R-Car S4 Series
Guide to Porting FreeRTOS to the CR52 in the R-Car S4

Introduction
The R-Car S4 series incorporate multiple application cores and a realtime core. Porting a realtime OS to the realtime core (Cortex-R52; hereafter called the CR52) among the incorporated cores enables implementation of realtime processing by higher-level functions in the software hierarchy. This application note describes how to run FreeRTOS on the CR52.

Target Board
R-car S4 evaluation board (RTP8A779F0ASKB0SP2SA080 serial number 2023 to 2132)
We simply refer to it as “evaluation board” in this document.
Contents

1. Overview ........................................................................................................................................4
1.1 Purpose .......................................................................................................................................4

2. Contents of the Sample Program Package ....................................................................................4

3. Environment for Confirming Operation .........................................................................................6

4. Overview of FreeRTOS Porting ......................................................................................................7

5. Outline of Software Operation .......................................................................................................8

6. Details of the Software ..................................................................................................................9
6.1 Startup Module ...........................................................................................................................9
6.1.1 Setting up the MPU to Specify the Memory Regions .........................................................10
6.1.2 Setting the Timer Frequency ..............................................................................................12
6.1.3 Mode Transition from the EL2 to the EL1 Level .................................................................12
6.1.4 Setting the Stack Pointer and Vector Base Address for EL1 ............................................14
6.1.4.1 Exception Handling Vector Table ..............................................................................15
6.1.4.2 Supervisor Call ...........................................................................................................15
6.1.4.3 IRQ Interrupt ...............................................................................................................15
6.1.5 Setting the Device Region ..................................................................................................16
6.1.6 Invalidating the Instruction Cache Memory .......................................................................16
6.1.7 Invalidating the Data Cache Memory ................................................................................17
6.1.8 Enabling Cache Control ....................................................................................................19
6.2 Processor-Dependent Module ....................................................................................................20
6.3 Driver Module .............................................................................................................................21
6.3.1 Setting up the Tick Timer ..................................................................................................22
6.3.2 IRQ Interrupt .....................................................................................................................24
6.3.3 Clearing the Tick Timer Interrupt ....................................................................................25
6.4 Application Module ...................................................................................................................26
6.4.1 Initializing the GIC Interrupt Facilities .............................................................................27
6.4.1.1 Overview of GIC Facilities .........................................................................................27
6.4.1.2 Registers of the GIC ...................................................................................................28
6.4.1.3 Outline of Initialization ...............................................................................................28
6.4.1.4 Initializing the Distributor ............................................................................................29
6.4.1.5 Initializing a CPU Interface .........................................................................................30
6.4.1.6 Activating a Redistributor ..........................................................................................30
6.4.2 Starting Task Processing .......................................................................................................31

7. How to Build from the Source Code .............................................................................................32
7.1 Procedure of Building ...............................................................................................................32
7.2 Executing the Sample Program ..................................................................................................33
7.2.1 Setting up the Terminal Software .....................................................................................33
7.2.2 Execution through the Debugger ........................................................................................33
7.2.2.1 Connecting a Debugger ..............................................................................................33
7.2.2.2 Operating the Debugger ..............................................................................................34
The following registered trademarks and trademarks of third parties, as well as company names, system names, and product names mentioned in this document are generally registered trademarks or trademarks of their respective companies. ™ mark and ® mark for companies trademark or registered trademark are omitted in this document.

- Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. ([https://www.arm.com/ja/company/policies/trademarks](https://www.arm.com/ja/company/policies/trademarks))
- FreeRTOS is a trademark of Amazon Web Services, Inc. ([https://freertos.org/copyright.html](https://freertos.org/copyright.html))
1. Overview

1.1 Purpose

This application note gives an example of porting FreeRTOS to the realtime core (CR52) incorporated in the R-Car S4 and supports the user in setting up the realtime core and developing user applications.

2. Contents of the Sample Program Package

This section explains the R-Car S4 CR52 FreeRTOS sample program package. Download it from the following page of the Renesas official web site.

Renesas Official Web site

It contains the modules framed by red rectangles in the following figure.

![Figure 1 R-Car S4 CR52 FreeRTOS Package](image)

The following is a list of modules the package contains.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>CR52 activation processing</td>
<td>startup</td>
</tr>
<tr>
<td>Processor-dependent</td>
<td>Parts of FreeRTOS processing that depend on the CPU</td>
<td>portable</td>
</tr>
<tr>
<td>Driver</td>
<td>Drivers for serial communications and timer operation for generating ticks</td>
<td>driver</td>
</tr>
<tr>
<td>Application</td>
<td>Sample program including main</td>
<td>src</td>
</tr>
</tbody>
</table>
The folders of this package are structured as follows.

```
|-- arm/         Interrupt vector table and region settings
|-- config/     Basic FreeRTOS settings
|-- driver/     Heap processing, serial driver, and timer driver ....  Drivers
|-- include/    Header files
|-- `-- cfg/     Compiler-dependent settings
|-- portable/
| `-- GCC/
|   |-- RCarS4_CR52/ CPU-dependent FreeRTOS processing ................. Processor-dependent
|-- src/        Sample program ................................................ Application
|-- startup/    CR52 activation processing ................................. Startup
standalone.ld  Linker script
```

*Figure 2  Structure of Folders in the Package*
3. Environment for Confirming Operation

The following lists the items of the operating environment in which operation of the FreeRTOS sample program was confirmed.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>CR52 incorporated in the R-Car S4. Operating frequency: 1066 MHz</td>
</tr>
<tr>
<td>Evaluation board</td>
<td>R-Car S4 evaluation board</td>
</tr>
<tr>
<td>LED</td>
<td>LED for 1.8-V to 2.0-V operation</td>
</tr>
<tr>
<td>Boot loader</td>
<td>ICUMXA Loader Rev.0.5.0</td>
</tr>
<tr>
<td>Debugger</td>
<td>Partner-Jet2 from Kyoto Microcomputer Co., Ltd.</td>
</tr>
</tbody>
</table>
| Development environment| OS: Ubuntu 18.04 LTS  
Compiler: gcc version 10.3.1 (GNU Arm Embedded Toolchain) |

The following shows the connection between the evaluation board and a Windows PC (for the use of a serial terminal).

![Connection Diagram](image)

Set the switches SW1, SW2, and SW8 as shown below.

![Settings of SW1, SW2, and SW8](image)

The Windows PC (for the use of a serial terminal) is not always necessary but it is useful in the development and debugging of software. For the output from the serial port, see section 8, Restrictions.
4. Overview of FreeRTOS Porting

The size of the FreeRTOS program can be reduced by using the FreeRTOSConfig.h file to select the functions to be ported and to control the sizes of the stack, heap, etc. The FreeRTOSConfig.h file this sample program package contains is optimized for general use, so modify the selection of functions or the control of memory sizes as required. For detailed definitions in the FreeRTOSConfig.h file, refer to the Customisation page on the FreeRTOS Web site.

The sample program package contains the source code for the CR52 startup processing, the tick timer and CR52 processor-dependent processing that are required to run FreeRTOS, and a sample FreeRTOS application. Link these sets of code and the FreeRTOS code downloaded from the FreeRTOS Web site to generate an executable module. Note that the downloaded FreeRTOS contains various processor-dependent modules but does not include the CR52 processor-dependent module. Therefore, this sample program package provides the source code of the CR52 processor-dependent module. For details, see section 6.2, Processor-Dependent Module. The generated executable module can be executed by loading it to the R-Car S4 through the debugger or through the flash ROM on the board.
5. Outline of Software Operation

The startup processing after release from the reset state executes the initialization processing that is required for the operation of application programs. After the startup processing, execution branches to the beginning (the main function) of the user application module. The application module enables interrupts, initializes the serial port, generates FreeRTOS tasks, and activates the task scheduler. This sample program generates two tasks; one of which (task A) periodically sends a message to the other task (task B), which turns the LED on or off upon receiving the message. The following figure is a flowchart of operation from the startup processing to running the application.

