option must be additionally specified.

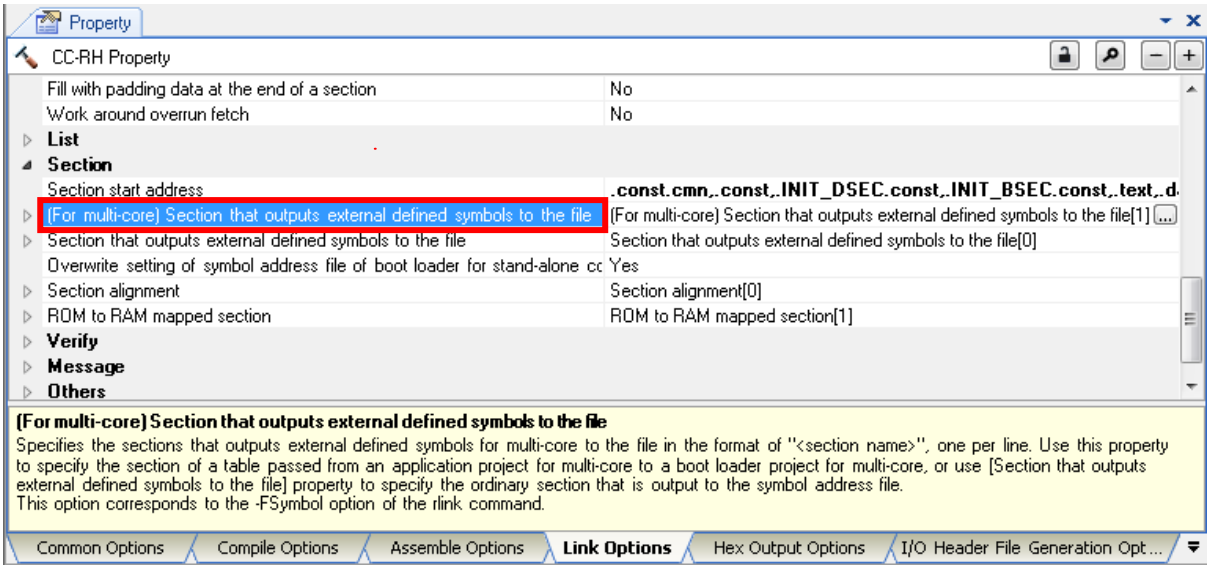
[Example] .sdata23 and .sbss23 are newly added (-rom=.sdata23=.sdata23.R is specified), and these variables are specified to be located at address 0xFEEE0000 which is in the global RAM area.



.sdata23 which is specified as the <*ROM section name*> parameter of the -rom option is specified to be located in the ROM area while .sdata23.R which is specified as the <*RAM section name*> parameter is specified to be located in the RAM area.

-fsymbol option

This option makes externally defined symbols be output to the symbol address file (*.fsy). Externally defined symbols in the section which are specified as parameters of this option are output to the *.fsy file in the assembler directive format. Specify it from [Link Options] tab -> [Section] category -> [(For multi-core) Section that outputs external defined symbols to the file].



In [(For multi-core) Section that outputs external defined symbols to the file], specify a section name that includes external symbols shared between application projects or between the boot loader project and application projects.

The .const.cmn section is specified by default. This means that the address of externally defined symbol _pm1_setting_table is output to the .fsy file. The .fsy file is an assembly source file in which the externally defined symbol is written by an assembler directive. By inputting the .fsy file to a project which wants to share this externally defined symbol and building them together, the externally defined symbol can be shared between projects.

[cstartm.asm by default]

```
;------------------------------------------------------------------------
;               processing module setting table
;------------------------------------------------------------------------
                .section   ".const.cmn", const
                .public    _pm1_setting_table
                .align     4
_pm1_setting_table:
                .dw        #__start  ; ENTRY ADDRESS
```

[Output example of *.fsy file]

```
;SECTION NAME = .const.cmn
  .public _pm1_setting_table
_pm1_setting_table .equ 0x1000
```

Externally defined symbol Located address

_pm1_setting_table can also be referenced from the boot loader project by inputting this *.fsy file to the boot loader project. The following code in boot.asm of the boot loader project triggers a branch to the value (address of __start) in address 0x1000.

```
.OFFSET_ENTRY .set      0
...
mov     #_pm1_setting_table, r13
ld.w    .OFFSET_ENTRY[r13], r10 ; r10 <- #__start
jmp     [r10]                   ; jump to #__start
```

If there is an external symbol shared between projects other than the external symbols in the .const.cmn section, add the section name that contains the external symbol.

3.4 Sharing the Variables

Variables can be shared between projects by using the symbol address file (*.fsy). In other words, variables can be shared between cores.

In this section, variable val1 with an initial value and variable val2 without an initial value are defined in project PE1 and the method for referencing the variables from project PE3 is described.

(1) Changing the section names
By default, variables with initial values are located in the .data section and variables without initial values are located in the .bss section. In the C source file registered in project PE1 as a source file, change the section name of the variables that are to be shared with project PE3.

