RH850 Family C Compiler Package (CC-RH)

PIC/PID Facilities

R20AN0505EJ0100
Rev.1.0
Aug. 06, 2018

Introduction

This application note gives an outline of the compiler's PIC/PID (position-independent code and data) facilities and describes how to use them with the aid of some examples.

Target Revisions

CC-RH V1.07.00 and later

Contents

1. PIC/PID Facilities

The PIC/PID facilities enable the relocation of code and data and their execution and handling at desired addresses without re-linkage even after their allocation addresses have been determined through previously completed linkage.

This application note describes how to use the PIC/PID facilities with an example where two types of program, referred to as "application program" and "master program", are created. The application program is to be allocated and executed at a desired address in memory through the PIC/PID facilities. The master program is used to execute the application program.

1.1 Outline of the Facilities

The CC-RH compiler provides the following three facilities.

PIC (position-independent code) facility

This enables the allocation of code (functions) to desired addresses in memory and executed from there.

The "-pic" option generates a position-independent section for the allocation of the functions.

PIROD (position-independent read-only data) facility

This enables the allocation of constant data (const variables) to desired addresses in memory and referred to there.

The "-pirod" option generates a position-independent section for the allocation of constant data.

PID (position-independent data) facility

This enables the allocation of data (variables) to desired addresses in memory and referred to there.

The "-pid" option generates a position-independent section for the allocation of the data.

1.2 Example of Usage

When the PIC facility is used, an updated application program can be allocated to a desired address and executed from there without affecting an earlier version of the application program that is already being executed.

Non-PIC shared routines such as standard library functions can also be called from the application program. In this case, however, a project for the shared routines should be built before the application program, and the application program should refer to the absolute addresses of the shared routines.

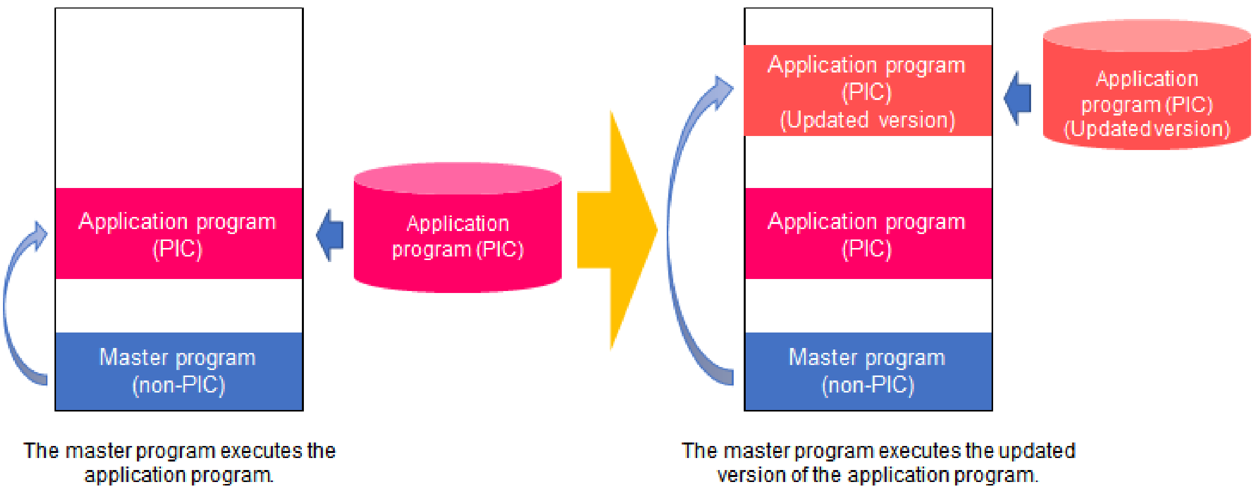

Figure 1-1 Example of PIC/PID Usage

2. PIC Facility

2.1 Compiler Option

The compiler option "-pic" enables the PIC facility.

Note that this option should be specified together with the "-pirod" option.

In the CS+ IDE, select the [Common Options] tab → [PIC/PID] category → [Enable PIC and PIROD functions] → [Yes(-pic -pirod)] to enable the PIC facility.

When the PIC facility is enabled in CS+, both the "-pic" and "-pirod" options are specified together.

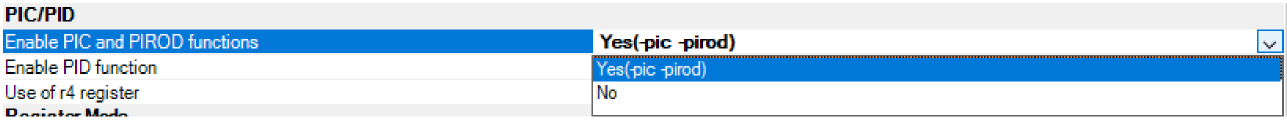

Figure 2-1 Specifying the -pic and -pirod Options

2.2 Section Used for the PIC Facility

Specifying the "-pic" option changes the default name of the section where code is to be allocated from ".text" to ".pctext".

When a function is allocated to the .text section, a call (execution) of the function or reference to the function address is in PC-relative mode or 32-bit r0 (address 0)-relative mode. As 32-bit r0-relative access is used, the code in the .text section is not position-independent. In contrast, when a function is allocated to the .pctext section, access to the function is always in PC-relative mode to ensure that the code is position-independent.

The section specification in the "-start" option (the option for specifying section addresses) should also be changed from ".text" to ".pctext". The address specified for the .pctext section with this option is used to determine the distance between PC-relative sections and therefore does not have to be a runtime address.

Table 2-1 Section Used for the PIC Facility

Section Relocation Attribute	Default Section Name	Access Mode	Alignment Value
pctext	.pctext	32-bit addresses relative to the PC	2

The section name can be changed by using the #pragma section directive.

In the following example, the section name is changed to "test.pctext".

```
#pragma section pctext "test"
void func(void) {    //  test.pctext
…
}
```

2.3 Examples of the Use of the PIC Facility

2.3.1 Calling a PIC Function from a Non-PIC Function

For an example of calling a PIC function from a non-PIC function, see section 6.1.3.

