R8C Family
Implementing Interrupts in MR8C/4

Introduction
The ability to handle multi tasks simultaneously is important in embedded systems. Such capability is achieved by having external hardware devices that perform device-specific operations in parallel to the core processor. This concurrency is achieved using interrupts. RTOSs handle interrupts in different ways and require varying methods of implementing interrupt handlings.

This document describes interrupts, Renesas R8C family interrupts architecture, and how they are implemented in MR8C/4, their impact on scheduling and real-time, and some interrupt-management strategies.

Target Device
Applicable MCU: R8C Family

Contents

1. Guide in using this Document ................................................................. 2
2. Introduction to Interrupts in Embedded Systems .............................. 3
3. Overview of Renesas R8C Family Interrupts Architectures .............. 5
4. Understanding RTOS Interrupt Architecture .................................. 10
5. Implementing Interrupts in MR8C/4 .................................................. 13
6. Reference Documents ........................................................................... 26
1. Guide in using this Document

This document aims to equip users with the confidence and capability of implementing interrupts in MR8C/4.

With sufficient coverage ranging from introduction of interrupts in embedded systems to an overview of Renesas R8C Family interrupt architectures, users will be able to relate the explanation on the implementation of interrupts in MR8C/4 for the R8C Family devices.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Objective</th>
<th>Pre-requisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Interrupts in Embedded Systems</td>
<td>A basic introduction to interrupts</td>
<td>Fundamental knowledge in embedded systems</td>
</tr>
<tr>
<td>Overview of Renesas R8C Family Interrupts Architectures</td>
<td>To reinforce users’ understanding on the interrupt architectures of R8C Family devices</td>
<td>Knowledge in Microcontroller</td>
</tr>
<tr>
<td>Understanding RTOS Interrupt Architecture</td>
<td>Provide a brief overview of interrupts implementation in RTOS and its associated challenges and concerns</td>
<td>Knowledge in RTOS</td>
</tr>
<tr>
<td>Implementing Interrupts in MR8C/4</td>
<td>Guide users in the implementation of interrupts in MR8C/4</td>
<td>Knowledge in interrupts architecture of R8C Family devices</td>
</tr>
<tr>
<td>Reference Documents</td>
<td>Listing of documents that equip users with knowledge in the pre-requisite requirements</td>
<td></td>
</tr>
</tbody>
</table>
2. Introduction to Interrupts in Embedded Systems

2.1 Why Interrupts?
To satisfy end-consumers’ increasingly demands, more functionality is consistently added to the embedded system. As the functional requirements proliferate, size of the software grows. Thereby, it becomes more difficult to ensure that time-critical items (e.g. capture data from external inputs) are performed properly.

“Interrupts” was introduced to handle this problem. Interrupt signals an event to the microprocessor to instruct it to stop what it is doing, and handle the incoming task.

2.2 Types of Interrupts
There are two types of interrupts; Hardware and Software. Software interrupts are generated from software through the issuance of a specified command. Hardware interrupts are triggered by peripherals and external devices connected to the microprocessor.

2.3 Interrupts Components

2.3.1 Programmable Interrupt Controller (PIC)
PIC is a device that accept and prioritize multiple interrupt sources so that the highest priority interrupt is directed to core CPU for processing at any point of time. In addition, PIC helps core CPU to determine an interrupt’s exact source.

2.3.2 Interrupts Service Routine (ISR)
ISRs are software routine that handle and process interrupt requests as specified by users. When an interrupt is generated, processor breaks away its present task, switch to interrupt stack pointer and points to its ISR to process the interrupt. Execution control is returned to the main program when the ISR completed.

2.3.3 Interrupt Vector Table
Interrupt vector table carries a list of all interrupt service routine and their corresponding properties (e.g. priority level, address, descriptions, etc) that define the dynamic characteristics of the interrupt sources. Interrupt vector table provides processor the means to jump into the ISR of the corresponding interrupt by extracting its ISR address listed in the table. R8C family devices consist of two interrupt vector tables; fixed and relocatable interrupt tables shown in Figure 1.
2.3.4 Interrupt Priority Level

To facilitate nested interrupts, a “priority” level is generally provided for each interrupt. An interrupt priority level (IPL) is assigned by users to individual interrupts to represent how critical the interrupt is. When two interrupts happen at the same time, the higher priority interrupt will take precedence over the lower priority one.