[Sample code of C source file]

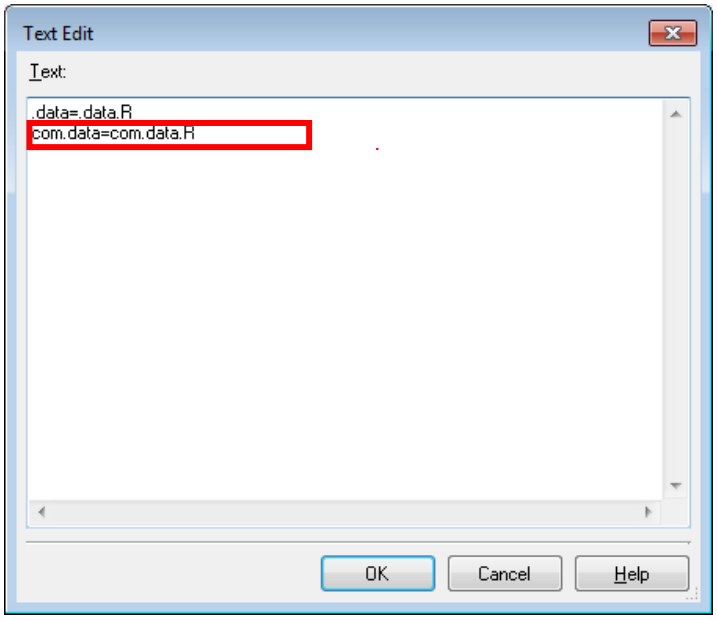
```
#pragma section r0_disp32 "com"

int val1 = 1;          <- com.data section
int val2;              <- com.bss section

#pragma section default
```

When the -Xmulti_level=0 (default) option is specified, val1 is located in the com.data section and val2 is located in the com.bss section.

(2) ROMization
Using the -rom option, specify the section to which variables with initial values are to be located as a target to perform ROMization.
In CubeSuite+, click the [...] button at the right edge of [Link Options] tab -> [Section] category -> [ROM to RAM mapped section], and make the specification in the [Text Edit] dialog box.

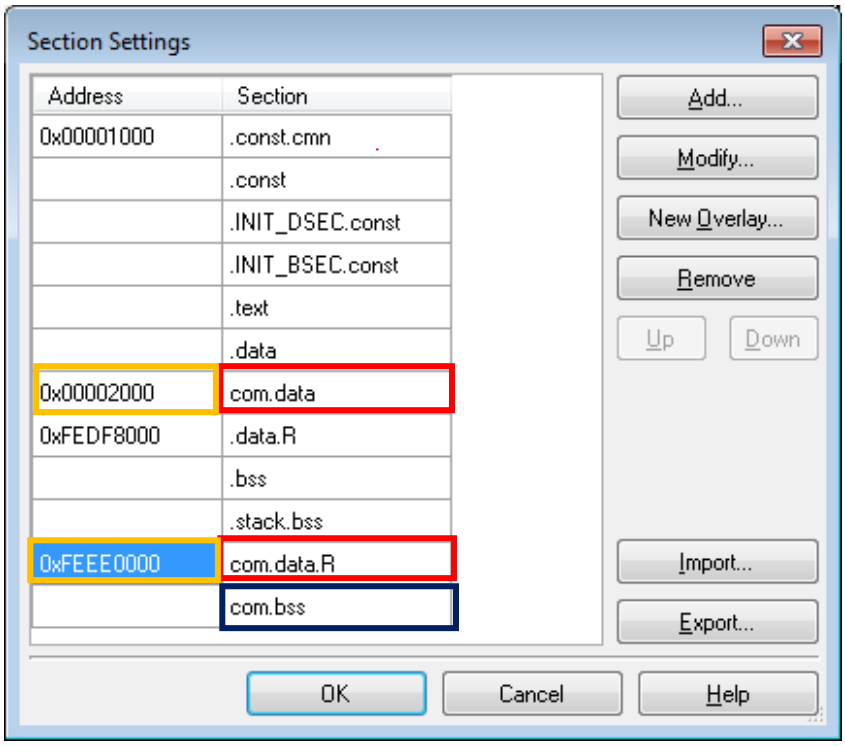
[When com.data.R is specified as the RAM section name of com.data]

```
com.data=com.data.R
```



(3) Allocating the sections
Using the -start option, specify the com.data section to be allocated to the ROM area and the com.data.R section to the RAM area. Similarly, specify the com.bss section to be allocated to the RAM area.
In CubeSuite+, click the [...] button at the right edge of [Link Options] tab -> [Section] category -> [Section start address], and make the specifications in the [Section Settings] dialog box.

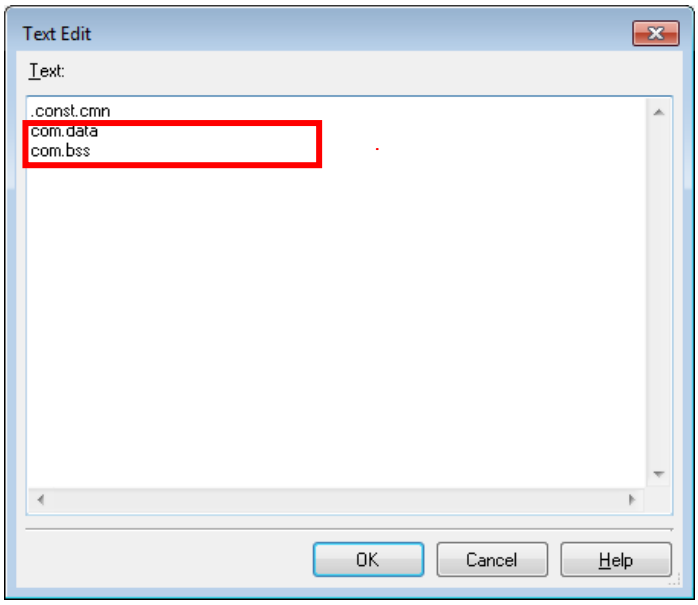
[Example] When allocating the com.data section to address 0x2000 and the com.data.R and com.bss sections to address 0xFEEE0000 which is in the global RAM area



(4) Adding to the section initialization table
Add the address value (#__s section name) of the starting label and the address value (#__e section name) of the ending label in the added section to the section initialization table defined in cstartm.asm.

```
        .section  ".INIT_DSEC.const", const
        .align    4
        .dw       #__s.data,        #__e.data,        #__s.data.R
        .dw       #__scom.data,     #__ecom.data,     #__scom.data.R

        .section  ".INIT_BSEC.const", const
        .align    4
        .dw       #__s.bss,         #__e.bss
        .dw       #__scom.bss,      #__ecom.bss
```

(5) Outputting the *.fsy file
Output the variable name and its located address to the *.fsy file using the -fsymbol option. In CubeSuite+, click the [...] button at the right edge of [Link Options] tab -> [Section] category -> [(For multi-core) Section that outputs external defined symbols to the file], and specify the com.data and com.bss sections in the [Text Edit] dialog box.