2.3.2 Calling a PIC Function from a PIC Function

In C source code, a PIC function can be called by its name from a PIC function in the same way as for an ordinary function.

C source example:

```
void func1(void) {
func2();
}
```

In the compiled code, PC-relative mode is used to call the function.

Result of compilation:

```
_func1:
        .stack   _func1 = 4
        prepare 0x00000001, 0x00000000
        jarl     _func2, r31
        dispose 0x00000000, 0x00000001, [r31]
```

2.3.3 Calling a Non-PIC Function from a PIC Function

In C source code, a non-PIC function can be called by its name from a PIC function in the same way as for an ordinary function.

C source example:

```
#pragma section text  // Section defined in the master program (non-PIC)
void nopic_func();
#pragma section default

void func1(){         // func1 is allocated to the .pctext section.
  nopic_func();       // Calls a non-PIC function.
}
```

The compiler generates the code for calling the absolute address of the non-PIC function in r0-relative mode. When the application program calls a function in the master program, the application program refers to the symbol address file (*.fsy) created for the master program to determine the address of the function. After execution of the non-PIC function, execution is returned to the PIC function by using the r31 register (LP).

Result of compilation:

```
_func1:
    .stack   _func1 = 4
    prepare 0x00000001, 0x00000000
    mov     #_nopic_func, r2   // Get absolute address of non-PIC function.
    jarl    [r2], r31
    dispose  0x00000000, 0x00000001, [r31]
```

When the "-pic" option is specified, note that functions can only be defined in sections having the pctext attribute.

3. PIROD Facility

3.1 Compiler Option

The compiler option "-pirod" enables the PIROD facility.

Note that this option should be specified together with the "-pic" option and cannot be specified together with "-Omap" or "-Osmap" option.

Refer to section 2.1 for information regarding how to specify this option in the CS+ IDE.

3.2 Section Used for the PIROD Facility

Specifying the "-pirod" option changes the default name of the section where constant data are to be allocated from ".const" to ".pcconst32".

When constant data are allocated to the .const section, reference to constants or their addresses is in 32-bit r0 (address 0)-relative mode. Therefore, the constant data in the .const section are not position-independent. In contrast, access to the data in the .pcconst32 section is always in PC-relative mode to ensure that the data are position-independent. Access from the PIC to the PIROD is based on the relative addresses determined at linkage, so the distances between them cannot be changed.

The section specification in the "-start" option (the option for specifying section addresses) should also be changed from ".const" to ".pcconst32". The address specified for the .pcconst32 section with this option is used to determine the distance between PC-relative sections and therefore does not have to be a runtime address.

Table 3-1 Section Used for the PIROD Facility

Section Relocation Attribute	Default Section Name	Access Mode	Alignment Value
pcconst32	.pcconst32	32-bit addresses relative to the __pc_data symbol	4

The section name can be changed by using the #pragma section directive.

In the following example, the section name is changed to "test.pcconst32".

```
#pragma section pcconst32 "test"
const int a = 1;   //  test.pcconst32
```

Instructions for reference to constant data can be shortened by changing the section for allocation to ".pcconst16" or ".pcconst23".

```
#pragma section pcconst16
const int a = 1;  //  .pcconst16
```

3.3 Example of the Use of the PIROD Facility

In C source code, reference to a constant handled as PIROD can be by its name in the same way as for an ordinary constant.

C source example:

```
const int a = 3;
int func() {
return a;
}
```

In the compiled code, PC-relative mode is used for reference to PIROD variables from PIC functions.

Result of compilation:

```
.public    _a, 4
.public    _func
.section   ".pctext", pctext
func:
     .stack   func = 0
     jarl     .BB.LABEL.1_1, r2  ; Sets r2 to the runtime address of .LABEL.1_1.

.BB.LABEL.1_1:
     mov      #.BB.LABEL.1_1-#__pc_data, r5   ; Relative address determined at
                                              ; linkage
     sub      r5, r2
     movhi    HIGHW1(#_a-#__pc_data), r2, r2  ; PC-relative reference to _a
     ld.w     LOWW(#_a-#__pc_data)[r2], r10
     jmp      [r31]
     .section ".pcconst32", pcconst32
     .align   4
a:
     .dw      0x00000003
```

__pc_data: Base symbol for PC-relative access, which is automatically generated by the linker.

4. PID Facility

4.1 Compiler Option

The compiler option "-pid" enables the PID facility.

Note that this option cannot be specified together with the "-r4=none", "-Omap", or "-Osmap" option.

In the CS+ IDE, select the [Common Options] tab → [PIC/PID] category → [Enable PID function] → [Yes(-pid)] to enable the PID facility.

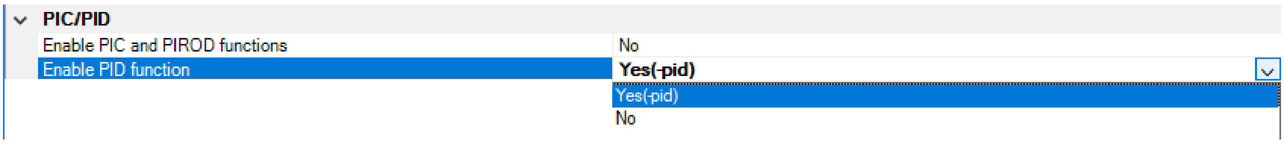

Figure 4-1 Specifying the –pid Option

4.2 Section Used for the PID Facility

Specifying the "-pid" option changes the default names of the sections where variables are to be allocated from ".data" and ".bss" to ".sdata32" and ".sbss32", respectively.

When variables are allocated to the .data or .bss section, reference to variables or their addresses is in 32-bit r0 (address 0)-relative mode. Therefore, the variables in the .data or .bss section are not position-independent. In contrast, access to the variables in the .sdata32 or .sbss32 section is always in GP-relative mode to ensure that they are position-independent.

Note that the .sdata32, .sbss32, .edata32, and .ebss32 sections are dedicated to the PID facility. By default, variables to be handled as PID are allocated to the .sdata32 or .sbss32 section and reference to them is always in GP-relative mode. When EP-relative access is specified by the #pragma section directive, variables are allocated to the .edata32 or .ebss32 section and referenced in EP-relative mode.