2.3.5 Masked/ Unmasked Interrupts

A processor can mask or block interrupts requests to perform a task. Hardware interrupts can be further categorized into maskable interrupts (IRQ) and non-maskable interrupt (NMI).

IRQs are interrupts that can be disabled to let higher priority ISRs be executed uninterruptedly. Interrupt masking is usually done by clearing the Interrupt Enable Flag.

NMI is a special type of interrupt that cannot be disabled by standard masking technique and is typically used to signal attention for reporting of non-recoverable hardware errors, methods for system debugging, and special case handling such as system resets.
3. Overview of Renesas R8C Family Interrupts Architectures

In the µITRON4.0 specification, implementation of an interrupt handler is generally dependent on the processor interrupt architecture and the Programmable Interrupt Controller (PIC). As MR8C/4 conforms to µITRON4.0 specification and specifically designed for R8C family devices, users are required to understand the interrupt architectures of the R8C family devices in the setup of interrupt handlers and interrupt service routine.

For R8C family devices, there are two main types of interrupts (Software and Hardware). For hardware interrupt, it is further categorized into Special and Peripheral I/O interrupts as shown in Figure 2.

**Notes:**
1. Peripheral function interrupts are generated by the peripheral functions built into the microcomputer system.
2. This is a dedicated interrupt for development support tools. Do not use this interrupt.

![Figure 2 Classifications of Interrupts](image)

For R8C family devices, there are two vector tables, namely, “Fixed Vector Table” and “Relocatable Vector Table”.

As the names imply, “Fixed Vector Table” resides in allocated addresses (e.g. 0FFDCh to 0FFFFFFh). Whereas “Relocatable Vector Table” can be allocated at any desired location within the devices’ entire memory space by INTB register relative addressing.

Figure 3 and Figure 4 provide the “Fixed Vector Table” and “Relocatable Vector Table” of R8C/Lx devices respectively.
## Implementing Interrupts in MR8C/4

### Table 1: Interrupt Sources and Addresses

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Vector Addresses Address (L) to Address (H)</th>
<th>Remarks</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unhandled instruction</td>
<td>OFFCH to OFFDH</td>
<td>Interrupt with UNH instruction</td>
<td>R8C/4 Family Software Manual</td>
</tr>
<tr>
<td>Overflow</td>
<td>OFFE8h to OFFE9h</td>
<td>Interrupt with INTO instruction</td>
<td>R8C/4 Family Software Manual</td>
</tr>
<tr>
<td>BRK instruction</td>
<td>OFFE4h to OFFE7h</td>
<td>If the content of address OFFE7h is FFh, program execution starts from the address shown by the vector in the relocatable vector table.</td>
<td>R8C/4 Family Software Manual</td>
</tr>
<tr>
<td>Address match</td>
<td>OFFE8h to OFFE9h</td>
<td>12.6 Address Match Interrupt</td>
<td>R8C/4 Family Software Manual</td>
</tr>
</tbody>
</table>

### Figure 3: Fixed Vector Tables of R8C/Lx Devices

- **Interrupt Source**: BRK instruction (3)
  - **Vector Addresses**: `0 to 3` (0000h to 0003h)
  - **Software Interrupt Number**: 0
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)

- **Interrupt Source**: Flash memory ready
  - **Vector Address**: 4 to 7 (0004h to 0007h)
  - **Software Interrupt Number**: 1
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Flash Memory

- **Interrupt Source**: INT0
  - **Vector Address**: 10 to 15 (0010h to 0015h)
  - **Software Interrupt Number**: 2
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: INT1
  - **Vector Address**: 16 to 19 (0016h to 0019h)
  - **Software Interrupt Number**: 4
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: INT2
  - **Vector Address**: 20 to 23 (001Ah to 001Dh)
  - **Software Interrupt Number**: 6
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: Timer RC
  - **Vector Address**: 24 to 26 (0018h to 001Ah)
  - **Software Interrupt Number**: 7
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: Timer RA
  - **Vector Address**: 28 to 31 (001Ch to 001Fh)
  - **Software Interrupt Number**: 9
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: Key input
  - **Vector Address**: 32 to 35 (0020h to 0023h)
  - **Software Interrupt Number**: 11
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: A/D conversion
  - **Vector Address**: 36 to 39 (0024h to 0027h)
  - **Software Interrupt Number**: 13
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: UART interface
  - **Vector Address**: 40 to 43 (0028h to 002Bh)
  - **Software Interrupt Number**: 15
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: PC bus interface
  - **Vector Address**: 44 to 47 (002Ch to 002Fh)
  - **Software Interrupt Number**: 17
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: SYSCON (synchronous serial communication unit)
  - **Vector Address**: 48 to 51 (0030h to 0033h)
  - **Software Interrupt Number**: 19
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