![Flowchart of the Application Program](image-url)
6. Details of the Software
This section describes the details of the individual modules of this sample program.

6.1 Startup Module
The startup module is the first to be executed after a reset. This module mainly executes the following processes: setup of the stack pointer, setup of the MPU, modification of the exception level, setup of the memory regions to be used, and control of the cache. The CR52 based on the ARMv8-R architecture is activated at exception level EL2 for supporting virtual machines implemented in hardware, which the ARMv7-R architecture does not provide. This sample program does not support virtual machines and therefore runs at a lower exception level, EL1. The startup module makes the basic settings that are necessary for the operation of the CR52 regardless of whether FreeRTOS is to be used, so it can be applied to many other applications. The following figure is a flowchart of the processing by the startup module. The steps marked with * are described in detail in the sections listed below the flowchart.

![Flowchart of Processing by the Startup Module](image)

Figure 6  Flowchart of Processing by the Startup Module
### 6.1.1 Setting up the MPU to Specify the Memory Regions

The CR52 has a memory protection unit (MPU) to protect the memory region against program execution going out of control. The memory regions are not protected immediately after release from the reset state because the MPU is disabled. Therefore, the memory regions other than for program execution should firstly be protected and then the protection of the device region should be canceled. For details, see section 6.1.5, Setting the Device Region. If code execution from an address beyond the specified region is attempted, a prefetch abort exception will occur. If read or write access to such an address is attempted, an abort exception will occur. The following describes the processing to set up the MPU to specify the memory regions.

#### Table: MPU Attribute Settings

<table>
<thead>
<tr>
<th>Address</th>
<th>Attribute Setting</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>nG nR nE</td>
<td>Normal cache</td>
<td>nG, G, nR, R, E</td>
</tr>
<tr>
<td>0000b</td>
<td>nG nR</td>
<td>Normal non-cacheable memory</td>
<td>nG, G, nR, R</td>
</tr>
<tr>
<td>0100b</td>
<td>nR</td>
<td>Normal memory (L1 write-back non-transparent)</td>
<td>nR, R</td>
</tr>
<tr>
<td>0100b</td>
<td>G, R, E</td>
<td>Normal memory (L1 write-through non-transparent)</td>
<td>G, R, E</td>
</tr>
<tr>
<td>1100b</td>
<td></td>
<td>Normal memory (L1 write-through transient)</td>
<td></td>
</tr>
<tr>
<td>1000b</td>
<td>nG R E</td>
<td>Normal memory (L2 write-back non-transparent)</td>
<td>nG, R, E</td>
</tr>
<tr>
<td>0000b</td>
<td></td>
<td>Normal memory (L2 write-through transient)</td>
<td></td>
</tr>
<tr>
<td>0000b</td>
<td></td>
<td>Normal memory (L2 write-through non-transparent)</td>
<td></td>
</tr>
<tr>
<td>0000b</td>
<td></td>
<td>Normal memory (L2 write-through non-transparent)</td>
<td></td>
</tr>
<tr>
<td>1000b</td>
<td></td>
<td>Normal memory (L2 write-through transient)</td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td></td>
<td>Normal memory (L2 write-through transient)</td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td></td>
<td>Normal memory (L2 write-through non-transparent)</td>
<td></td>
</tr>
</tbody>
</table>

#### Code Sample

```
mov r0, #0x6F00
mcr p15, #0, r0, c10, c2, #0
```

Specify attributes for access to memory regions. Eight attribute settings, Attr0 to Attr7, can be made. The MAIR0 register (32 bits) is used to specify four of them (Attr0 to Attr3). The remaining four attribute settings (Attr4 to Attr7) are made in the MAIR1 register. Each 32-bit register is divided into four fields; each Attr(n) consists of eight bits. Specify either of the following values unless a special setting is necessary.

- Device register: 0000 0000b
  - This sample program specifies the Attr1 attribute.
  - [7:4] 0000b Device memory
  - 00RWb Normal memory, write-through transient
  - 0100b Normal memory, L2 non-cacheable
  - 01RWb Normal memory, L2 write-through transient
  - 10RWb Normal memory, L2 write-through non-transparent
  - 11RWb Normal memory, L2 write-back non-transparent

- [3:0] (When bits 7 to 4 are 0000b)
  - 0000b nG nR nE
  - 0100b nG R E
  - 1100b G R E

Any pattern other than the above causes undefined operation. The meanings of nG, nR, nE, G, R, and E are described later.

- [3:0] (When bits 7 to 4 are not 0000b)
  - 0000b Undefined operation
  - 00RWb Normal memory, L1 write-through transient
  - 0100b Normal memory, L1 non-cacheable
  - 01RWb Normal memory, L1 write-back transient
  - 10RWb Normal memory, L1 write-through non-transparent
  - 11RWb Normal memory, L1 write-back non-transparent

Setting “RW” to 00 is prohibited except for the patterns marked with * in the above list.

- R: Read-Allocate: 1 = Enabled. 0 = Disabled.
- W: Write-Allocate: 1 = Enabled. 0 = Disabled.

#### Enabling or Disabling Read- or Write-Allocate

If a cache miss occurs (the target address is not found in the cache) during a write operation, data are directly written to the memory when “Write-Allocate” is disabled whereas the block at the target address is first fetched from memory to the cache and then data are written to the cache when Write-Allocate is enabled. In the same way, if a cache miss occurs during a read operation, data are directly read from memory when “Read-Allocate” is disabled whereas the block at the target address is first allocated in the cache and then data are read from memory and stored in the cache.

- nG, nR, nE, and E in the Device Region Settings
  - nG: Non-Gathering
  - G: Gathering
    - When “Non-Gathering” is specified, access by the processor to the bus is according to the number of times and size of access specified in the code. When “Gathering” is specified, the processor optimizes access. For example, two 1-byte write operations may be merged into a single 16-bit write operation.
  - nR: Non-Reordering
  - R: Reordering
    - When “Non-Reordering” is specified, access by the processor to the bus is in the order specified in the code. When “Reordering” is specified, the processor optimizes access and may change the order of access where this is possible.
  - nE: Non Early Write Acknowledgement
  - E: Early Write Acknowledgement
    - When “Non Early Write Acknowledgement” is specified, the processor waits for the response indicating write completion sent from peripherals over the bus. When “Early Write Acknowledgement” is specified, the processor proceeds with the next processing without waiting for a response.

---

Figure 7 Details of MPU Setup for Specifying the Memory Regions
Up to 24 memory regions can be specified through the MPU.

Use the protection region selection registers (PRSELR) to specify the target memory region numbers. The memory region settings for the other registers in the MPU are applied to the last memory region specified in the PRSELR. In the example shown to the left, the 0-th memory region is specified.

Specify the lower limit of memory addresses to be used, cache shareability, and access permission in the given PRBARn register.

- **[31:6]** Specify the lower limit of memory addresses with bits 5 to 0 set to 000000. This sample program specifies e210 0000h as the address “Start”.
- **[5]** Reserved
- **[4:3]** Cache shareability
  - The sample program specifies 00 (non-shareable) because the CPU has a single core.
- **[2:1]** Access permission
  - 00: Read/write
  - 01: Read-only
  - The sample program specifies 01 (readable and writable).
- **[0]** Execute-never flag
  - 0 = Executable, 1 = Not executable.
  - This is set to 0 because the memory region includes an instruction region.

Specify the upper limit of memory addresses to be used and memory attribute index in the given PRLARn register.

- **[31:6]** Specify the upper limit of memory addresses with bits 5 to 0 set to 000000. This sample program specifies the address of the 8-Mbyte range starting at e210 0000h (Start).
- **[5:4]** Reserved
- **[3:1]** Memory attribute index
  - The sample program specifies 001, which indicates Attr1 that has been set up in the MAIR0 register.
- **[0]** Enable flag
  - Set to 1 (enabling the use of the region).

Use the system control register (SCTLR) to enable the MPU settings at exception level EL1.