GP-relative sections and EP-relative sections other than those stated above can also be used for the PID facility. For the available sections, refer to the CC-RH Compiler User's Manual.

The section specifications in the "-start" option (the option for specifying section addresses) should also be changed from ".data" and ".bss" to ".sdata32" and ".sbss32", respectively. The addresses specified for the .sdata32 and .sbss32 sections with this option are used to determine the distances between the base symbols and the GP-relative or EP-relative sections and therefore they do not have to be runtime addresses.

Table 4-1 Sections Used for the PID Facility by Default

Section Relocation Attribute	Default Section Name	Variables to be Allocated	Access Mode	Alignment Value
sdata32	.sdata32	Initialized variables	32-bit addresses relative to r4 (GP)	4
sbss32	.sbss32	Uninitialized variables		

The section names can be changed by using the #pragma section directive.

In the following example, the section names are changed to "test.sdata32" and "test.sbss32".

```
#pragma section sdata32 "test"
int a = 1;    //  test.sdata32
int b ;       //   test.sbss32
```

4.3 Example of the Use of the PID Facility

In C source code, reference by a PIC function to a variable handled as PID can be by its name in the same way as for an ordinary variable.

C source example:

```
int a = 1;
int func() {
    return a;
}
```

In the compiled code, GP-relative or EP-relative mode is used for reference to the PID variable.

Result of compilation:

```
.public   _a, 4
.public   _func1
_func1:
     .stack    _func1 = 0
     movhi    HIGHW1($_a), r4, r2  // GP-relative reference
     ld.w     LOWW($_a)[r2], r10
     jmp      [r31]
.section .sdata32, sdata32
.align    4
_a:
     .dw 0x00000001
```

5. Startup Routine

When the PIC/PID facilities are enabled, the standard startup routine cannot be used. The following processes in the startup routine require modification.

- Initialization of base registers
- Initialization of RAM sections
- Branching to the main function

Sample code for a startup routine is given in the appendix. The following describes the modifications to processing in the sample code.

5.1 Initialization of Base Registers

When using the PID facility, determine the means of passing the information regarding how much a section is offset from the start address of the RAM section specified at linkage (hereafter referred to as **the RAM offset value**) in advance. For example, write the RAM offset value to a specific location in RAM or data flash memory*.

*: Since reference to the specific location has to be with an absolute address in this case, the PID or PIROD facility cannot be used for the location.

When restarting the program without shutting off the power supply of the microcontroller, store the RAM offset value in a specific register. Add the received RAM offset value to the base register values, and the resulting values are used as the base addresses at runtime.

```
$ifdef __PID
     mov    #_PID_offset, r28     ; Memory address for passing the RAM offset value
                                  ; The RAM offset value is stored in this address.
     ld.w   0[r28], r28           ; Stores the offset (RAM offset value) between
data
                                  ; allocation at linkage and data allocation at
runtime
                                  ; in the r28 register.
$endif
          ; When using both the GP and EP registers in the PIC
          mov    #_stacktop, sp           ; Sets up the SP register.
          mov    #__gp_data, gp           ; Sets up the GP register.
          mov    #__ep_data, ep           ; Sets up the EP register.
$ifdef __PID
          add    r28, sp     ; Prevents overlapping of data allocation areas and
                             ; stack area when GP and EP base addresses are offset.
                             ; This line can be omitted when overlapping never
occurs.
          add    r28, gp
          add    r28, ep
$endif
```

5.2 Initializing the RAM Sections

The _INITSCT_RH() function cannot be used for initializing sections for which the PID facility is enabled because the function receives and uses the section information tables. Therefore, the initial values should be directly copied from ROM to RAM within the startup routine. Obtain the offset between the address of the code and constant data area at linkage and that at runtime (hereafter referred to as **the ROM offset value**) in advance.

```
      jarl  .pic_base, r29     ; Stores the address of the .pic_base label at runtime in r29.
 .pic_base:
      mov   #.pic_base, r10    ; Stores the address of the pic_base label at linkage in r10.
      sub   r10, r29           ; The value (r29 – r10) is used as the ROM offset value.
```

Next, initialize the sections for the allocation of initialized data. To initialize a section, store the start and end addresses of the source area for copying initial values and the destination address for copying in the r6, r7, and r8 registers, respectively. (*1)

When using the PIROD facility, add the ROM offset value to the start address (r6 register value) and end address (r7 register value) of the source area for copying initial values. (*2)

When using the PID facility, add the RAM offset value to the destination address (r8 register value) where initial values are to be copied. (*3)

```
      mov   #__s.sdata32, r6   ; (*1) Stores the start address of the source area for copying.
      add   r29, r6            ; (*2) Adds the ROM offset value.

      mov   #__e.sdata32, r7   ; (*1) Stores the end address of the source area for copying.
      add   r29, r7            ; (*2) Adds the ROM offset value.

      mov   #__s.sdata32.R, r8  ; (*1) Stores the address of the destination area for copying.
      add   r28, r8             ; (*3) Adds the RAM offset value.
```

As preparation for copying is complete at this point, call the copying routine.

```
    jarl        _copy4, lp
....
    ; r6: Source begins (4-byte aligned)
    ; r7: Source ends (r6 <= r7)
    ; r8: Destination begins (4-byte aligned)
          .align     2
copy4:
    sub   r6, r7
.copy4.1:
    cmp   4, r7
    bl    .copy4.2
    ld.w  0[r6], r10
    st.w  r10, 0[r8]
    add   4, r6
    add   4, r8
    add   -4, r7
    br    .copy4.1
.copy4.2:
    cmp   2, r7
    bl    .copy4.3
    ld.h  0[r6], r10
    st.h  r10, 0[r8]
    add   2, r6
    add   2, r8
    add   -2, r7
.copy4.3:
    cmp   0, r7
    bz    .copy4.4
    ld.b  0[r6], r10
    st.b  r10, 0[r8]
.copy4.4:
    jmp    [lp]
```

Repeat these steps as many times as the number of sections that require initial values.