### Figure 4: Relocatable Vector Tables of R8C/Lx Devices

- **Interrupt Source**: UART interface
  - **Vector Address**: 60 to 63 (0030h to 0033h)
  - **Software Interrupt Number**: 21
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: UART interface
  - **Vector Address**: 64 to 67 (0034h to 0037h)
  - **Software Interrupt Number**: 23
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

- **Interrupt Source**: UART interface
  - **Vector Address**: 68 to 71 (0038h to 003Bh)
  - **Software Interrupt Number**: 25
  - **Interrupt Control Register**: \( \text{ICS}[2:0] = 0 \)
  - **Reference**: Unknown

### Figure 5

Figure 5 illustrates an example of defining the starting address locations of "Fixed Vector Table" and "Relocatable Vector Table" for a R8C23 device.
3.1 Software Interrupts

Software interrupts refers to interrupt requests that are put up by instructions in the R8C instruction set. Figure 6 provides a list of software interrupts for R8C22/23 devices.

<table>
<thead>
<tr>
<th>No.</th>
<th>Types</th>
<th>Activation Method</th>
<th>Location</th>
<th>Maskable Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Undefined Instruction Interrupt</td>
<td>When an UND instruction is executed</td>
<td>Fixed vector table (0FFDCh to 0FFDFh)</td>
<td>Non-maskable</td>
</tr>
<tr>
<td>2</td>
<td>Overflow Interrupt</td>
<td>When an arithmetic operation overflow</td>
<td>Fixed vector table (0FFE0h to 0FFE3h)</td>
<td>Non-maskable</td>
</tr>
<tr>
<td>3</td>
<td>BRK interrupt</td>
<td>When a BRK instruction is executed</td>
<td>Fixed vector table (0FFE4h to 0FFE7h)</td>
<td>Non-maskable</td>
</tr>
<tr>
<td>4</td>
<td>INT Instruction Interrupt</td>
<td>When an INT instruction followed by an interrupt number (0 to 53) is executed</td>
<td>Relocatable vector table</td>
<td>Non-maskable (0 to 63)</td>
</tr>
</tbody>
</table>

The maskable property of INT instruction interrupt differs for R8C/Lx devices. In R8C/Lx devices, software interrupt numbers 0 to 31 and 42 to 55 are maskable, whereas software interrupt numbers 32 to 41 and 56 to 63 are non-maskable.

3.1.1 Writing Software Interrupts

With reference to the software interrupts listing in Figure 6, only INT instruction interrupt (that resides in relocatable vector table) may be defined by users. Therefore, software interrupts are written by defining the INT number as illustrated in Figure 7.
3.2 Hardware Interrupts

Hardware interrupts comprise of Special and Peripheral I/O interrupts.

3.2.1 Special Interrupts

Special interrupts are used to handle non-recoverable errors pertaining to the processor, which need immediate attention. They are therefore non-maskable. Listings of special interrupts vary from one device to another in the R8C family. Figure 8 shows classification of special interrupts for R8C22/23 and R8C/Lx devices respectively.

<table>
<thead>
<tr>
<th>No.</th>
<th>R8C22/23 Group</th>
<th>R8C/Lx Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Watchdog timer</td>
<td>Watchdog timer</td>
</tr>
<tr>
<td>2</td>
<td>Oscillation stop detection</td>
<td>Oscillation stop detection</td>
</tr>
<tr>
<td>3</td>
<td>Voltage monitor 2</td>
<td>Voltage monitor 2/ comparator A2</td>
</tr>
<tr>
<td>4</td>
<td>Single step</td>
<td>Single step</td>
</tr>
<tr>
<td>5</td>
<td>Address break</td>
<td>Address break</td>
</tr>
<tr>
<td>6</td>
<td>Address match</td>
<td>Address match</td>
</tr>
<tr>
<td>7</td>
<td>Voltage monitor 1/ comparator A1</td>
<td></td>
</tr>
</tbody>
</table>

With reference to Figure 8, R8C/Lx devices consist of one additional special interrupts “Voltage monitor 1/ comparator A1” as compared to R8C22/23 devices even though they belonged to the same R8C family. Thus, it is crucial for users to find out the Special interrupts available to the specific device prior to using it.