When rebuilding is executed, the *.fsy file as shown below is output. The file name is "*Project name.fsy*". "_val1" being located at address 0xFEEE0000 and "_val2" being located at address0xFEEE0004 as externally defined symbols will be indicated.
Registering this *.fsy file in another project as a source file enables externally defined symbols, variables "_val1" and "_val2" in this case, to be referenced from another project.

```
;SECTION NAME = .const.cmn
  .public _pm1_setting_table
_pm1_setting_table .equ 0x1000
;SECTION NAME = com.data
  .public _val1
_val1 .equ 0xfeee0000
;SECTION NAME = com.bss
  .public _val2
_val2 .equ 0xfeee0004
```

(6) Registering the *.fsy file as a source file
Register the *.fsy file that is output in step 5 in the File node of the Project Tree for project PE3 which references externally defined symbols "_val1" and "_val2". Registration can be done by dragging and dropping the file into the File node or right-clicking the File node and selecting [Add].

3.5 Sharing the Functions

Functions can be shared between projects by using the symbol address file (*.fsy). In other words, functions can be shared between cores.

In this section, function func is defined in project PE1 and the method for referencing the function from project PE3 is described.

(1) Changing the section name
By default, functions are located in the .text section. In the C source file registered in project PE1 as a source file, change the section name of the function that is to be shared with project PE3.

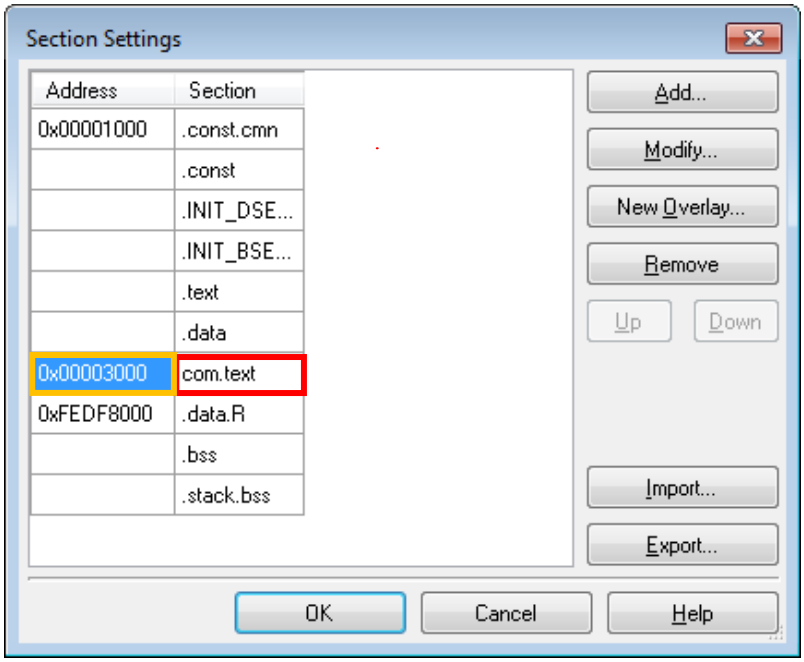
[Sample code of C source file]

```
#pragma section text "com"

void func (void) {
  ...
}

#pragma section default
```

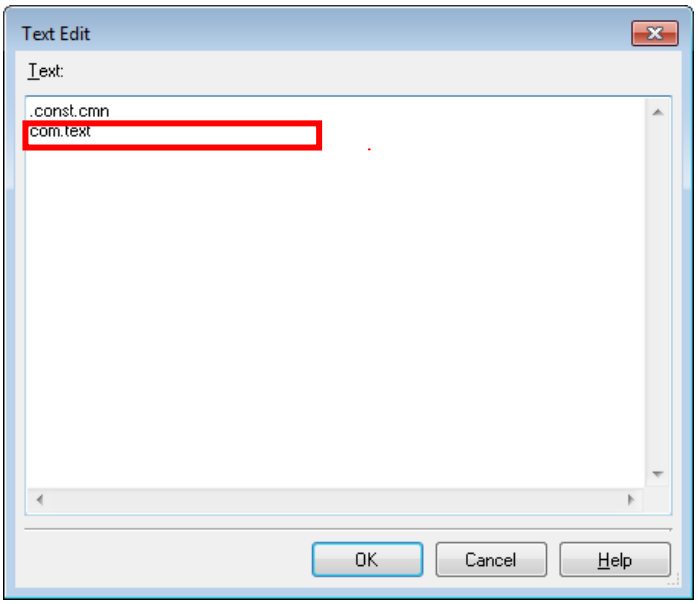
func is located in the com.text section.

(2) Allocating the section
Using the -start option, specify where to allocate the com.text section.
In CubeSuite+, click the [...] button at the right edge of [Link Options] tab -> [Section] category -> [Section start address], and make the specification in the [Section Settings] dialog box.

When allocating the com.text section to address 0x3000



(3) Outputting the *.fsy file
Output the function name and its located address to the *.fsy file using the -fsymbol option.
In CubeSuite+, click the [...] button at the right edge of [Link Options] tab -> [Section] category -> [(For multi-core) Section that outputs external defined symbols to the file], and specify the com.text section in the [Text Edit] dialog box.



When rebuilding is executed, the *.fsy file as shown below is output. The file name is "*Project name.fsy*". "_func" being located at address 0x3000 as an externally defined symbol will be indicated.
Registering this *.fsy file in another project as a source file enables externally defined symbol "_func" to be referenced from another project.

```
;SECTION NAME = .const.cmn
  .public _pm1_setting_table
_pm1_setting_table .equ 0x1000
;SECTION NAME = com.text
  .public _func
_func .equ 0x3000
```

(4) Registering the *.fsy file as a source file
Register the *.fsy file that is output in step 3 in the File node of the Project Tree for project PE3 which references externally defined symbol "_func". Registration can be done by dragging and dropping the file into the File node or right-clicking the File node and selecting [Add].