Next, initialize the sections for allocating uninitialized data with 0s. Store the start and end addresses of a target section in the r6 and r7 registers, respectively. When using the PID facility, add the RAM offset value to the start address (r6 register value) and end address (r7 register value).

```
$ifdef __PID
    mov   #__s.sbss32, r6
    mov   #__e.sbss32, r7
    add   r28, r6          ; Adds the RAM offset value.
    add   r28, r7          ; Adds the RAM offset value.
$else
    mov   #__s.sbss, r6
    mov   #__e.sbss, r7
$endif
```

Call the initialization routine to initialize the target section with 0s.

```
      jarl        _clear4, lp
 ....
      ; r6: Destination begins (4-byte aligned)
      ; r7: Destination ends (r6 <= r7)
 .align     2
 _clear4:
      sub   r6, r7
 .clear4.1:
      cmp   4, r7
      bl    .clear4.2
      st.w  r0, 0[r6]
      add   4, r6
      add   -4, r7
      br    .clear4.1
 .clear4.2:
      cmp   2, r7
      bl    .clear4.3
      st.h  r0, 0[r6]
      add   2, r6
      add   -2, r7
 .clear4.3:
      cmp   0, r7
      bz    .clear4.4
      st.b  r0, 0[r6]
 .clear4.4:
      jmp    [lp]
```

Repeat these steps as many times as the number of sections that require initialization.

5.3 Branch to the main Function

When using the PIC facility and branching to the main function with the FERET instruction, add the ROM offset value to the value that is to be stored in the FEPC register.

```
      mov   #_exit, lp                ; lp <- #_exit
      mov   #_main, r10
 $ifdef __PIC
      add   r29, lp                   ; Adds the ROM offset value.
      add   r29, r10                  ; Adds the ROM offset value.
 $endif
      ldsr  r10, 2, 0                 ; FEPC <- #_main

      feret                           ; Sets up the PSW and PC to start execution in the user mode.
```

6. Examples of Application of the PIC/PID Facilities

This section describes the method for creating CS+ projects using the PIC/PID facilities.

6.1 Configuring Projects that Include Use of the PIC/PID Facilities

The following describes how to create CS+ projects for calling an application program (PIC) from a master program (non-PIC).

6.1.1 Structure of the CS+ Projects

After creating the master project (non-PIC) in the same way as when creating an ordinary CS+ project, add the application project (PIC) as a subproject. The following description uses an example where an application program is configured as a single project. This means that the distances between the base address and individual runtime addresses in the application program are fixed.

The following shows the structure of projects in the CS+ project tree.

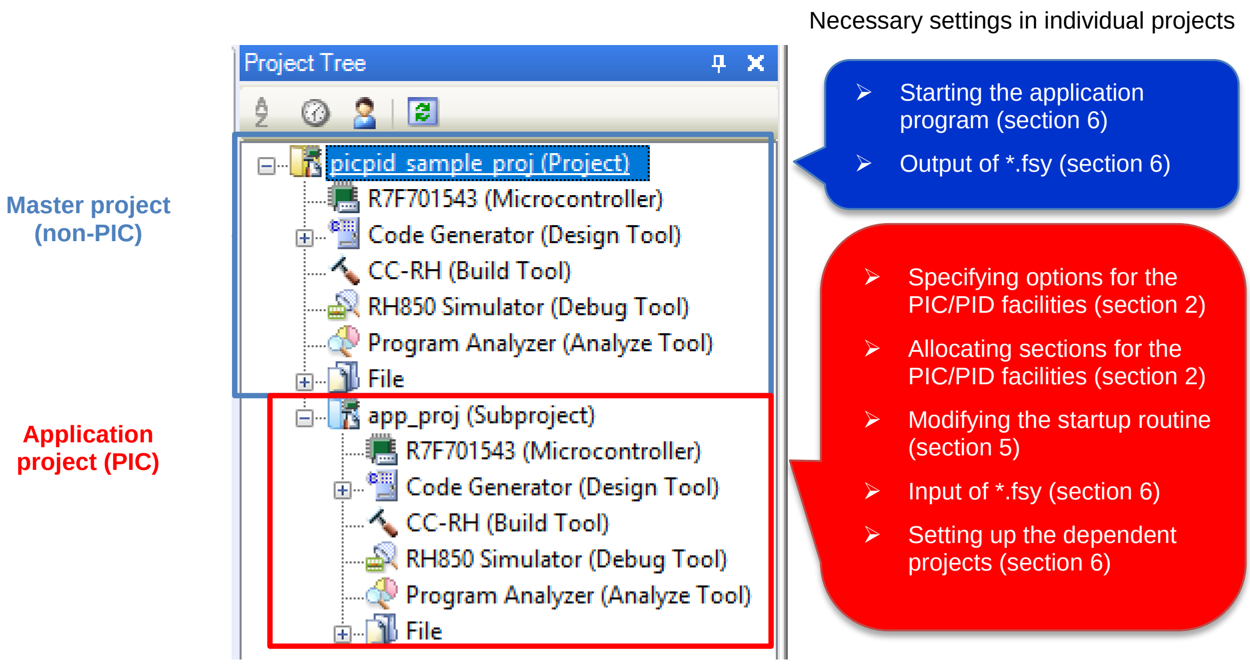

Figure 6-1 Structure of Projects

6.1.2 Creating the Master Project

Start up the CS+ IDE, click on the [Start] button on the toolbar, and click on the [GO] button for [Create New Project] in the [Start] panel to create a project.

6.1.3 Starting the Application Program from the Master Program

Start up the application program (PIC) from the master program by specifying the entry point for the PIC that was allocated at runtime.

To use the PID facility, the RAM offset value should be passed to the application program and used to initialize the GP and EP base registers (see section 5.1). The following description is based on the sample code given in the appendix.

The RAM offset value is stored at a specific address and passed to the application program through that address. Add "-start=PID_OFFSET.bss/<address for passing the offset>" to the linker option settings to allocate PID_offset to the address used for passing the offset.