- **[31:1]** Description omitted
- **[0]** EL1-controlled MPU enable.
  - The sample program sets this bit to 1 because memory regions for peripherals should only be set up after the exception level is changed from EL2 to EL1.
6.1.2 Setting the Timer Frequency

Specify the clock frequency for the local timer in the CR52. The following shows an example of setting the timer frequency.

As the frequency of the clock input to the timer is not automatically detected, specify it through software. Note that this value should be the actual frequency input to the timer because this timer does not have a clock generator or a frequency divider. The CNTFRQ register can only be set up at the highest exception level. If setting this register is attempted at a lower exception level, an undefined instruction exception will occur. Therefore, set up this register before the transition to the EL1 level. A 16.6-MHz clock is input to the timer of the CR52 in the R-Car S4; convert 16,600,000 to a hexadecimal value and specify it in this register.

Set the CNTFRQ register.

```
movw r0, #0x4bc0
movt r0, #0x00fd
mcr p15, 0, r0, c14, c0, 0
```

![Figure 9 Details of Timer Frequency Setting](image)

6.1.3 Mode Transition from the EL2 to the EL1 Level

The CR52 has three exception levels and eight processor modes. Privileges of memory access and instruction execution are controlled according to the exception level. The following lists the correspondences between the exception levels and processor modes.

<table>
<thead>
<tr>
<th>Exception Level</th>
<th>Processor Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>User</td>
<td>Lowest privilege level</td>
</tr>
<tr>
<td>EL1</td>
<td>System, FIQ, IRQ, supervisor, abort, and undefined</td>
<td>For use with an OS</td>
</tr>
<tr>
<td>EL2</td>
<td>Hypervisor</td>
<td>Support for virtual machines</td>
</tr>
</tbody>
</table>

If an exception occurs, the values of the program counter (PC) and current program status register (CPSR) are automatically saved in the link register (LR) and saved program status register (SPSR). Note that separate LR and SPSR are provided for the individual processor modes. When the processor returns to a previous processor mode, the LR and SPSR for that processor mode are used to provide the values. The following describes the processing of a transition from the hypervisor mode (exception level EL2) to the supervisor mode (exception level EL1).
To change the exception level, specify the target processor mode to be used in the mode[4:0] bits of the saved program status register (SPSR) and execute the instruction for returning from exception handling. The return instruction changes the exception level from EL2 to EL1 according to the mode value specified in the SPSR. The example shown to the left obtains the value of the current program status register (CPSR), changes bits 4 to 0 to 0x13 (supervisor mode), and sets this value in the SPSR.

Use the vector base address register (VBAR) to specify the EL2 vector table address.

Set the VBAR register.

```assembly
ldr r0, =EL2_Vectors
mcr p15, 0, r0, c12, c0, 0
```

[31:5] Specify an address on a 32-byte boundary (that is, set bits 4 to 0 to 00000). The sample program specifies e210 0000h as the address “EL2_Vectors”.

[4:0] Reserved

Execute the return-from-exception instruction.

```assembly
ldr r0, =EL1
msr elr_hyp, r0
dsb
isb
ere
```

Specify the address of the instruction to be executed after the return from exception handling in the exception link register (ELR) provided for the given exception level. In the example shown to the left, the address of label “EL1:” is specified in the ELR for the hypervisor mode.

Execute the dsb instruction to cause a wait until all memory access operations are completed and then the isb instruction to flush the instruction pipeline. Execute the eret instruction to return from exception handling, and the processor will be placed in the processor mode specified in the mode[4:0] bits of the SPSR — that is, the supervisor mode at exception level EL1 — and then the processor will continue execution from the address specified for “EL1:” in the ELR.

---

**Figure 10** Details of Mode Transition from the EL2 to the EL1 Level
6.1.4 Setting the Stack Pointer and Vector Base Address for EL1

The stack pointer is not set up immediately after the exception level is changed from EL2 to EL1 because the stack pointers are switched every time the processor mode changes. The following shows an example of setting the stack pointer (SP) and specifying the base address of the vector table in the vector base address register (VBAR).

Set the stack pointer for SVC.

```
ldr r0, .svc_stack
mov sp, r0
```

Specify the stack pointer for the supervisor mode. Memory allocation for ".svc_stack" shown to the left should be determined by a linker script.

Set the VBAR register.

```
ldr r0, =EL1_Vectors
mcr p15, 0, r0, c12, c0, 0
```

Use the vector base address register (VBAR) to specify the address of the EL1 exception handling vector table. For the exception handling vector table, see section 6.1.4.1, Exception Handling Vector Table. Specify an address on a 32-byte boundary (that is, set bits 4 to 0 to 00000). The sample program specifies e210 0020h as the address "EL1_Vectors". [31:5] Reserved

Set the ICC_IGRPEN1 register.

```
mov r0, #0
mrc p15, 0, r0, c12, c12, 7
```

Disable interrupts. Set the ICC_IGRPEN1 register to 0 to disable all interrupts at exception level EL1.

```
msr cpsr_c, {mode value}
ldr r0, {address}
mov sp, r0
```

The CR52 has six processor modes at exception level EL1: system, FIQ, IRQ, supervisor, abort, and undefined modes. The stack pointer for the supervisor mode was specified in the first step in this figure, so the stack pointers for the remaining five modes then require specification.

In the example shown to the left, the processor modes are changed by the "msr cpsr_c" instruction (cpsr_c is bits 7 to 0 (the control field mask byte) of CPSR) and the stack pointer for the specified mode is then set by the "mov sp, r0" instruction. This processing should be repeated five times to specify the stack pointers for all five processor modes at exception level EL1. For {address} in the example, specify a label determined in the linker script to indicate the location of each stack. The following is a list of the correspondences between the processor modes and mode values.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Mode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
<td>13h</td>
</tr>
<tr>
<td>FIQ interrupt</td>
<td>11h</td>
</tr>
<tr>
<td>IRQ interrupt</td>
<td>12h</td>
</tr>
<tr>
<td>Abort</td>
<td>17h</td>
</tr>
<tr>
<td>Undefined</td>
<td>1bh</td>
</tr>
<tr>
<td>System</td>
<td>1fh</td>
</tr>
</tbody>
</table>

Note that this sample program actually uses a C language function (handler_reset in startup.c) for this processing but the results of the processing are the same as those of the assembly-language code written here.

Figure 11 Details of Setting the Stack Pointer and Vector Base Address for EL1
6.1.4.1 Exception Handling Vector Table

Eight types of exception handling can be specified for the CR52 processor and a 32-byte region is used for this purpose. A branch instruction to each exception handling routine should be specified in the exception handling vector table. The base address of the exception handling vector table can be modified through the VBAR register. The following table lists the contents of the exception handling vector table in this sample program. Modify the contents as required.

Table 4 Exception Vector Table for EL2

<table>
<thead>
<tr>
<th>Exception</th>
<th>Offset Address</th>
<th>Sample Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>00h</td>
<td>Branch to the reset processing (start.s)</td>
</tr>
<tr>
<td>Undefined instruction exception</td>
<td>04h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Hypervisor call</td>
<td>08h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Prefetch abort</td>
<td>0Ch</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Data abort</td>
<td>10h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Reserved</td>
<td>14h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>IRQ interrupt</td>
<td>18h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>FIQ interrupt</td>
<td>1Ch</td>
<td>Infinite loop (startup.c)</td>
</tr>
</tbody>
</table>

Table 5 Exception Vector Table for EL1

<table>
<thead>
<tr>
<th>Exception</th>
<th>Offset Address</th>
<th>Sample Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>00h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Undefined instruction exception</td>
<td>04h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Supervisor call</td>
<td>08h</td>
<td>Switching of FreeRTOS tasks (portASM.S)</td>
</tr>
<tr>
<td>See section 6.1.4.2, Supervisor Call.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetch abort</td>
<td>0Ch</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Data abort</td>
<td>10h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>Reserved</td>
<td>14h</td>
<td>Infinite loop (startup.c)</td>
</tr>
<tr>
<td>IRQ interrupt</td>
<td>18h</td>
<td>Interrupt processing (portASM.S)</td>
</tr>
<tr>
<td>See section 6.1.4.3, IRQ Interrupt.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIQ interrupt</td>
<td>1Ch</td>
<td>Infinite loop (startup.c)</td>
</tr>
</tbody>
</table>

6.1.4.2 Supervisor Call

A supervisor call exception occurs when a software interrupt (SVC) instruction is executed. In this case, the instruction stored at the offset address 08h of the EL1 exception vector table is executed. In this sample program, an instruction for branching to the FreeRTOS_SVC_Handler (portASM.S) function is stored at this offset address and tasks are switched if this is required. When the internal processing of a FreeRTOS system call requires task switching, the "\_asm ("SVC  0 \n");" instruction defined in portmacro.h is executed to generate a supervisor call exception.