3.2.2 Peripheral I/O Interrupts

These maskable interrupts are used to response to events triggered by peripherals built into the microcomputer (MCU) system. With the types of peripheral features vary with each R8C model, so do the types of Peripheral I/O interrupts. For R8C family, software interrupt numbers are appended to the peripheral I/O interrupts. This implies MCU will execute the same interrupt routine when the INT instruction is executed as when a peripheral function interrupt is generated. Figure 9 shows few of peripheral I/O interrupts available for R8C/Lx devices.
3.2.3 Writing Hardware Interrupts

For hardware interrupts, only peripheral interrupts (that resides in relocatable vector table) may be defined by users. There are two steps involved in writing a hardware interrupt in R8C family (refer to Figure 10). They are

1. Define interrupt service routine for the hardware interrupt (To be executed by user)
2. Register interrupt service routine in interrupt vector table (Automatically generated by compiler)
Interrupts are part of a mechanism provided by embedded processor architectures to allow for disruption of processor’s normal execution path and attend to external events. Generally, handling of interrupts in an embedded application with RTOS is implemented in two layers, namely RTOS layer (first layer) and application layer (second layer).

**Figure 10 Writing Hardware Interrupts**

1. **Define Interrupt Service Routine for the Hardware interrupt (Generated by User)**

```c
#pragma interrupt _timer_re(vect=10)

void _timer_re (void) {
    interrupt handling codes
}
```

2. **Register Interrupt Service Routine in Interrupt Vector Table (Generated by Compiler)**

```assembly
; variable vector section

.section       vector,ROMDATA
.org           __VECTOR_ADR__

.if 0
    .lword dummy_int ; vector 0
    .lword dummy_int ; vector 1
    .lword dummy_int ; vector 2
    .lword dummy_int ; vector 3
    .lword dummy_int ; vector 4
    .lword dummy_int ; vector 5
    .lword dummy_int ; vector 6
    .lword dummy_int ; vector 7
    .lword dummy_int ; vector 8
    .lword dummy_int ; vector 9
    .lword _timer_re ; vector 10
    .lword dummy_int ; vector 11
    .lword dummy_int ; vector 12
    .lword dummy_int ; vector 13
    .lword dummy_int ; vector 14
```

**4. Understanding RTOS Interrupt Architecture**

Interrupts are part of a mechanism provided by embedded processor architectures to allow for disruption of processor’s normal execution path and attend to external events. Generally, handling of interrupts in an embedded application with RTOS is implemented in two layers, namely RTOS layer (first layer) and application layer (second layer).
The RTOS layer, being the first layer, functions as an interrupt handler in accepting the control of an interrupt when it occurred. Upon receiving the control, the RTOS interrupts handler calls the application layer interrupt service routine corresponding to the interrupt to process.

4.1 RTOS Interrupt Architecture Design Challenges

Generally, there are two fundamental challenges in the designing of RTOS interrupt architecture. The first challenge is to ensure the integrity of internal RTOS data when servicing interrupt/s. The second challenge is to achieve a deterministic and low interrupt latency that will not affect the real-time performance of the system.

4.1.1 Challenge 1: Ensure Integrity of Internal RTOS Data

A major challenge in RTOS design involves supporting asynchronous access by interrupt routines and RTOS service calls to internal RTOS data structures. An ISR or RTOS service call while modifying a data structure should not be allowed to be interrupted that allows a different ISR or RTOS service call to make unrelated modifications to the same data structure as shown in Figure 11.

**Figure 11 Race Condition of Interrupts**

Unified and segmented interrupt architectures are two common approaches to prevent corruption to internal RTOS data structures by interrupt service routines (ISRs) and RTOS service calls.