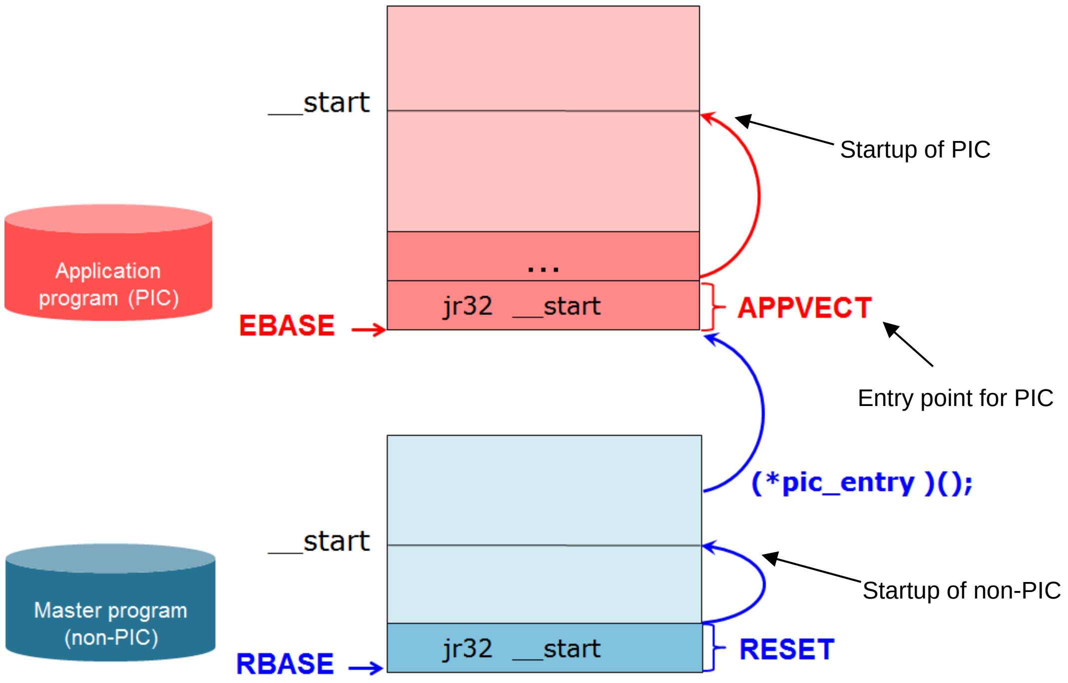

Figure 6-2 Branch from the Master Program to the Application Program in the Sample Code Given in the Appendix

C source example:

```
#pragma section PID_OFFSET
unsigned long PID_offset;                    // Location for storing the RAM offset value
#pragma section default

void main() {
  void (*pic_entry)(void) = (void*)<entry point at runtime>;// Address of APPVECT
  PID_offset = <RAM offset value at runtime>;
  (*pic_entry)();
}
```

The <entry point at runtime> should contain a program that dynamically obtains the first address of the area where the application program (PIC) is stored.

6.1.4 Adding an Application Project

Select the project node in the project tree and select [Add] → [Add New Subproject...] or [Add Subproject...] from the context menu to add an application project.

To create a new application project, refer to sections 2 to 5 regarding setting up options and sections and edit the startup routine.

6.1.5 Reference to the Master Program from the Application Program

(1) Reference to externally defined symbols in the master program

For reference to a function or variable in the master program from the application program, write the declaration of the function or variable and the processing to refer to the function or variable in the application program. The section where this declaration of the function or variable is allocated has to match the section where the function or variable is defined in the master program. The processing (function) for reference in the application program should be defined in a PIC section.

Example of C source code in the application program:

```
#pragma section text "comm"
extern void *nopic_func(void);  // Non-PIC function

#pragma section pctext          // To define the function as PIC, change
                                // the section relocation attribute back to that for PIC.
void pic_func(void) {
  nopic_func();
}
```

Building the application program requires information regarding externally defined symbols in the master program. In building the master program, output the addresses of functions or variables for which you desire reference from the application program to a symbol address file (*.fsy).

When building the master program, select the [Link Options] tab → [Section] category → [Section that outputs external defined symbols to the file], click on the [...] button on the right side, and specify the name of the section to which the externally defined symbols are to be allocated.

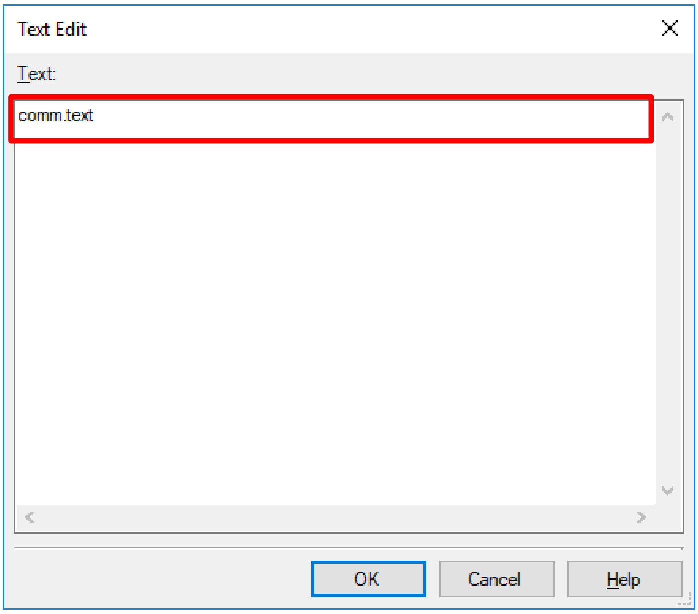

Figure 6-3 Dialog Box for Specifying the Section where Externally Defined Symbols are to be Allocated

Example of output to *.fsy:

```
;SECTION NAME = comm.text
.public _nopic_func
_nopic_func .equ 0xXXXXXXXX
```

Externally defined symbol name ↑ (points to _nopic_func)   Allocated address ↑ (points to 0xXXXXXXXX)

Next, register the *.fsy file with the application project. Right-click on the [File] node in the project tree and select [Add] to add the file.