6.1.4.3 IRQ Interrupt

The instruction stored at the offset address 18h of the EL1 exception vector table is executed if an interrupt occurs. In this sample program, an instruction for branching to the FreeRTOS_IRQ_Handler (portASM.S) function is stored at this offset address. The FreeRTOS_IRQ_Handler function calls the specific interrupt handler registered for the originating peripheral module. In the sample program, only the timer interrupt handler is registered. To register an interrupt handler, use the Interrupt_InstallHandler function in Interrupt.c. For details, see section 6.3.1, Setting up the Tick Timer.
### 6.1.5 Setting the Device Region

Specify the address range of the device region through the MPU in a similar way to that described in section 6.1.1, Setting up the MPU to Specify the Memory Regions. Be sure to specify the device region as non-executable and non-cacheable. The following table shows the settings of the device region together with the region specified in section 6.1.1, Setting up the MPU to Specify the Memory Regions.

<table>
<thead>
<tr>
<th>Region</th>
<th>Address</th>
<th>Size</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory region</td>
<td>E210 0000H to E28F FFFFH</td>
<td>8 Mbytes</td>
<td>Region 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Memory, cacheable, executable</td>
</tr>
<tr>
<td>Device region</td>
<td>E605 0000H to FFFF FFFFH</td>
<td>Approx. 416 Mbytes</td>
<td>Region 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Device, non-cacheable, non-executable</td>
</tr>
</tbody>
</table>

In this sample program, the code for setting of the memory regions is written in assembly language as described in section 6.1.1, Setting up the MPU to Specify the Memory Regions, but the code for setting of the device region is written in the C language (EL1_MPU_Setup function in arm_cr_mpu.h).

### 6.1.6 Invalidating the Instruction Cache Memory

Activating the application program leads to cache control being enabled, but before cache control is enabled, the instruction and data caches should be invalidated. The following shows an example of the procedure for invalidating the instruction cache. In this sample program, the processing for invalidating the instruction cache is written in the C language (L1C_InvalidCacheAll function in startup.c).

- **Set the ICIALLU register.**
  
  ```
  mov r0, #0
  mrc p15, 0, r0, c7, c5, 0
  ```

- **Invalidate the entire L1 instruction cache.** The parameter specified for the ICIALLU register is ignored.

- **dsb**

- **isb**

  *Execute the dsb instruction to cause a wait until all memory access operations are completed.*
  *The isb instruction flushes the instruction pipeline so that instruction code is again fetched from the main memory.*

**Figure 12** Details of Invalidating the Instruction Cache Memory
6.1.7 Invalidating the Data Cache Memory

As the data cache memory is not invalidated after a reset, it requires invalidation by a program.

The CR52 only has an L1 cache, but the following shows an example of general processing for invalidating the data cache, which can be used regardless of the core type.

---

Read the CLIDR register.

\[
\text{mrc p15, 1, r0, c0, c0, 1} \\
\text{clidr} = [r0]);
\]

for(i = 0; i < 7; i++) { 
ctype = (clidr >> i*3)&7;
if ((ctype & 2) != 0)
 invalidate_dcache(i);
}

---

Read the cache level ID register (CLIDR) to check the cache level of each cache incorporated in the CPU. This information is necessary to write code that is independent of the CPU type (CR52 in this case).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value Read from CR52</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20:18]</td>
<td>L7 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[17:15]</td>
<td>L6 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[14:12]</td>
<td>L5 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[11:9]</td>
<td>L4 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[8:6]</td>
<td>L3 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[5:3]</td>
<td>L2 cache information</td>
<td>000b</td>
</tr>
<tr>
<td>[2:0]</td>
<td>L1 cache information</td>
<td>011b</td>
</tr>
</tbody>
</table>

Cache Information
000b No cache
001b Instruction cache only
010b Data cache only
011b Instruction cache + data cache

The CR52 only incorporates L1 instruction and data caches.

In the sample code shown to the left, the mcr instruction is written in assembly language and the other processing is represented in the C language.

Note that "clidr=\[r0\]" is a simplified representation because syntactically correct writing is too complicated to show here.

This processing uses the invalidate_dcache function to invalidate the data cache at the obtained cache level.

Specify the target cache level in the cache size selection register (CSSELR).

\[
[3:1] \quad \text{Cache level setting. Only 000 can be specified for the CR52 because the CR52 incorporates caches at level L1 only.}
\]

\[
[0] \quad 0 = \text{Data cache}, 1 = \text{Instruction cache}
\]

Read the current cache size ID register (CCSIDR) to check the information on the cache level specified in the CSSELR described above.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
<th>CR52</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>Write-through</td>
<td>0 = Not supported; 1 = Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>[30]</td>
<td>Write-back</td>
<td>0 = Not supported; 1 = Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>[29]</td>
<td>Read-allocation</td>
<td>0 = Not supported; 1 = Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>[28]</td>
<td>Write-allocation</td>
<td>0 = Not supported; 1 = Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>[27:13]</td>
<td>Number of sets in cache – 1</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>[12:3]</td>
<td>Number of ways in cache – 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>[2:0]</td>
<td>Value indicating the cache line size (LS)</td>
<td>2 (64 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Cache line size (bytes) = \((2^{\text{LS}} + 2) \times 4\)

---

Lines, Sets, and Ways in Caches

The CR52 incorporates set-associative caches.

A line is the unit in which data can be stored in the cache.
A set consists of tags indicating addresses, V (valid) bits, and lines.
A way is a group of cached data for the number of sets.

To store data in a cache, whether data for the target memory address are already stored in the cache should first be checked. In the set-associative method, the number of locations in the cache that should be searched is limited to the number of ways in the cache because a set can be found by using the target memory address (a set includes tags for indicating memory addresses).

The size of the entire cache can be obtained from the line size, number of sets, and number of ways.

\[\text{Cache size (bytes) = number of sets x number of ways x cache line size}\]

L1 cache size in CR52 = \(128 \times 4 \times 64 = 32\) Kbytes

---

Figure 13 Details of Invalidating the Data Cache Memory
"Details of Invalidating the Data Cache Memory" is continued in the figure below.

Invalidate the entire data cache according to the information read from the CCIDR register described earlier. Use the data cache line invalidate by set/way (DCISW) register to invalidate the cache. Specify parameters for indicating a line in the DCISW register. This processing should proceed line by line, so use "for" statements to repeat it for the number of ways and sets and invalidate the entire cache. Calculation of the parameters that should be specified in the DCISW register is a little complicated as shown to the left.

Here, the mcr instruction is written in assembly language and the representation of values such as ccsidr is simplified.