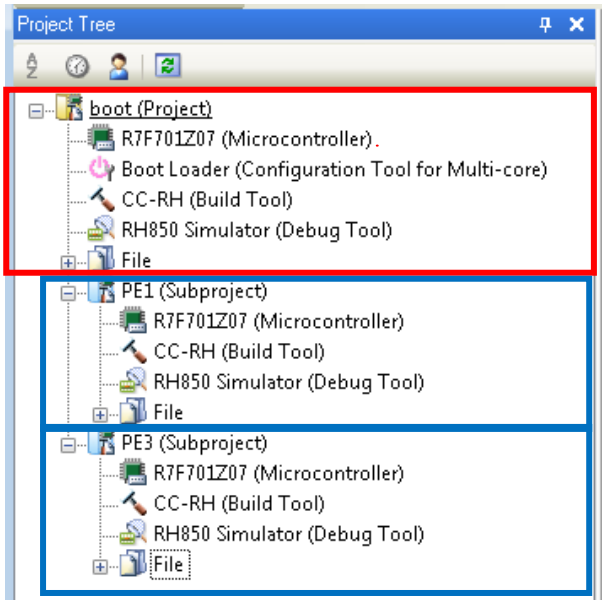
(5) Specifying the -Xcpu=g3k option
When compiling a source file that includes a shared function, individually specify the -Xcpu=g3k option.
In CubeSuite+, right-click the relevant source file in the Project Tree -> [Property] -> [Build Settings] tab -> select "Yes" in [Set individual compile option]. Then, directly specify the -Xcpu=g3k option in [Individual Compile Options] tab -> [Others] category -> [Other additional options].

Section 4 Rebuilding

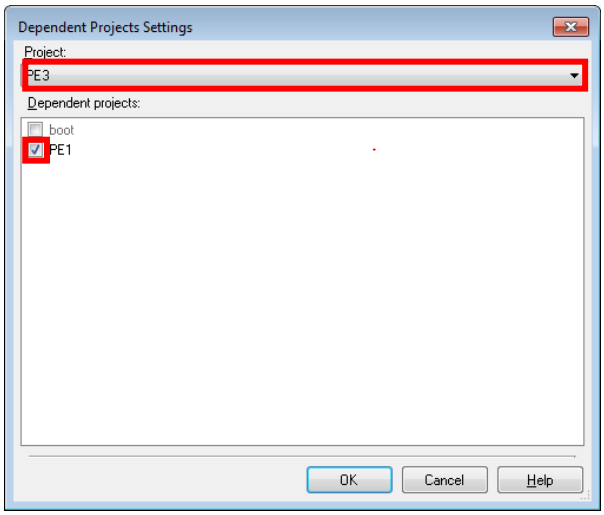
This section describes the method for rebuilding the boot loader project and application projects.

4.1 Rebuilding Multiple Projects

Rebuild the boot loader project and two application projects. Note that it is recommended to define the shared variables and shared functions in the subproject that is to be rebuilt first.
Rebuilding is performed in the order of PE1 (subproject) => PE3 (subproject) => boot (main project), by default. If the shared variables and shared functions are defined in PE1 which is to be rebuilt first and the symbol address file (*.fsy) including the definitions is registered in PE3, an *.fsy file that is always updated will be input to PE3 at rebuilding and so the rebuild process needs to be executed only once.

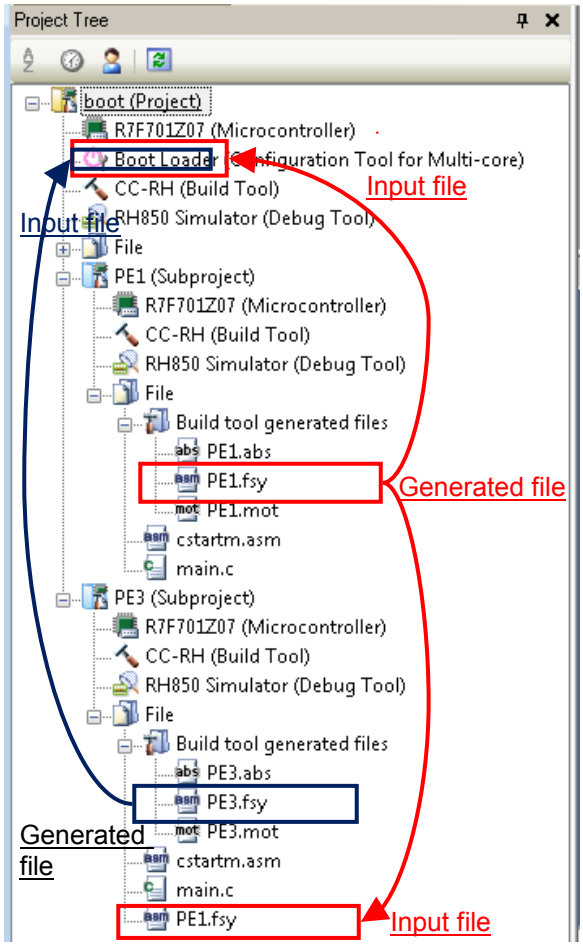


Note that the building order of projects can be controlled in CubeSuite+. Select [Dependent Projects Settings] from the [Project] menu and make settings in the [Dependent Projects Settings] dialog box.