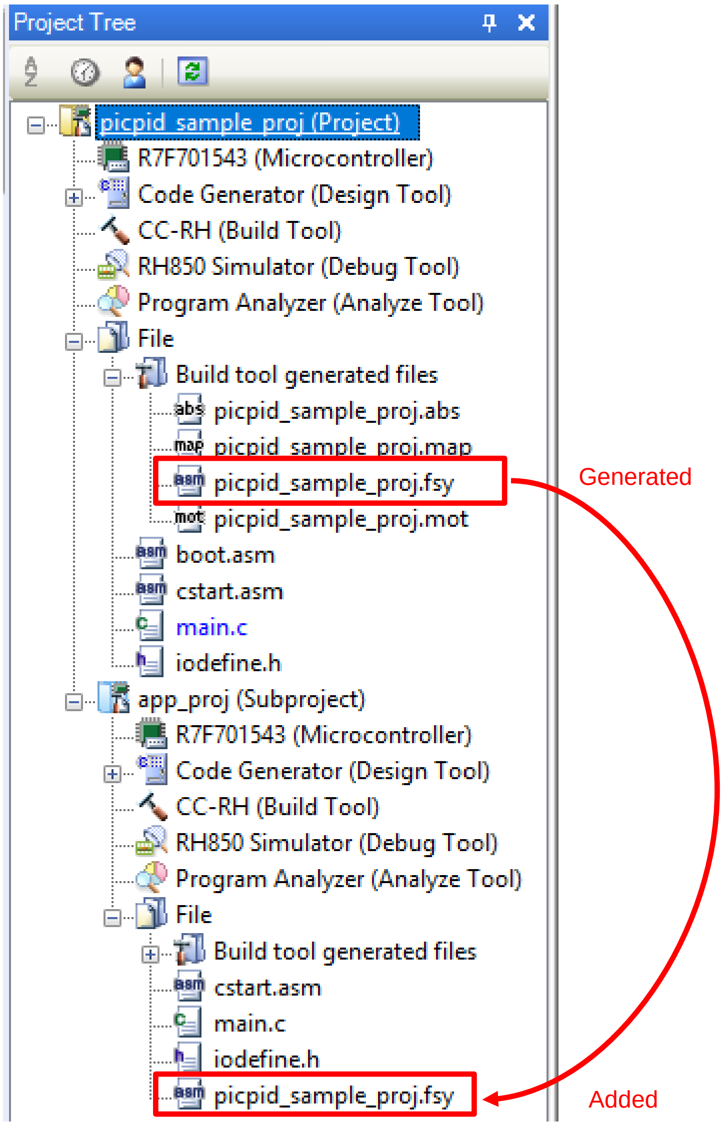

Figure 6-4 Project Tree After the *.fsy File has been Added to the Application Project

Supplementary note: For reference to a standard library used by the master program

To refer to a standard library from the application program, refer to the link map file for the master program and manually write the symbol names and addresses of the functions and variables in the *.fsy file.

C source example:

For reference to library functions used by the master program from the application program, write dummy code for referring to the functions as shown below so that the library functions are linked to the master program.

```
#include <string.h>
void* const dummy_libcall[] = {&memcpy, &memcmp, &strcpy};
```

Example of link map file output:

```
 SYMBOL                     ADDR         SIZE      INFO    COUNTS  OPT
FILE = memcpy
                            00002024  0000203b       18
 _memcpy
                            00002024         0   none ,g
```

Example of *.fsy contents:

```
;SECTION NAME = text
 .public _memcpy
_memcpy .equ 0x00002024
```

(2) Setting up the dependent projects

Reference by the application program to the master program requires building of the master project and application project in that order.

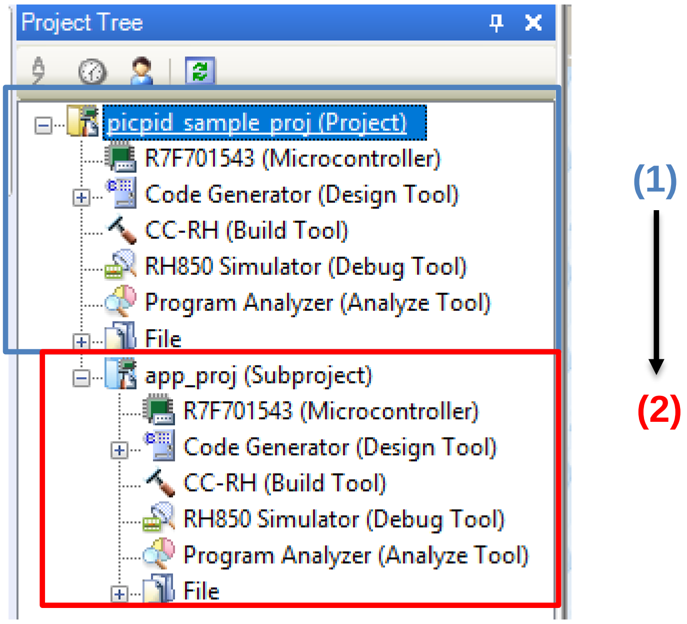

Figure 6-5 Order of Building Projects

In the CS+ IDE, the order of building projects can be controlled as desired. Select the [Project] menu → [Dependent Projects Settings] and specify the order in the [Dependent Projects Settings] dialog box.

With the settings shown in the following figure, the app_proj project depends on the picpid_sample_proj project and the projects are built in the order picpid_sample_proj then app_proj.