**DCISW Register**

- **[31:32-A]** Specify the way value.
  - \( A = \) Value obtained by converting the number of ways to the number of bit shifts.
  - For the CR52, the number of ways is 4 and the number of bit shifts is 2.
  - Result = [31:30]

- **[B-1:L]** Specify the set value.
  - \( L = \) Value obtained by converting the line size to the number of bit shifts.
  - \( S = \) Value obtained by converting the number of sets to the number of bit shifts.
  - For the CR52, the line size is 64 and the number of bit shifts \( L \) is 6, and the number of ways is 128 and the number of bit shifts \( S \) is 7.
  - \( B = 13 \)
  - Result = [12:6]

Log2() converts a given parameter to the number of bit shifts.

\[
\begin{align*}
\text{shift_val} &= \log2(\text{raw_val}) \\
2^{\text{shift_val}} &= \text{raw_val}
\end{align*}
\]

Finally, execute the dmb instruction of the CR52 to guarantee the completion of memory access.
### 6.1.8 Enabling Cache Control

Enable the instruction cache and data cache. The following shows an example of settings.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrc p15, 0, r0, c1, c0, 0</td>
<td>Read the SCTLR register.</td>
</tr>
<tr>
<td>orr r0, r0, #0x1000</td>
<td></td>
</tr>
<tr>
<td>mcr p15, 0, r0, c1, c0, 0</td>
<td>Set the SCTLR register.</td>
</tr>
<tr>
<td>isb</td>
<td></td>
</tr>
</tbody>
</table>

Set bit 12 (the I bit) of the system control register to enable control of the instruction cache.

SCTLR

[12] Control of the instruction cache  0 = Disabled. 1 = Enabled.

The isb instruction flushes the instruction pipeline so that instruction code is again fetched from the main memory.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrc p15, 0, r0, c1, c0, 0</td>
<td>Read the SCTLR register.</td>
</tr>
<tr>
<td>orr r0, r0, #4</td>
<td></td>
</tr>
<tr>
<td>mcr p15, 0, r0, c1, c0, 0</td>
<td>Set the SCTLR register.</td>
</tr>
<tr>
<td>isb</td>
<td></td>
</tr>
</tbody>
</table>

Set bit 2 (the C bit) of the system control register to enable control of the data cache.

SCTLR

[2] Control of the data cache  0 = Disabled. 1 = Enabled.

The isb instruction flushes the instruction pipeline so that instruction code is again fetched from the main memory.

This completes the processing by the startup module. Execution advances to the main processing in the application.
6.2 Processor-Dependent Module

FreeRTOS is configured such that the processor-dependent module is separated from the main OS unit and the OS is easy to port. The following table lists the functions in the CR52 processor-dependent module. They are implemented for this sample program by the code in the "portable/GCC/RCarS4_CR52/" directory.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pxPortInitialiseStack</td>
<td>The task creation section of FreeRTOS calls this function to implement the processing for initializing the task stack.</td>
</tr>
<tr>
<td>xPortStartScheduler</td>
<td>The task scheduler start processing in FreeRTOS calls this function to implement the processing for activating the tick timer and the first task. In this sample program, activation of the tick timer is implemented by the code for setting up the tick timer and activation of a task is implemented by the vPortRestoreTaskContext function in portASM.S. For details on setting up the tick timer, see section 6.3.1, Setting up the Tick Timer.</td>
</tr>
<tr>
<td>vPortEndScheduler</td>
<td>The scheduler end processing calls this function. Although this function is prepared in the sample program, no processing is necessary because normal applications do not stop the scheduler.</td>
</tr>
<tr>
<td>vPortEnterCritical</td>
<td>Sections of code that requires exclusive control such as for queue handling call this function to implement the processing for disabling all interrupts so that task switching does not occur.</td>
</tr>
<tr>
<td>vPortExitCritical</td>
<td>This function implements the processing for returning from the interrupt-disabled state.</td>
</tr>
</tbody>
</table>
### 6.3 Driver Module

The peripherals used in the sample code for this application note are the timer and serial communications module. Descriptions of the serial communications are omitted in this application note because the general SCIF facilities are used. This section describes the interrupt controller (GIC) and timer, which are necessary for porting FreeRTOS. The following figure shows the configuration of the driver module.

![Driver Module Diagram](image)

*1 See section 6.3.1, Setting up the Tick Timer.
*2 See section 6.3.2, IRQ Interrupt.
*3 See section 6.3.3, Clearing the Tick Timer Interrupt.

**Figure 16 Configuration of the Driver Module**

The functions for "Set up the tick timer" and "Clear the tick timer interrupt" shown in Figure 16, Configuration of the Driver Module, are called from FreeRTOS. The names of the functions to be called are to be defined in FreeRTOSConfig.h. Create the contents of the defined functions to implement the necessary processing. The following table is a list of the names of and processing by the functions defined in FreeRTOSConfig.h.

<table>
<thead>
<tr>
<th>Processing</th>
<th>Function Definition Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up the tick timer</td>
<td>configSETUP_TICK_INTERRUPT</td>
<td>This function sets up the tick timer. It is called once after FreeRTOS is booted up. This function sets up the GIC to enable the timer interrupt and make initial settings of the timer.</td>
</tr>
<tr>
<td>Clear the tick timer interrupt</td>
<td>configCLEAR_TICK_INTERRUPT</td>
<td>This function clears the tick timer interrupt. It is called from the IRQ interrupt processing routine. As the timer in the CR52 does not have a facility for generating periodic interrupts, this function makes a new setting of the timer to clear the current interrupt and implement periodic interrupts.</td>
</tr>
</tbody>
</table>
6.3.1 Setting up the Tick Timer

Create the configSETUP_TICK_INTERRUPT function, which is required for the operation of FreeRTOS, to implement the processing for setting up the tick timer.

The following figure shows an example of processing for executing an interrupt handler after the specified tick time has elapsed. This processing is implemented for the sample program by the vConfigureTickInterrupt function.

---

**Register an interrupt handler.**

```c
#define TIMER_IRQ   (30)

Interrupt_InstallHandler(
    FreeRTOS_Tick_Handler,
    TIMER_IRQ);
```

**Use the Interrupt_InstallHandler function to register a handler for a timer interrupt number.**

The interrupt numbers are defined in the specifications of the generic timer as follows.

<table>
<thead>
<tr>
<th>Timer Type</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL1 physical timer</td>
<td>30</td>
</tr>
<tr>
<td>EL1 virtual timer</td>
<td>27</td>
</tr>
<tr>
<td>Non-secure EL2 physical timer</td>
<td>26</td>
</tr>
<tr>
<td>Non-secure EL2 virtual timer</td>
<td>28</td>
</tr>
<tr>
<td>EL3 physical timer</td>
<td>29</td>
</tr>
<tr>
<td>Secure EL2 physical timer</td>
<td>20</td>
</tr>
<tr>
<td>Secure EL2 virtual timer</td>
<td>19</td>
</tr>
</tbody>
</table>

Use the EL1 physical timer because FreeRTOS operates at exception level EL1. In the Interrupt_InstallHandler function, specify each pointer to indicate an interrupt handler in the array for the interrupt vector table with the index set to the target interrupt number.

```c
IntVector[irqno] = handler;
```

The Interrupt_InstallHandler function sets the pointer to a handler in the array with the index set to the target interrupt number. When an interrupt is generated, the handler (FreeRTOS_IRQ_Handler function) specified in the EL1 exception vector table described earlier is called. See section 6.1.4.1, Exception Handling Vector Table for details. The FreeRTOS_IRQ_Handler function, which is described later, obtains the interrupt source number (no) from a register and calls the handler determined by IntVector[no] shown to the left. If the interrupt number is 30 (timer), the FreeRTOS_Tick_Handler function is called.

The generic interrupt controller (GIC) controls interrupts. Many registers are controlled through the mcr (coprocessor control) instruction but the registers in the GIC are controlled by direct memory access because the GIC registers are mapped to addresses in memory.

The base address of the registers is the sum of the following addresses.

- **Base address**: F000 0000h
- **Offset to GICR**: 0011 0000h

**Specify an interrupt group.**

```c
#define GICR_IGROUP (0xF0110080)

reg = read_reg(GICR_IGROUP);
reg |= (1 << TIMER_IRQ);
write_reg(GICR_IGROUP, reg);
```

**The Interrupt_InstallHandler function sets the pointer to a handler in the array with the index set to the target interrupt number.**

When an interrupt is generated, the handler (FreeRTOS_IRQ_Handler function) specified in the EL1 exception vector table described earlier is called. See section 6.1.4.1, Exception Handling Vector Table for details. The FreeRTOS_IRQ_Handler function, which is described later, obtains the interrupt source number (no) from a register and calls the handler determined by IntVector[no] shown to the left. If the interrupt number is 30 (timer), the FreeRTOS_Tick_Handler function is called.