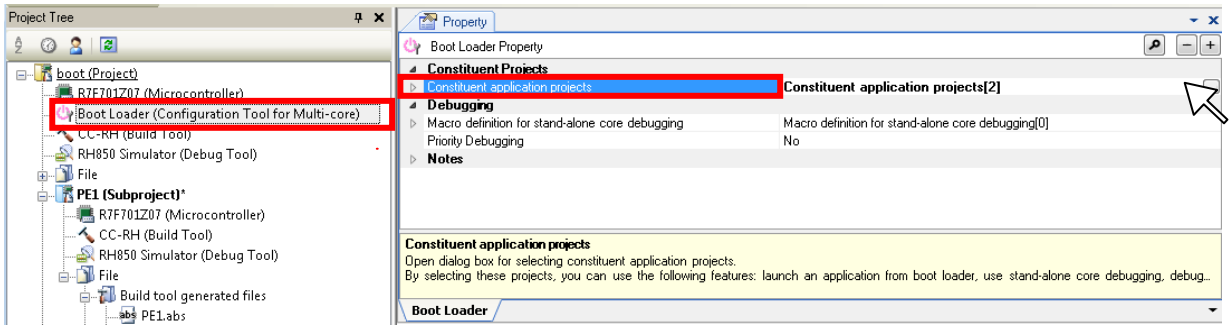
This dialog box setting specifies that PE3 is a project dependent on PE1. As a result, rebuilding takes place in the order of PE1 => PE3 => boot.

A project configuration example is shown below. In this configuration, "boot" is the boot loader project, "PE1" is the application project for the main CPU, and "PE3" is the application project for PCU.

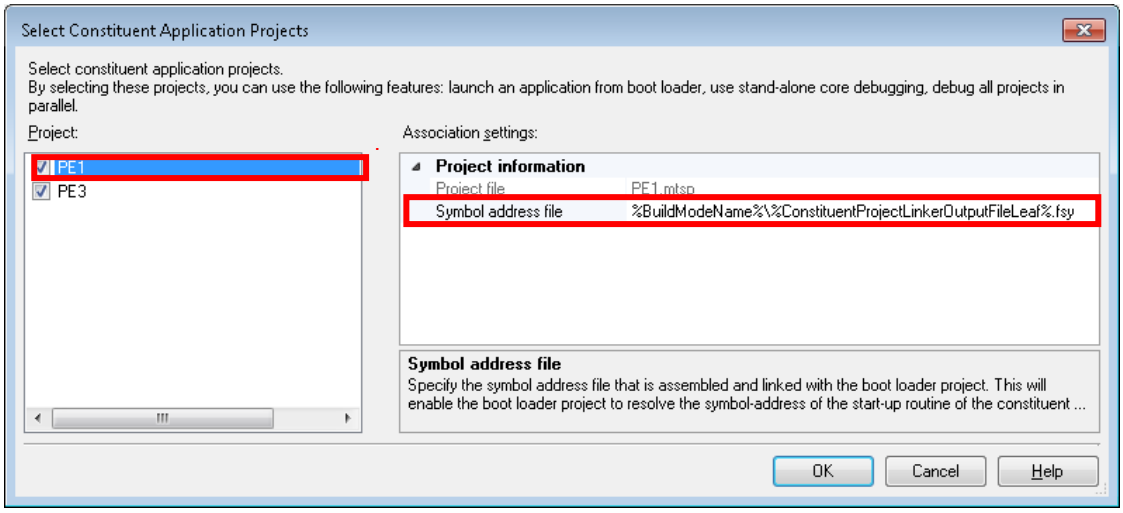


1. When the entire project is rebuilt, first the "PE1" project will be rebuilt. This generates the symbol address file (PE1.fsy) containing the located addresses of the application information table (_pm1_setting_table) and shared variables and shared functions which are defined in "PE1".
2. Next, the "PE3" project will be rebuilt. Inputting PE1.fsy to "PE3" permits the shared variables and shared functions defined in "PE1" to be accessed from "PE3". Also, the symbol address file (PE3.fsy) containing the located address of the application information table (e.g. "_pm3_setting_table") defined in "PE3" is generated.
3. Finally, the "boot" project will be rebuilt. By inputting PE1.fsy and PE3.fsy to "boot", the located addresses of the application information tables defined in "PE1" and "PE3" are determined.
   Note that if "PE1" and "PE3" are registered as constituent projects of "boot", these *.fsy files will be automatically registered.