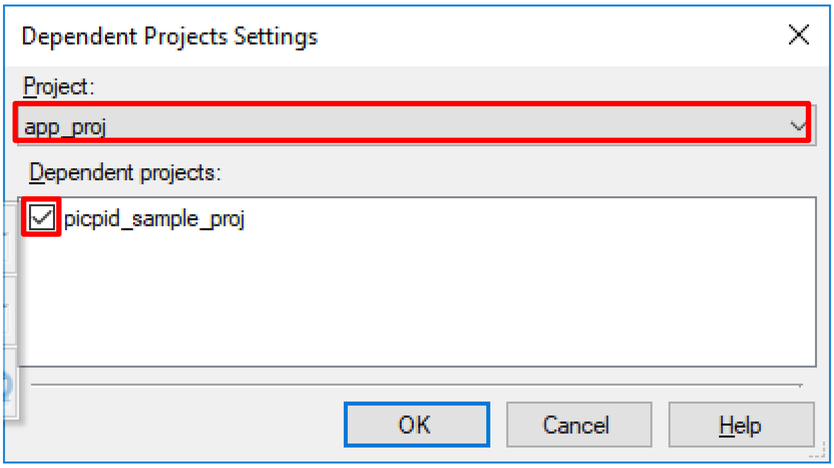

Figure 6-6 Setting up Dependent Projects

6.2 Making Interrupt and Exception Handlers Position-Independent

When the interrupt and exception handlers for the application program are made position-independent*, the address of each handler is the sum of the address specified in the EBASE register (base address at runtime for the allocation of handlers) and the offset for each exception or interrupt source. Specify the vector table address in the EBASE register and set the EVB bit in the PSW to 1. The section for the interrupt and exception handlers should be aligned with a 512-byte boundary.

*: The RESET vector cannot be position-independent. Instead of the RESET vector, specify the entry point to the application program.

Example startup routine:

```
$ifdef __PIC
; Specifies the vector table address in EBASE.
  mov       #__sAPPVECT, r10       ; Stores the entry point address of the PIC in r10.
  add       r29, r10               ; Adds the ROM offset value.
  ldsr      r10, 3, 1              ; EBASE <- r10
  stsr      5, r10, 0              ; r10 <- PSW
  mov       0x00008000, r11
  or        r11, r10
  ldsr      r10, 5, 0              ; Set PSW.EBV to 1.
```

Example vector table:

```
;--------------------------------------------------------
;    Exception vector table
;--------------------------------------------------------
     .section "APPVECT", pctext   ; Entry point for the PIC
     .align   512
     jr32     __start            ; Branches to the startup location of the PIC.

     .align   16
     jr32     _Dummy1            ; Interrupt or exception processing 1

     .align   16
     jr32     _Dummy2            ; Interrupt or exception processing 2
…
```

7. Points for Caution

Note the following points when using the PIC/PID facilities.

7.1 Reference to Variables and Functions

There are some restrictions on the mode for reference to the functions or variables of the master program (non-PIC) from the application program (PIC).

The following table shows the allowable combinations of referring and referred sides and modes of access between functions and variables in the application program and between the application program and master program.

Table 7-1 Allowable Combinations of Reference and Access Mode between Variables and Functions

		Referred Side					
		PIC Functions	Non- PIC Function	PIROD Variable	Non-PIROD Variable	PID Variable*2	Non-PID Variable
Referring Side	PIC Function	PC-relative	r0-relative	PC-relative	r0-relative	GP- or EP-relative	GP-, EP-, or r0-relative
	Non-PIC Function	Not allowed*1	PC- or r0-relative	Not allowed*1	r0-relative	GP- or EP-relative	GP-, EP-, or r0-relative

Notes: 1. When a non-PIC function is linked, no direct reference is possible because the linker cannot determine the addresses of PIC functions or PIROD variables at runtime. However, reference through a pointer received at runtime is possible.

2. "PID variable" does not refer to all variables allocated to the GP-relative or EP-relative sections but only to those variables that were compiled with the -pid option specified.

7.2 Acquisition of Static Addresses

Execution of code and access to data compiled with the PIC/PID facilities are at different addresses from those determined at linkage. Therefore, the addresses of the code and data cannot be specified as the initializers of static variables.

Attempting to compile the following code will lead to errors.

```
  const int c;
  int d = 0;

  // Specifying the address of a PIROD variable causes an error.
void* vp1 = &c;
  // Assigning a literal to a PIROD variable causes an error.
const char* cp const = "string";
  // Specifying the address of a PID variable causes an error.
void* vp2 = &d;
```

7.3 Use of GP-Relative and EP-Relative Sections

Data handled as PID and non-PID can both be allocated to the GP-relative and EP-relative sections. However, since the GP and EP registers are shared between the two sets of data, if the GP or EP register value is changed due to use of the PID facility, the addresses for reference to the non-PID are also changed. We recommend determining a coherent policy on whether to use each of the GP and EP registers for PID or non-PID throughout the program.

7.4 Use of Standard Libraries

The standard libraries do not support the PIC/PID facilities. The libraries should be linked to the master program.

7.5 Compiler Options

To inter-link the application program and master program, the following compiler options should be set to the same values for both programs.

Table 7-2 Options to be Set to the Same Values for the Application and Master Programs

Option	Description
-Xenum_type	Specifies in which integer type the enumeration type is handled.
-Xdbl_size	Specifies the data size of the double and long double types.
-Xpack	Performs the packing of structures.
-Xbit_order	Specifies the order of bit-field members.
-Xreg_mode	Specifies the register mode.
-Xreserve_r2	Reserves the r2 register.
-Xep	Specifies how to handle the EP register.
-Xfloat	Controls the generation of floating-point operation instructions.
-Xround	Specifies the mode for rounding floating-point constants.