**Register an interrupt group.**

- **Base Address**
  - This address can be read from the IMP_CBAR register.
- **Offset to the GIC Redistributor (GICR)**
  - The base offset address for the registers of the GICR differs from core to core. The timer interrupt in the CR52 is a private peripheral interrupt (PPI) and the offset to the registers is 0011 0000h. For details, see the Cortex-R52 Technical Reference Manual and section 6.4.1, Initializing the GIC interrupts, in this application note.

**Specify the target interrupt group.**

Interrupts are classified into the following two groups.

<table>
<thead>
<tr>
<th>Interrupt Group</th>
<th>Description</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 0</td>
<td>FIQ interrupts</td>
<td>0</td>
</tr>
<tr>
<td>Group 1</td>
<td>IRQ interrupts</td>
<td>1</td>
</tr>
</tbody>
</table>

The sample program specifies "1" for the use of the timer interrupt as IRQ.

---

**Figure 17 Details of Setting up the Tick Timer**
"Details of Setting up the Tick Timer" is continued in the figure below

Set up the timer.
/* Read the CNTFRQ register. */
mrc p15, 0, r0, c14, c0, 0
freq = {r0}
tval = freq / configTICK_RATE_HZ;
{r0} = tval;
/* Set the CNTP_TVAL register. */
mcr p15, 0, r0, c14, c2, 0
{r0} = 1;
/* Set the CNTP_CTL register. */
mcr p15, 0, r0, c14, c2, 1

Set the priority mask to 29.
/* Set the GICR_ISENABLER register. */
#define GICR_ISRENABLER (0xF0110100)
reg = 1 << TIMER_IRQ;
write_reg(GICR_ISRENABLER, reg);

Specify the timer value indicating the tick time.
To calculate the timer value, read the CNTFRQ register, which will have been set up by the startup module, to obtain the frequency of the timer clock. Divide the obtained clock frequency by configTICK_RATE_HZ and the result is the timer value indicating the tick time.
configTICK_RATE_HZ is a mandatory definition item for FreeRTOS, and is set to 1,000 in most cases — that is, a tick interrupt is generated every 1 ms.

Specify the timer value in the CNTP_TVAL register. The CNTP_TVAL register is a down-counter; the timer interrupt signal is asserted when counting reaches 0 (zero).

Use the CNTP_CTL register to start the timer.
[31:3] Reserved
[2] State (read-only bit) 0 = Interrupt signal is not being asserted.
1 = Interrupt signal is being asserted.
[1] Mask for the timer output signal 0 = Not masked. 1 = Masked.
[0] 0 = Timer is disabled. 1 = Timer is enabled.

Interrupts are masked if their priority level is not higher than the value specified as the priority mask.
As the INTID for the timer is 30, interrupts having a priority level of 29 or less are masked.

Use the GICR_ISENABLER register to enable timer interrupts.
Sixteen private peripheral interrupts (PPIs) are prepared for each core (the CR52 is a single core). A specific peripheral is assigned to a PPI and the timer is one of the peripherals assigned to PPIs. The PPI assigned to a peripheral is fixed to level-sensitive and cannot be changed to edge-triggered.

Figure 18  Details of Setting up the Tick Timer (Continued)
6.3.2 IRQ Interrupt

When an IRQ interrupt is generated, the instruction specified at the offset address 18h in the EL1 vector table is executed, as described in section 6.1.4.1, Exception Handling Vector Table. In this sample program, an instruction for branching to the FreeRTOS_IRQ_Handler function is specified at this offset address and the FreeRTOS_IRQ_Handler function in portASM.S is executed.

The following figure shows an example of IRQ interrupt processing.

This processing is implemented for the sample program by the FreeRTOS_IRQ_Handler function in portASM.S.

![Diagram](image-url)

sub lr, lr, #4
push lr
mrs lr, spsr
push lr
cps #0x13
push {r0-r3, r12}
:
sub lr, lr, #4
The program counter (PC) of Arm CPUs points to the next instruction to be fetched instead of the current instruction. When a normal subroutine is called, the instruction "mov pc, lr" can be used to return from the subroutine to the correct location. However, in the case of return from an interrupt handler, the value 4 should be subtracted from the link register (lr), which stores the return address, because the lr value indicates the next instruction following the target instruction where execution should return.

cps #0x13
Upon entry to IRQ exception handling, the processor mode is automatically set to the IRQ mode. Among the six processor modes at exception level EL1 at which FreeRTOS operates, application tasks should be executed in the supervisor mode and the FreeRTOS kernel should be executed in the supervisor mode. As the interrupt handling requires kernel processing such as context switching, the processor mode should be changed from the IRQ mode (0x12) to the supervisor mode (0x13). Only the cps instruction can switch the processor modes within a given exception level.

push {r0-r3, r12}
Save the register values in preparation for a subroutine call in the exception handling. Only the r0, r1, r2, r3, and r12 registers are saved here because the standard method of function calls is determined in the Arm Architecture Procedure Call Standard (AAPCS) and the values of the other registers are guaranteed.

The sample program calls the vApplicationIRQHanler function (arm/interrupt.c) to execute peripheral interrupt handling.

Return from Interrupt Handling

Execute the inverse processing of that for entry to interrupt handling. Execute the cps instruction to return to the IRQ mode. Use the "msr SPSR_cxsf, lr" instruction to restore the saved program status register (SPSR). The four characters cxsf are the initial letters of the names that indicate the meaning of individual four 8-bit fields of the 32-bit SPSR data: c = control, x = extension, s = status, and f = flags. In this example, all 32 bits of data are restored. Only specifying certain fields is also possible. After the SPSR value is restored, the processor mode changes to the system mode (the information on the processor mode is contained in "c" among cxsf).

Finally, restore the PC to the lr register value to return to the processing to have been executed next before the interrupt.

Peripheral Interrupt Processing

Read interrupt controller interrupt acknowledge register 1 (ICC_IAR1) to find the interrupt number (intid) of the interrupt source. This sample program only uses the timer interrupt (intid = 30). Therefore, the value 30 is always returned when ICC_IAR1 is read. As the timer interrupt handler (FreeRTOS_Tick_Hander) was registered in IntVector[30] in the step “Set up the tick timer” of the driver module, IntVector[intid] shown to the left calls the registered timer interrupt handler. Finally, specify the interrupt number of the interrupt source in interrupt controller end of interrupt register 1 (ICC_EOFR1) to clear the highest active priority, and subsequent interrupts will be accepted according to the priority levels assigned before the interrupt generation.
6.3.3 Clearing the Tick Timer Interrupt

Create the configCLEAR_TICK_INTERRUPT function, which FreeRTOS requires to implement the processing for clearing the tick timer interrupt. This function is called from the IRQ interrupt processing routine.

The following figure shows an example of processing for clearing the timer interrupt and making a new setting of the timer.

This processing is implemented for the sample program by the vClearTickInterrupt function.

```
{r0} = tval;
/* Set the CNTP_TVAL register. */
mcr p15, 0, r0, c14, c2, 0
```

Specify the timer value for the tick time.

Use the timer value (tval) that was calculated in the step "Set up the tick timer" to specify the timer value in the CNTP_TVAL register. As soon as the timer value is specified, the current interrupt is cleared and the timer restarts counting down. Repeat this processing every time a tick timer interrupt is generated.

To simplify the processing, the time between generation of an interrupt and resetting of the timer value is not taken into account in the example shown to the left. Therefore, the tick time generated in this example is slightly delayed from the accurate tick time. When a more accurate tick time is needed, obtain the timer value to be set in the CNTP_TVAL from the system counter value.