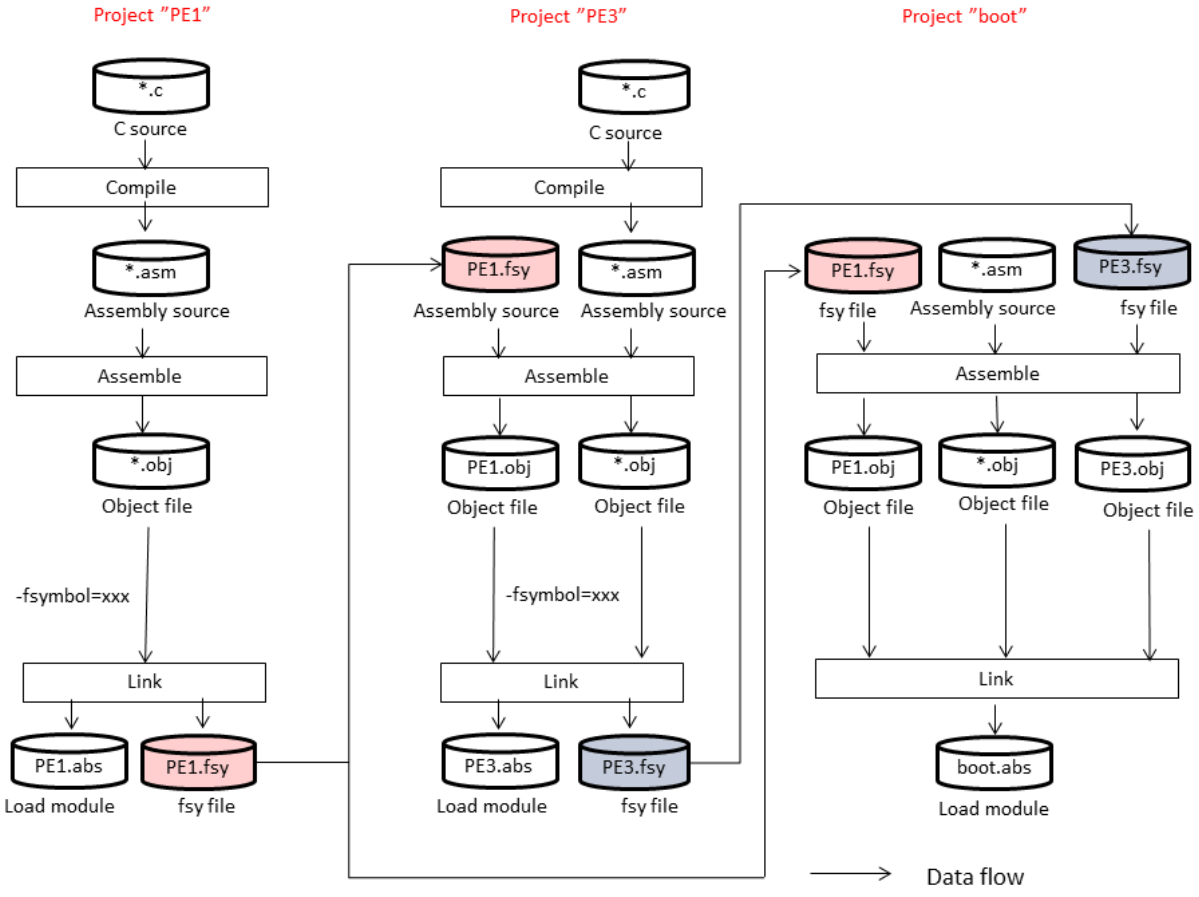
To check if *.fsy files are registered in the boot loader project, right-click [Boot Loader (Configuration Tool for Multi-core)] and select [Property]. In the Property panel, click the [...] button at the right edge of [Constituent Projects] category -> [Constituent application projects].



This opens the [Select Constituent Application Projects] dialog box as shown below. It can be confirmed that the application projects added to the boot loader project have been selected and the *.fsy files that will be generated in that application project are associated.



The flow for rebuilding the "PE1", "PE3", and "boot" projects is as follows:

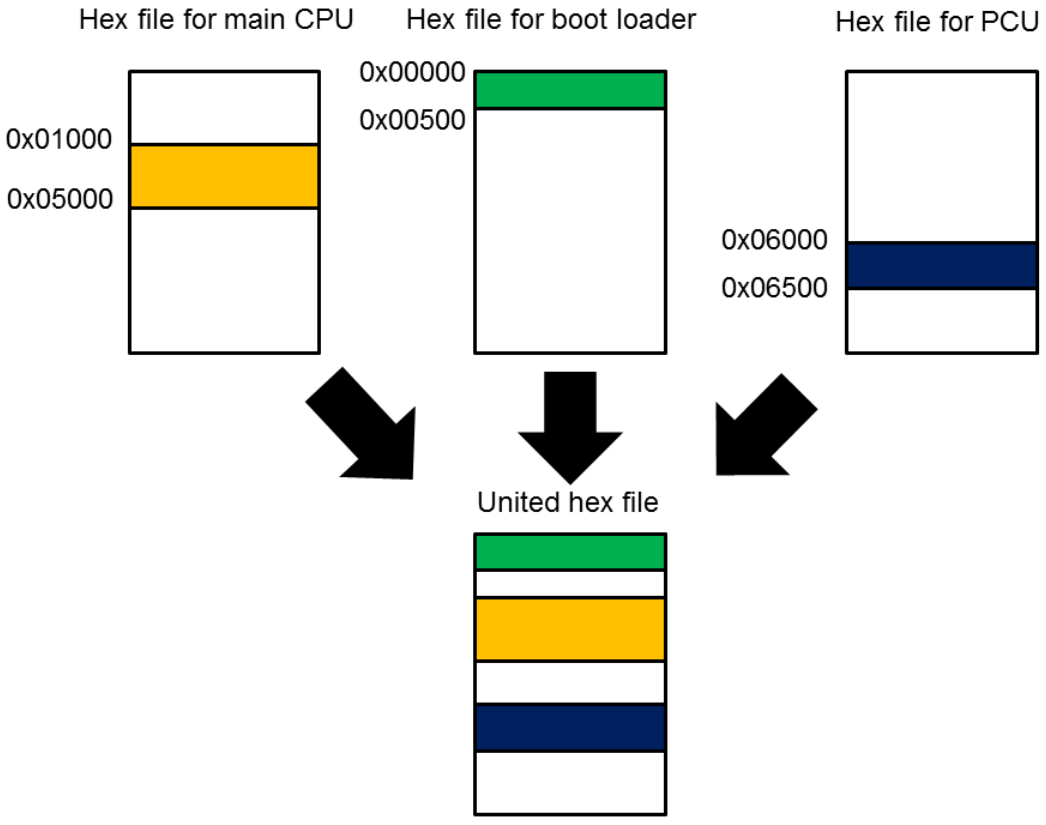


Section 5 Uniting the Objects

This section describes the function for selecting constituent application projects and uniting multiple objects.