In unified interrupt architecture (refer to Figure 12), interrupts are temporarily disabled when ISR or RTOS service calls is accessing the critical sections of code/data. This prevents other interrupts from making uncoordinated changes to the same critical sections.
In segmented interrupt architecture, interrupts are not disabled during ISR. To prevent interrupts occurring during ISR from accessing the same RTOS data structure and thereby corrupting it, segmented architecture disable the application scheduler, and divide the ISR into two or more pieces as shown in Figure 13.

In unified interrupt architecture approach, interrupt latency is introduced while the interrupts are temporary disabled. Additional system resources are also required for the allocation of separate system stack used to process all interrupts and nested interrupts. However, unified interrupt architecture is easier to implement, introduce less system overhead, less likely to result in stack resource problems and shorter interrupt completion time as compared with segmented interrupt architecture.

MR8C/4 employs unified interrupt architecture, and thus enjoys its benefits.

4.1.2 Challenge 2: Achieve Deterministic and Low Interrupt Latency

The deterministic characteristic of an embedded system and its corresponding worse case interrupt latency can be computed by examining all of the sources of interrupt response delays to ascertain which particular source causes the longest delay to the servicing and completion of the highest priority interrupt. Areas of examination encompass:

- Worse case hardware-induced delay
- Worse case software-induced (e.g. by RTOS kernel) delay
- Longest interrupt disabling region by a lower priority interrupt

To achieve a deterministic and low interrupt latency system, quality-programming techniques coupled with proper RTOS interrupt architecture are required.
Quality programming techniques generally involves:

1. Keeping ISRs as simple and short as possible
2. Avoid using instructions that take many cycles to execute
3. Avoid improper usage of RTOS service calls in ISRs
4. Prioritize interrupts appropriately with relative to tasks
5. Keep interrupt disabling regions as short as possible

R8C family devices leverage on an efficient and versatile instruction set and register architecture. Equipped with fast instruction execution time which has 20 (out of total 89) instructions that execute in a single cycle, powerful mathematical instructions and frequently used instructions (MOV, ADD, SUB, JMP) that are just 1-byte long, interrupt latency incurred by R8C MCUs is minimal.

MR8C/4 provides an adequate set of absolutely necessary and deterministic RTOS service calls that can be executed from ISRs. This gives users the convenience and ensures improper RTOS service calls are not used in ISRs.

By employing unified interrupt architecture, MR8C/4 ensures interrupt disabling regions are kept sufficiently short.

5. Implementing Interrupts in MR8C/4

MR8C/4 RTOS provides a flexible interrupt control mechanism that is fast and deterministic with the following characteristics:

- Options to create interrupt handlers in C or Assembly.
- Selection for OS-dependent (kernel) or OS-independent (non-kernel) interrupts
- Support for nested interrupts
- Support up to 7 priority levels (Highest priority at 7)
- Execute at higher precedence than dispatcher
- Execute in own independent contexts (non-task)
- Dedicated interrupt stack

5.1 Understanding Interrupt Processing in MR8C/4

There are basically three different kind of scenarios of interrupt processing in MR8C/4:

1. Interrupt occurs when processing task (refer Figure 14)
2. Interrupt occurs when processing service call (refer Figure 15)
3. Interrupt occurs within interrupt handler (refer Figure 16)
Figure 14 Interrupt Occurring when Processing Task

Figure 15 Interrupt Occurring when Processing Service Call
5.2 Defining Kernel/Non-Kernel Interrupts

In MR8C/4, interrupts are classified into kernel and non-kernel types.

A kernel interrupt allows kernel service calls to be issued within its ISR. Longer interrupt processing time will be incurred for kernel interrupts as their ISRs can only be completed after the service calls within the ISRs are being processed (refer to Figure 17). Therefore, non-maskable interrupts and Watchdog Timer interrupt must not be defined as kernel interrupts.

A non-kernel interrupt does not allow kernel service calls to be issued within its ISR. Since service call is not allowed within the non-kernel interrupt service routine, no extra delay will be incurred due to the service call processing. Non-kernel interrupts are therefore generally reserved for the NMI and critical interrupts.
The interrupt priority level and kernel interrupt mask level dictates whether it is a kernel or non-kernel interrupts. To define an interrupt as a kernel interrupt, assign its priority level to be lower or equal to the kernel interrupt mask level. Conversely, assign an interrupt priority level to be higher than the kernel interrupt mask level if it is designated to be a non-kernel interrupt (refer to Figure 18).