Appendix

The following is sample code for the vector table and startup routine in the application program when the PIC/PID facilities are used.

```
;---------------------------------------------------------
;    Exception vector table
;---------------------------------------------------------
     .section "APPVECT", pctext   ; Entry point for the PIC
     .align   512
     jr32     __start

     .align   16
     jr32     _Dummy1             ; Interrupt or exception processing 1

     .align   16
     jr32     _Dummy2             ; Interrupt or exception processing 2
…
;---------------------------------------------------------
;    Startup
;---------------------------------------------------------
     .section    ".pctext", pctext
     .align      2
__start:                       ; Startup location of the PIC
     jr32   __cstart
```

```
$ifdef __PIC
  .TEXT .macro
    .section .pctext, pctext
  .endm
$else
  .TEXT .macro
    .section .text, text
  .endm
$endif


$ifdef __PID
  .STACK_BSS .macro
    .section .stack.bss, sbss32
  .endm
$else
  .STACK_BSS .macro
    .section .stack.bss, bss
  .endm
$endif

;-------------------------------------------------------------------------------
; System stack
;-------------------------------------------------------------------------------
STACKSIZE  .set   0x200
```

```
    .STACK_BSS
    .align    4
    .ds        (STACKSIZE)
    .align    4
_stacktop:

;------------------------------------------------------------------------------
;  Startup
;------------------------------------------------------------------------------
    .TEXT
    .public __cstart
    .align    2
__cstart:

$ifdef __PIC
    jarl      .pic_base, r29
.pic_base:
    mov       #.pic_base, r10
    sub       r10, r29
$endif

$ifdef __PID
    mov       0xfedf0000, r28         ; Memory address for passing the RAM offset
value.
    ld.w      0[r28], r28             ; Offset (RAM offset value) between data
allocation
                                      ; at linkage and data allocation at runtime.
$endif

    mov       #_stacktop, sp          ; Sets up the SP register.
    mov       #__gp_data, gp          ; Sets up the GP register.
    mov       #__ep_data, ep          ; Sets up the EP register.
$ifdef __PID
    add       r28, sp
    add       r28, gp
    add       r28, ep
$endif

    ; Initialize the .sdata32 section
$ifdef __PID
  $ifdef __PIROD
    mov       #__s.sdata32, r6
    add       r29, r6
    mov       #__e.sdata32, r7
    add       r29, r7
    mov       #__s.sdata32.R, r8
    add       r28, r8

  $else
    mov       #__s.sdata32, r6
    mov       #__e.sdata32, r7
    mov       #__s.sdata32.R, r8
    add       r28, r8
  $endif
$else
  $ifdef __PIROD
    mov       #__s.data, r6
    add       r29, r6
```

```
    mov       #__e.data, r7
    add       r29, r7
    mov       #__s.data.R, r8
  $else
    mov       #__s.data, r6
    mov       #__e.data, r7
    mov       #__s.data.R, r8
  $endif
$endif
    jarl      _copy4, lp

    ; Initialize the .sbss32 section.
$ifdef __PID
    mov       #__s.sbss32, r6
    mov       #__e.sbss32, r7
    add       r28, r6
    add       r28, r7
$else
    mov       #__s.bss, r6
    mov       #__e.bss, r7
$endif
    jarl      _clear4, lp

    ; Enable the FPU
$if 1 ; Disable this block when the FPU is not to be used.
    stsr      6, r10, 1               ; r10 <- PID
    shl       21, r10
    shr       30, r10
    bz        .L1                     ; Detects the FPU.
    stsr      5, r10, 0               ; r10 <- PSW
    movhi     0x0001, r0, r11
    or        r11, r10
    ldsr      r10, 5, 0               ; Enables the FPU.

    movhi     0x0002, r0, r11
    ldsr      r11, 6, 0               ; Initializes the FPSR.
    ldsr      r0, 7, 0                ; Initializes the FPEPC.
.L1:
$endif

    ; Set flags in PSW via FEPSW

    stsr      5, r10, 0               ; r10 <- PSW
    ;xori     0x0020, r10, r10        ; Enables interrupts.
    ;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

$ifdef __PIC
    add       r29, lp
    add       r29, r10
$endif
    ldsr      r10, 2, 0               ; FEPC <- #_main

    feret            ; Sets up the PSW and PC to start execution in the user mode.
```

```
_exit:
    br        _exit                 ;  End of program

;------------------------------------------------------------------------------
; Copy routine
;------------------------------------------------------------------------------
    ; r6: Source begins (4-byte aligned)
    ; r7: Source ends  (r6 <= r7)
    ; r8: Destination begins  (4-byte aligned)
    .align    2
_copy4:
    sub       r6, r7
.copy4.1:
    cmp       4, r7
    bl        .copy4.2
    ld.w      0[r6], r10
    st.w      r10, 0[r8]
    add       4, r6
    add       4, r8
    add       -4, r7
    br        .copy4.1
.copy4.2:
    cmp       2, r7
    bl        .copy4.3
    ld.h      0[r6], r10
    st.h      r10, 0[r8]
    add       2, r6
    add       2, r8
    add       -2, r7
.copy4.3:
  cmp         0, r7
  bz          .copy4.4
  ld.b        0[r6], r10
  st.b        r10, 0[r8]
.copy4.4:
  jmp         [lp]

;------------------------------------------------------------------------------
; Clear routine
;------------------------------------------------------------------------------
    ; r6: Destination begins (4-byte aligned)
    ; r7: Destination ends  (r6 <= r7)
    .align    2
_clear4:
    sub       r6, r7
.clear4.1:
    cmp       4, r7
    bl        .clear4.2
    st.w      r0, 0[r6]
    add       4, r6
    add       -4, r7
    br        .clear4.1

.clear4.2:
    cmp       2, r7
    bl        .clear4.3
    st.h      r0, 0[r6]
    add       2, r6
```

```
    add       -2, r7
.clear4.3:
    cmp       0, r7
    bz        .clear4.4
    st.b      r0, 0[r6]
.clear4.4:
    jmp       [lp]

;-----------------------------------------------------------------------------
; Dummy section
;-----------------------------------------------------------------------------
$ifdef __PID
    .section .sdata32, sdata32
.L.dummy.sdata32:
    .section .sbss32, sbss32
.L.dummy.sbss32:
$else
    .section .data, data
.L.dummy.data:
    .section .bss, bss
.L.dummy.bss:
$endif

$ifdef __PIROD
    .section .pcconst32, pcconst32
.L.dummy.pcconst32:
$else
    .section .const, const
.L.dummy.const:
$endif
;-------------------- End of startup module -------------------;
```

Website and Support

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

Inquiries
http://www.renesas.com/contact/

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Aug 06, 2018	-	First edition issued

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

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

California Eastern Laboratories, Inc.
4590 Patrick Henry Drive, Santa Clara, California 95054-1817, U.S.A.
Tel: +1-408-919-2500, Fax: +1-408-988-0279

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

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, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

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 1207, 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 India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 7.1