5.1 What is the Object Uniting Function

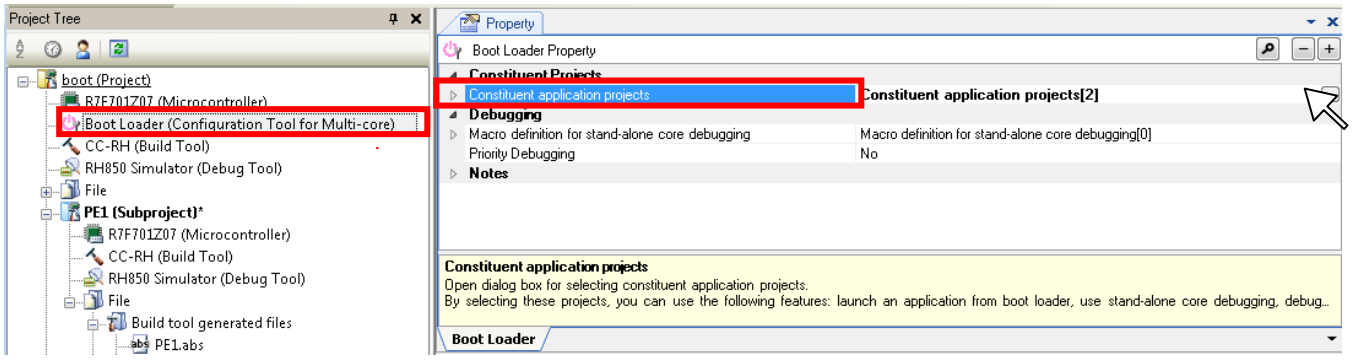
When the boot loader project and two application projects are rebuilt, three load modules and three hex files (Intel HEX files or Motorola S-record files) are generated. The generated multiple hex files can be united to generate a single hex file as a whole. This is performed by the object uniting function. The hex files can be managed as a single file using the object uniting function. However, since Intel HEX files and Motorola S-record files cannot be united, the hex format for the boot loader project and application projects must be the same.
Note that when uniting multiple hex files, if the addresses overlap, an overlap error occurs. However, RAM areas not containing any data cannot be checked. Therefore, the user must check whether addresses are not overlapping by referencing the map file or using other means. The "-cpu" option that checks the addresses where sections are allocated can be used for checking.



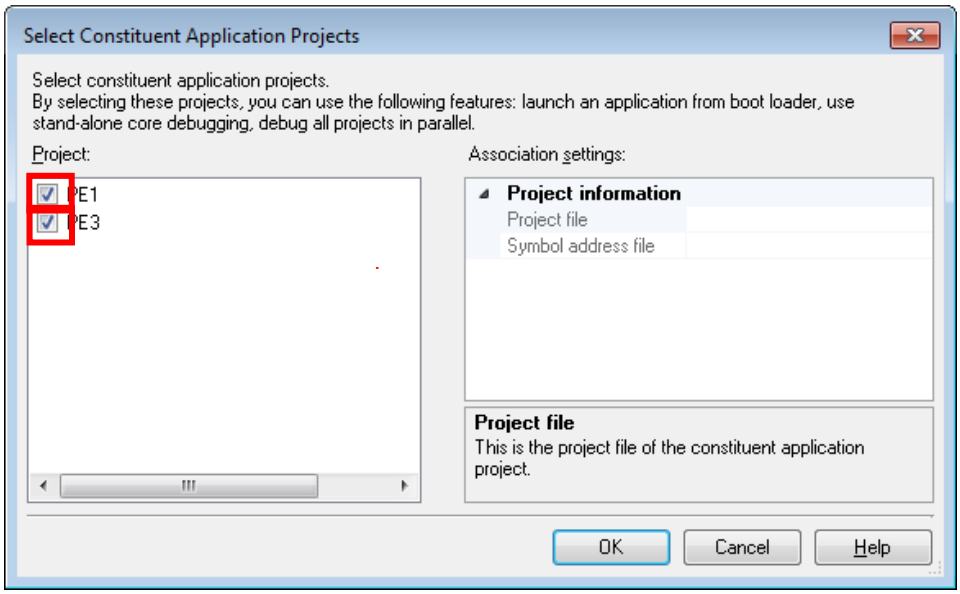
5.2 Selecting "Constituent application projects"

Created application projects need to be registered in the boot loader project as "Constituent application projects". Note that application projects added to the boot loader project are registered as "Constituent application projects" by default.

Constituent application projects can be changed in [Constituent Projects] category -> [Constituent application projects] on the Property panel. The Property panel is opened by right-clicking [Boot Loader (Configuration Tool for Multi-core)] of the Project Tree and selecting [Property].



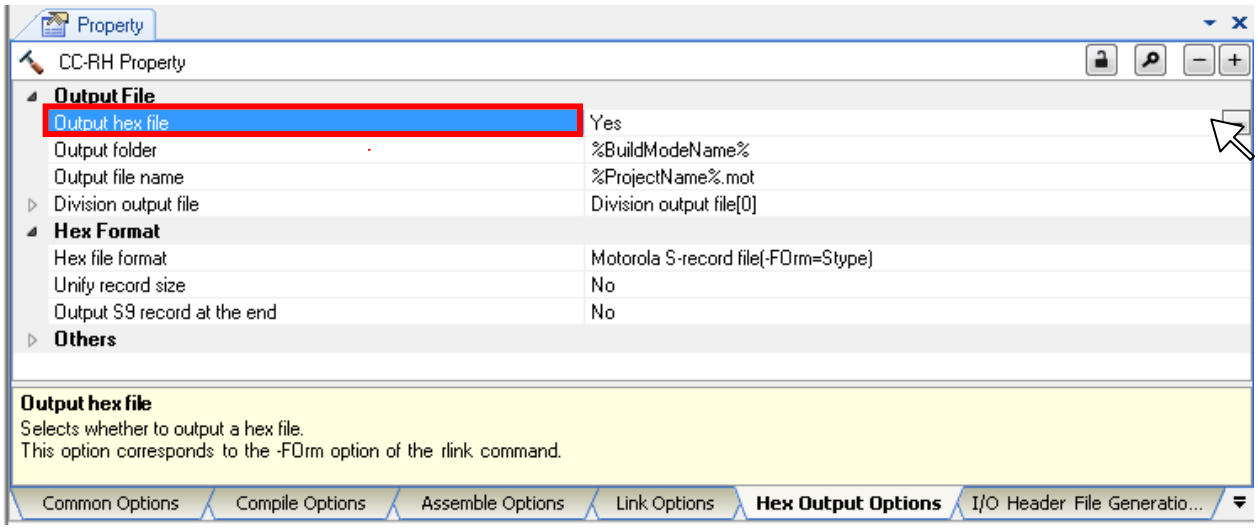
Click the [...] button at the right edge. This opens the [Select Constituent Application Projects] dialog box as shown below. Application projects "PE1" and "PE3" which have been added to boot loader project "boot" are selected, and they are registered as constituent application projects of "boot".