**Figure 18 Defining Kernel and Non-Kernel Interrupts Handler**

Figure 19 provides an example of defining INT1 as a kernel interrupt handler.

```
// System Definition
system{
  stack_size = 400;
  priority = 255;
  system_ipl = 4;
  tic_num = 1;
  tic_6eno = 1;
};
interrupt_vector[25][
  os_int = YES;
  entry_address = INT_INT1();
  pragma_switch = E;
];

void ConfigureInterrupts(void)
{
  /* Enable INTn Interrupt */
  intlen=1;
  /* Set INTn Interrupt priority */
  intic = 0x02;
```

**Figure 19 Defining INT1 as Kernel Interrupt Handler**

### 5.3 Enabling/Disabling Interrupts

Although it is possible to control (enable/disable) interrupts by manipulating the IPL value, it is not recommended as the manipulation can only be done by resetting the interrupt flag to zero. However, there are two other methods that are more appropriate to fulfill this requirement:

1. Modify corresponding interrupt control register (SRF) of the interrupt
2. Utilitize service calls “loc_cpu” and “unl_cpu”
3. Setting ‘I’ flag to enable/disable maskable interrupts
Method 3 uses the least overhead among the three methods mentioned above. In addition, it is the easier to execute. However, it is important to take note that no service call is allowed to be invoked in the setting of the ‘I’ flag. Figure 20 illustrates this concern.

---

```assembly
asnl("CLR I"); //Maskable Interrupts Disabled

...\{ Service calls not allowed
...\} in the setting of 'I' flag

... asnl("SET I"); //Maskable Interrupts Enabled
```

---

**Figure 20 Setting ‘I’ Flag for Controlling Interrupts**

5.3.1 Enable/Disable Non-Kernel Interrupts

Method 1 (as mentioned above) is used to control non-kernel interrupts. Figure 21 illustrates the process of disabling/enabling a non-kernel interrupt in R8C/Lx devices by modifying its SRF register.

5.3.2 Enable/Disable Kernel Interrupts

To control kernel interrupts, user is only required to use the "loc_cpu" and "unl_cpu" service calls. "loc_cpu" disable kernel interrupts by placing system in CPU locked state. "unl_cpu" release system from CPU locked state. Figure 22 illustrates the process of disabling/enabling a kernel interrupt in R8C/Lx devices by using service calls “loc_cpu” and “unl_cpu”.

---
1. Timer RB declared as non-kernel interrupt

```c
// System Definition
system{
    stack_size = 400;
    priority = 255;
    system_IP = 4;
    tic_name = 1;
    tic_deno = 1;
};

interrupt_vector[24]{
    os_int = NO;
    entry_address = TimerRBHandler();
    pragma_switch = E;
};
```

2. TRBIC interrupt control register of R8CLX

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Bit Name</th>
<th>Function</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td>ILVL2</td>
<td>ILVL2</td>
<td>Interrupt priority level select bit</td>
<td>R/W</td>
</tr>
<tr>
<td>3</td>
<td>IR</td>
<td>IR</td>
<td>Interrupt request bit</td>
<td>R/W</td>
</tr>
<tr>
<td>4-7</td>
<td></td>
<td></td>
<td>Nothing is assigned. If necessary, set to 0. When read, the content is undefined.</td>
<td></td>
</tr>
</tbody>
</table>

3. Modifying TRBIC interrupt control register to enable/disable Timer RB interrupt

```c
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task1(int stacd)
{
    trbic = 0;  /* Timer RB interrupt disabled */
    ...
    ...
    ...
    trbic = 5;  /* Timer RB interrupt enabled and interrupt priority level 5 defined */
}
```

Figure 21 Enable/Disable Non-Kernel Interrupts
1. INT1 declared as kernel interrupt in template.cfg

```c
// System Definition
system{
    stack_size = 400;
    priority = 256;
    system_IPL = 4;
    tic_num = 1;
    tic_den = 1;
};

interrupt_vector[25]{
    os_int = YES;
    entry_address = INT_INT1();
    pragma_switch = E;
};
```