**System Counter**

The system counter is a 64-bit timer used as the base for the timer facilities. The value is incremented per clock cycle. This counter is not affected by power management actions such as transitions to low-power states.

The value of the system counter can be read from the counter-timer physical count register (CNTPCT).

Figure 20  Details of Clearing the Tick Timer Interrupt
6.4 Application Module

The application module creates and runs FreeRTOS tasks. This sample program outputs the character string “Hello CR52 FreeRTOS” in the terminal window of the Windows PC connected for CR52 evaluation and then creates two tasks. The first task uses a queue to periodically send data to the second task. On receiving data from the queue, the second task turns an LED on or off. The following figure shows the connections between the connector and the LED. The output voltage on pin 5 of CN37 is approximately 1.8 V. A 1.8-V LED can thus be directly connected without resistors.

![Connection of LED](image)

Figure 21 Connection of LED

The following figure is a flowchart of the processing by the application module.

![Flowchart of the Application Module](image)

Figure 22 Flowchart of the Application Module

*1 See section 6.4.1, Initializing the GIC Interrupt Facilities.

*2 See section 6.3.1, Setting up the Tick Timer.

*3 See section 6.4.2, Starting Task Processing.
6.4.1 Initializing the GIC Interrupt Facilities

The generic interrupt controller (GIC) supports interrupt handling in a multi-core device. The GIC consists of three types of module: the distributor, redistributor, and CPU interface. The distributor accepts shared interrupts (SPI) and the redistributor accepts core-specific interrupts (PPI). The GIC enables direct execution of the interrupt handler having the highest priority level simply by reference to a table that holds the interrupt handler addresses. Thus the GIC can improve the response to interrupts in comparison to conventional processing that uses software to check the interrupt states of multiple peripherals. The CR52 in the R-Car S4 used in the sample program of this application note is a single-core CPU and this section does not describe the processing for multi-core interrupts. The following figure is a diagram of interrupt routing.

![Figure 23 Configuration of the GIC](image)

An SPI can be shared by multiple cores but a PPI is only routed to the core connected to the redistributor that receives the interrupt and cannot be shared by cores. The timer used in this sample program is a peripheral incorporated in each core and the interrupt from the timer is a PPI.

6.4.1.1 Overview of GIC Facilities

The following points give an overview of the distributor facilities.
- Enabling or disabling interrupts
- Controlling routing of interrupts
- Setting the attribute of each interrupt (level-sensitive or edge-triggered)
- Controlling the priority
- Setting interrupt groups (PPI, SGI, or others)

The following points give an overview of the redistributor facilities.
- Enabling or disabling SGIs and PPIs
- Setting the priority levels of SGIs and PPIs
- Assigning interrupts to groups
- Controlling the register base address
The following points give an overview of the CPU interface facilities.

- Notification of the end of interrupt processing
  After the end of interrupt processing, the interrupt handler writes to a register in the CPU interface to notify the GIC of the end of interrupt processing. Upon receiving the notification, the CPU interface allows other pending interrupts to be sent to the CPU.

- Priority masking
  Priority masking enables the masking of interrupts having lower priority levels.

- Selecting the highest-priority interrupt
  When multiple interrupts are input at the same time, the interface selects the interrupt having the highest priority level.

6.4.1.2 Registers of the GIC

The prefix (first four letters) of a GIC register name determines the module to which the register belongs. The following is a list of register name prefixes and the corresponding modules.

<table>
<thead>
<tr>
<th>Form of Register Name</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>GICD_xxxx</td>
<td>Distributor</td>
</tr>
<tr>
<td>GICR_xxxx</td>
<td>Redistributor</td>
</tr>
<tr>
<td>GICC_xxxx</td>
<td>CPU interface</td>
</tr>
</tbody>
</table>

6.4.1.3 Outline of Initialization

The following figure shows an outline of GIC initialization.

![Outline of GIC Initialization](image-url)
6.4.1.4 Initializing the Distributor

The sample program does not use SPIs but initialization of the distributor is still necessary to enable it. The following figure is a flowchart of distributor initialization.

![Flowchart of Distributor Initialization](image)

- **Start initialization of the distributor.**
- **Disable the distributor.**
- **num_irq = (GICD_TYPER[4:0]+1)*32**
- **n = 0**
- **Disable interrupts (n).**
  - Clear the pending bits of interrupts (n).
  - Set to level-sensitive (n).
  - Set the priority to an intermediate value (n).
  - Set to group 1 (n).
- **n = n + 1**
  - **n < num_irq ?**
    - **Yes**
    - **n = n + 1**
    - **n < num_irq ?**
      - **Yes**
      - **Set the two lower-order bits of the GICD_CTRL register to 00 to disable the distributor.**
      - **Obtain the number of interrupts.**
      - For the CR52: Bits 4 to 0 of the GICD_TYPER register are set to 30. num_irq = 992.
      - **Select group 1 because group 0 requires operations at exception level EL3. The selection of group affects the secure mode but the CR52 does not support the Trust Zone facilities.**
    - **No**
      - **Set the two lower-order bits of the GICD_CTRL register to 11.**
  - **No**
    - **Enable the distributor.**
- **return**

Figure 25 Details of Distributor Initialization
6.4.1.5 Initializing a CPU Interface

The following figure is a flowchart of CPU interface initialization by the sample program.

![Flowchart of CPU Interface Initialization](image)

Start initialization of the CPU interface.

Enable the CPU interface.

Specify 3 for the binary point setting.

Set the priority mask to 0x1F.

return

Set the lowest-order bit of the GICC_CTRL register to 1.

An interrupt priority value consists of eight bits and is separated into a group priority field and a subpriority field. The binary point specifies the number of bits assigned to the group priority and subpriority fields.

<table>
<thead>
<tr>
<th>Binary Point</th>
<th>Group Priority and Subpriority (g = group, s = sub)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ggggggggg.s</td>
</tr>
<tr>
<td>1</td>
<td>ggggggg.g.s</td>
</tr>
<tr>
<td>2</td>
<td>ggggg.sss</td>
</tr>
<tr>
<td>3</td>
<td>ggg.sssss</td>
</tr>
<tr>
<td>4</td>
<td>g.g.sssss</td>
</tr>
<tr>
<td>5</td>
<td>g.sssssss</td>
</tr>
<tr>
<td>6</td>
<td>sssssssss</td>
</tr>
</tbody>
</table>

Preemption is disabled.

Preemption does not occur when a new interrupt received by the CPU interface has the same group priority as the interrupt currently being processed.

Interrupts are masked if their priority level is not higher than the value specified as the priority mask.

---

6.4.1.6 Activating a Redistributor

The following figure shows how to activate a redistributor.

![Flowchart of Redistributor Activation](image)

Activate the redistributor.

Set the GICR_WAKER register to 0.

return

GICR_WAKER Register

- [31] Implement-dependent bit
- [30:3] Reserved
- [1] Power mode of the core: 0 = Normal mode, 1 = Low-power mode.
6.4.2 Starting Task Processing

The sample program handles processing for transferring data between two tasks and turning the LED on and off. The following figure shows an overview of task operation.

![Task Operation in the Sample Program](image)

The following listing is of the sample code of SendMessageTask.

```c
static void SendMessageTask( void *pvParameters )
{
    int32_t sw = 0;
    struct sample_que_t que;
    Console_Print("Send queue Task Started.
    ");
    for( ;; )
    {
        sw = !sw;
        que.sw = sw;
        xQueueSendToBack(l_queue, &que, 0);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}
```

The following listing is of the sample code of ReceiveMessageTask.

```c
static void ReceiveMessageTask( void *pvParameters )
{
    struct sample_que_t que;
    init_led();
    Console_Print("Receive Task Started.
    ");
    for( ;; )
    {
        xQueueReceive(l_queue, &que, portMAX_DELAY);
        led(que.sw);
        Console_Print("%s¥n", que.sw ? "ON" : "OFF");
    }
}
```
7. How to Build from the Source Code

7.1 Procedure of Building

- Deploying the Cross-Compiler

Download the ARM cross-compiler from the ARM Developer site.