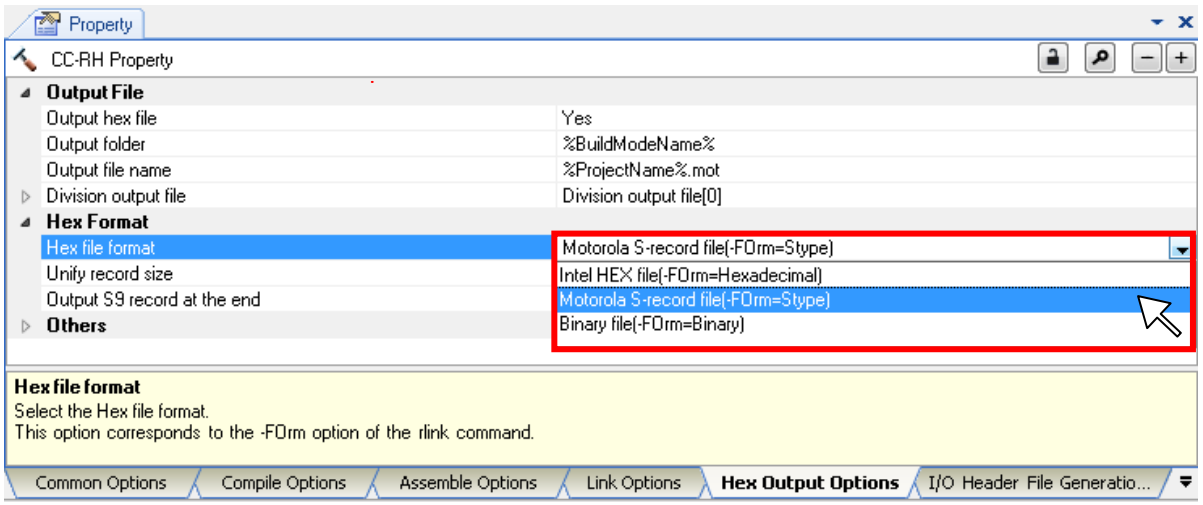
5.3 Uniting the Objects

The generated hex files (Intel HEX files or Motorola S-record files) can be united following the steps listed below. Uniting is performed by the object uniting function. Using the object uniting function, the hex files generated in the boot loader project can be united with the hex files which are for the main CPU and PCU and are generated in application projects to generate a single hex file.

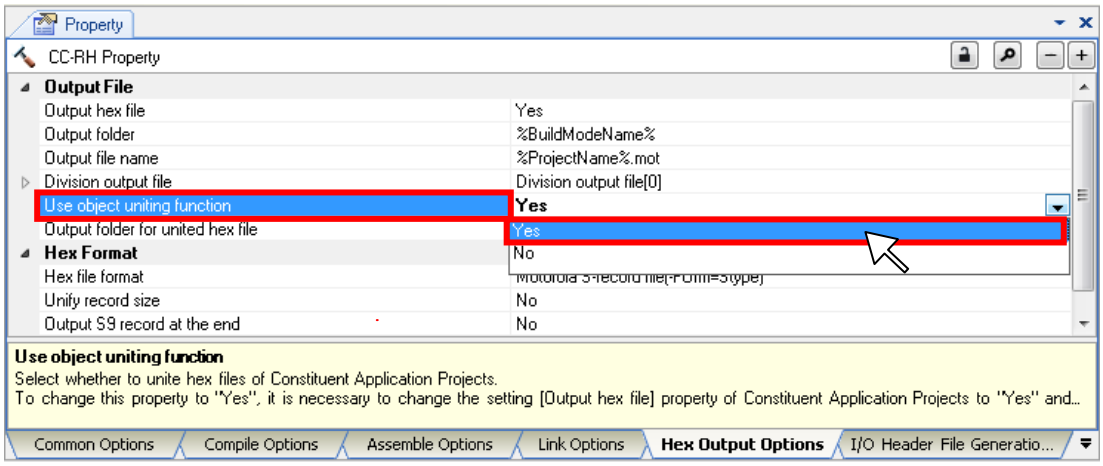
In the boot loader project and each application project, make settings for output of hex files and the hex format. Select "Yes" in [Hex Output Options] tab -> [Output File] category -> [Output hex file]. "Yes" is selected by default.



Next, select either "Intel HEX file" or "Motorola S-record file" in [Hex Output Options] tab -> [Hex Format] category -> [Hex file format]. Note that Intel HEX files and Motorola S-record files cannot be united. The hex format for the boot loader project and application projects must be the same.



Finally, specify the hex files to be united to form a single hex file in the boot loader project. Select "Yes" in [Hex Output Options] tab -> [Output File] category -> [Use object uniting function] of the boot loader project.



When the boot loader project is rebuilt, hex files of the boot loader project are united with hex files of the projects that have been specified as constituent application projects.
The united hex file is generated in [Output File] category -> [Output folder for united hex file]. The united hex file is generated in the "DefaultBuild_merged" folder by default.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation 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.
2. 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.
3. 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.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). 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 for which it is not intended. 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 which the product is not intended by Renesas Electronics.
6. 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.
7. 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 systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please 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.
9. 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. You should not use Renesas Electronics products or 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. When exporting the Renesas Electronics 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.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
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.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Ku, Seoul, 135-920, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 4.0