2. INT1 interrupt control register of R8C/LX

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Symbol</th>
<th>Bit Name</th>
<th>Function</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>b7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>b0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>---</td>
</tr>
</tbody>
</table>

3. Using service calls "loc_cpu" and "unl_cpu" to enable/disable INT1 interrupt

```c
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task1(VP_INT *stacl)
{
    int lic = 0x03;        /*Define INT 1 priority 3 */
    errd = loc_cpu();     /*Kernel interrupts disabled*/
    ...
    ...
    errd = unl_cpu();     /*Kernel interrupts enabled*/
}
```

Figure 22 Enable/Disable Kernel Interrupts
5.4 Coding Interrupts in C Language

5.4.1 Defining Interrupt Vector Tables

Figure 23 illustrates the definition of vector tables in “c_sec.inc” file for R8C/Lx devices.

```c
// VECTOR TABLE

.glob __INT_VECTOR
.section INTERRUPT_VECTOR
.org Oxff00

.section FIX_INTERRUPT_VECTOR
.org OFFD8
.reserved
.addr Oxfffffff
.OFS2
.byte OFFH

; WDT disable
.ofsreg OFFH

#define _WDT_OFF

*Important: To disable Watchdog Timer Interrupt

Figure 23 Defining Vector Tables in “c_sec.inc” in C Language
```

5.4.2 Writing Kernel Interrupt Handler

Step 1: Define kernel mask level

The kernel mask level is defined in configurator file (e.g. template.cfg) under the system definition.

```c
// System Definition
system{
    stack_size = 400;
    priority = 255;
    system_IPL = 5;
    tic_nume = 1;
    tic_deno = 1;
};

Kernel mask level defined at 5

Figure 24 Defining Kernel Mask Level in C Language
```

Step 2: Define interrupt vector

The interrupt vector is defined in the configurator file (e.g. template.cfg). Figure 25 illustrates an example of defining INT1 vector as a kernel interrupt.
Step 3: Define interrupt handler

The final step is to define the interrupt handler in the “.c” file. The function name for the interrupt handler is the same as being defined in step 2. Figure 26 illustrates an example of this step.

```c
#include <itron.h>
#include <kernel_id.h>

void ConfigureInterrupts(void) [ 
  intlen=1;
  intlic = 0x02;
]

void INT_INT1(void) [ 
  ........Processing starts...
  ........Processing ends...
]
```

**Figure 26 Defining Interrupt Handler**

1. Refers to the software interrupt number as specified for the particular interrupt in the device user manual.
2. “YES” denotes kernel interrupt handler. #pragma INTHANDLER=YES_INT_INT1 will be output.
   “NO” denotes non-kernel interrupt handler. #pragma INTERRUPT=YES_INT_INT1 will be output.
3. Refers to the function name of the interrupt handler
4. Two switch options are available (‘B’ and ‘E’) for non-kernel interrupt handlers. Only one option (‘B’) is available for kernel interrupt handlers. 
   ‘B’ indicates bank switching with register bank 1 specified.
   ‘E’ specify named interrupts allowed.

**Figure 25 Defining Interrupt Vector**

The function name for the interrupt handler is the same as being defined in step 2.
5.4.3 Writing Non-Kernel Interrupt Handler

Step 1: Define kernel mask level

The kernel mask level is defined in configurator file (e.g. template.cfg) under the system definition.

```c
// System Definition
system{
    stack_size = 400;
    priority = 255;
    system_IPL = 5;
    tic_nume = 1;
    tic_denom = 1;
};
```

**Figure 27 Defining Kernel Mask Level**

Step 2: Define interrupt vector

```c
interrupt_vector[25][
    os_int = No;
    entry_address = INT_INT1();
    pragma_switch = B;
];
```

**Figure 28 Defining Interrupt Vector**

Step 3: Define interrupt handler

Figure 29 illustrates an example of defining non-kernel interrupt handler in C language.

```c
#include <itrace.h>
#include <kernel.h>
#include "kernel_id.h"

void ConfigureInterrupts(void)
{
    int len = 1;
    int lic = 0x008;
}

void INT_INT1(void)
{
    // Processing starts...
    // Processing goes here...
    // Processing ends...
}
```