It can be deployed in any directory; the following shows an example of deployment in the $HOME/cross directory.

```
$ mkdir $HOME/cross
$ cd $HOME/cross
$ tar xf gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2
$ export PATH=$HOME/cross/gcc-arm-none-eabi-10.3-2021.10/bin:$PATH
```

- Deploying FreeRTOS

Download FreeRTOS from the Github site.

It can be deployed in any directory; the following shows an example of deployment in the $HOME/cr52 directory.

```
$ mkdir $HOME/cr52
$ cd $HOME/cr52
$ wget https://github.com/FreeRTOS/FreeRTOS-LTS/releases/download/202012.03-LTS/FreeRTOSv202012.03-LTS.zip
$ unzip FreeRTOSv202012.03-LTS.zip
```

- Deploying the Sample Software Package

Deploy RCarS4_CR52_FreeRTOS_Sample.tgz provided with this application note in the directory where FreeRTOS has been deployed.

```
$ cd $HOME/cr52
$ tar xf RCarS4_CR52_FreeRTOS_Sample.tgz
```

- Building

```
$ cd App/RCarS4_CR52
$ make clean; make
```

After the successful completion of building, the following files will have been generated in the gcc directory.

Table 10 List of Files Generated by Building

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOSDemo.axf</td>
<td>Executable module file for the debugger</td>
</tr>
<tr>
<td>RTOSDemo.mot</td>
<td>Motorola S-format file for flash ROM programming</td>
</tr>
</tbody>
</table>
7.2 Executing the Sample Program

7.2.1 Setting up the Terminal Software

Use the terminal software TeraTerm to check the output through the serial port. Note that some boards have a restriction for the use of TeraTerm. For details, see section 8, Restrictions.

- Serial Port Settings

The following lists the settings of the serial port.

<table>
<thead>
<tr>
<th>Item</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected port</td>
<td>CN20 (Micro USB)</td>
</tr>
<tr>
<td>Bit rate</td>
<td>115,200 bps</td>
</tr>
<tr>
<td>Data</td>
<td>8 bits</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Stop bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>Flow control</td>
<td>None</td>
</tr>
</tbody>
</table>

7.2.2 Execution through the Debugger

The operation of the sample program after building can be immediately checked on the R-Car S4 by using the debugger.

7.2.2.1 Connecting a Debugger

The following shows the connections between the R-Car S4, a debugger, and a Windows PC.
7.2.2.2 Operating the Debugger

Activate the Partner-Jet2 software installed on the Windows PC.

Figure 30  Window Displayed after Activation of Partner-Jet2

Figure 31  Dialog Box for Opening a File

Open RTOSDemo.axf.
Set the program counter (PC) value in the register window to 0 (zero).

![Register Window (PC Setting)](image)

**Execution**

Connect the Windows PC to the CN20 connector of the board via a USB serial cable and activate TeraTerm. Press the button for executing a program in the window of the Partner-Jet2 software or press the F5 key to start the CR52 program. The character strings “ON” and “OFF” will be alternately displayed in the TeraTerm window every 0.5 seconds and the LED will simultaneously blink on and off.
7.2.3 Execution by Writing to the Flash ROM

Connect the CN21 connector of the evaluation board to the Windows PC via a USB serial cable and activate TeraTerm. Note that some boards have a restriction for the use of CN21. For details, see section 8, Restrictions.

Serial Port Settings
The following lists the settings of the serial port.

Table 12 Serial Port Settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected port</td>
<td>CN21 (Micro USB)</td>
</tr>
<tr>
<td>Bit rate</td>
<td>1,843,200 bps</td>
</tr>
<tr>
<td>Data</td>
<td>8 bits</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Stop bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>Flow control</td>
<td>None</td>
</tr>
</tbody>
</table>

Modify the settings of the SW8 switch block on the evaluation board as shown below.

Figure 33 Connection for Writing to the Flash ROM

Figure 34 SW8 Settings for Writing to the Flash ROM
Turn on the power to the board; the message shown in the figure below appears and the board waits for downloading of the program.

Drag and drop the ICUMXA_Flash_writer_SCIF_DUMMY_CERT_EB200000_spider.mot file from the file explorer to TeraTerm or open the file by selecting [File] → [Send file] from the TeraTerm menu bar.

![Activation Message](image1)

Figure 35 Activation Message

After the completion of downloading, the program for writing to the flash ROM outputs the text shown in the figure below and waits for input by the user. Enter the command and values underlined in the figure. Note that after “1” is entered in response to “Select (1-3)”, the user is prompted to check the settings of SW1 and SW2. When the settings are correct, enter “y”. The line where “y” was entered is overwritten by the next message and does not remain on the screen. It is also not shown in the figure.

![Flash Writer Output Messages](image2)

Figure 36 Flash Writer Output Messages

The flash writer waits for downloading.
Drag and drop the RTOSDemo.mot file from the file explorer to TeraTerm or open the file by selecting [File] → [Send file] from the TeraTerm menu bar.

```
SPI Data Clear(H'FF) Check :H'00500000-0057FFFF,Clear OK?(y/n)
```

Figure 37 Flash Writer Message to Request Confirmation

After the file has been transferred, a message to request confirmation of the erasure of flash ROM is output. Check the storage addresses and enter “y”.

```
SPI Data Clear(H'FF) Check :H'00500000-0057FFFF Erasing...Erase Completed
SAVE SPI-FLASH....... complete!

-------- Ospi/HyperFlash Save Information  -----------------------
SpiFlashMemory Stat Address : H'00500000
SpiFlashMemory End Address : H'0055148F

```

Figure 38 Flash Writer Message for Completion of Writing

Display of the “SAVE SPI-FLASH....... complete!” message indicates completion of writing to the flash ROM. After turning off the power to the board, restore the SW8 switch block to the settings for normal booting up.

```
1 2 3 4 5 6 7 8
ON
OFF

```

Figure 39 SW8 Settings for Normal Booting

Turn on the power and check that the LED blinks.
8. Restrictions

8.1 Restrictions on the Evaluation Board

Evaluation boards with serial number 2133 or a higher number are subject to the following restrictions due to changes in the wiring patterns on the boards.

1. The messages from the sample program are not output to TeraTerm in section 7.2.1, Setting up the Terminal Software. The LED will blink, so check the operation of the sample program by observing the LED.

2. The connection port used in section 7.2.3, Execution by Writing to the Flash ROM, is changed from CN21 to CN20.
9. References

For the following documents, obtain the latest version from the Arm Developer Web site.

- Arm Generic Interrupt Controller Architecture Specification
- Aarch64 Programmer’s Guide Generic Timer
- Arm Architecture Reference Manual Supplement
  - Armv8, for the Armv8-R AArch32 architecture profile

For the following document, obtain the latest version from the the Renesas Web site.

- R-Car S4 Series User’s Manual
## Revision History

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Jun. 10, 2022</td>
<td>All pages</td>
<td>Initial version</td>
</tr>
<tr>
<td>1.01</td>
<td>Nov. 8, 2022</td>
<td>4</td>
<td>Modified description of R-Car Market Place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All pages</td>
<td>Corrected footer note.</td>
</tr>
</tbody>
</table>
General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity.
   Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.

2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.

3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.
   Follow the guideline for input signal during power-off state as described in your product documentation.

4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.

5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.

6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between $V_{IL}$ (Max.) and $V_{IH}$ (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between $V_{IL}$ (Max.) and $V_{IH}$ (Min.).

7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.

8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.
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 or any other use of the circuits, software, and information in the design of your product(s) or systems. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.

2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving 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, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.

3. 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 shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.

5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.

6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended 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; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or 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 damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVIULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.) and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.

9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety measures 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 and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

11. Renesas Electronics products and technologies shall 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 shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.

12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.

13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.

14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev 5.0-1 October 2020)

Corporate Headquarters
TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information
For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/

Trademarks
Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

© 2022 Renesas Electronics Corporation. All rights reserved.