**Figure 29 Defining Interrupt Handler**
5.5 Coding Interrupts in Assembly Language

5.5.1 Defining Interrupt Vector Tables

Figure 30 illustrates the definition of vector tables in “asm_sec.inc” file for R8C/Lx devices.

```
;VECTOR TABLE
;-------------------------------
.glb _INT_VECTOR
 SECTION _INTERRUPT_VECTOR
 .org 0FD00H
__INT_VECTOR:

 SECTION FIX_INTERRUPT_VECTOR
 .org OFF8H
 ; reserved
 .addr OFFFFFFH
 ; OFF2
 .byte OFFH
 ; WDT disable
 .ofsreg OFFH

*Important: To disable Watchdog Timer Interrupt
```

Figure 30 Defining Vector Tables in “asm_sec.inc”

5.5.2 Writing Kernel Interrupt Handler

Step 1: Define kernel mask level

The kernel mask level is defined in configurator file (e.g. template.cfg) under the system definition. This step is identical to step 1 of coding interrupts in C language.

```
// System Definition
system{
    stack_size = 400;
    priority = 255;
    system_IPL = 5;
    tic_nume = 1;
    tic_denom = 1;
};
```

Figure 31 Defining Kernel Mask Level

Step 2: Define interrupt vector

```
interrupt_vector[25]{
    os_int = YES;
    entry_address = INT_INT1();
    pragma_switch = E;
};
```

Figure 32 Defining Interrupt Vector
Step 3: Define interrupt handler

```
..INCLUDE sfr_r835a.inc
..GLB ConfigureInterrupts
BSET Initlen
MOV.B #02H, intlic

..INCLUDE mr8c.inc
..GLB INT_INT1

INT_INT1:

PUSHM RO, AO
irel_mai #1D_task1
POPM RO, AO

ret_int
```

**Figure 33 Defining Interrupt Handler**

### 5.5.3 Writing Non-Kernel Interrupt Handler

Step 1: Define kernel mask level

```
// System Definition
system{
  stack_size = 400;
  priority = 256;
  system_IPL = 5;
  tic_nume = 1;
  tic_den0 = 1;
};
```

**Figure 34 Defining Kernel Mask Level**

Step 2: Define interrupt vector

```
interrupt_vector[25][
  os_int = No;
  entry_address = INT_INT1()
  pragma_switch = B;
];
```

**Figure 35 Defining Interrupt Vector**
Step 3: Define interrupt handler

```assembly
INCLUDE sfr_r835a.inc
.egl ConfigureInterrupts
BSET intlen
MOV.B #00H,intlic

INCLUDE mr8c.inc
.egl INT_INT1

INT_INT1:
PUSHM RO,A0
;Interrupt Processing
PDPW RO,A0
REIT

INT1 interrupt priority level defined at 6 (higher than system IPL)

Public declaration of interrupt handler
function name

Save registers to stack

Service calls cannot be issued within
the kernel interrupt handler

Registers are restored

REIT must be used to return from
interrupt handler

Figure 36 Defining Interrupt Handler
6. Reference Documents

User’s Manual

- MR8C/4 V1.00 User’s Manual
- R8C Family Hardware Manual

The latest version can be downloaded from the Renesas Technology website

Document

- Pardon the Interruption: Two Approaches to RTOS Interrupt Architectures (William E. Lamie)
Website and Support

Renesas Technology Website
- http://www.renesas.com/

Inquiries
- http://www.renesas.com/inquiry

All trademarks and registered trademarks are the property of their respective owners.
## Revision Record

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>March 01.10</td>
<td>—</td>
<td>First edition issued</td>
</tr>
</tbody>
</table>
General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this manual, refer to the relevant sections of the manual. If the descriptions under General Precautions in the Handling of MPU/MCU Products and in the body of the manual differ from each other, the description in the body of the manual takes precedence.

1. Handling of Unused Pins
   • Handle unused pins in accord 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 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. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on
   • The state of the product is undefined at the moment 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 moment 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 moment 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 moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses
   • Access to reserved addresses is prohibited.
     ⎯ The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals
   • After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has 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. Moreover, 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.

5. Differences between Products
   • Before changing from one product to another, i.e. to one with a different type number, confirm that the change will not lead to problems.
     ⎯ The characteristics of MPU/MCU in the same group but having different type numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to