R8C Family

Implementing Interrupts in MR8C/4

R20AN0088ES0100
Rev.1.00
Mar 01, 2010

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

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 1 Explanation of Document Topics

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

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)

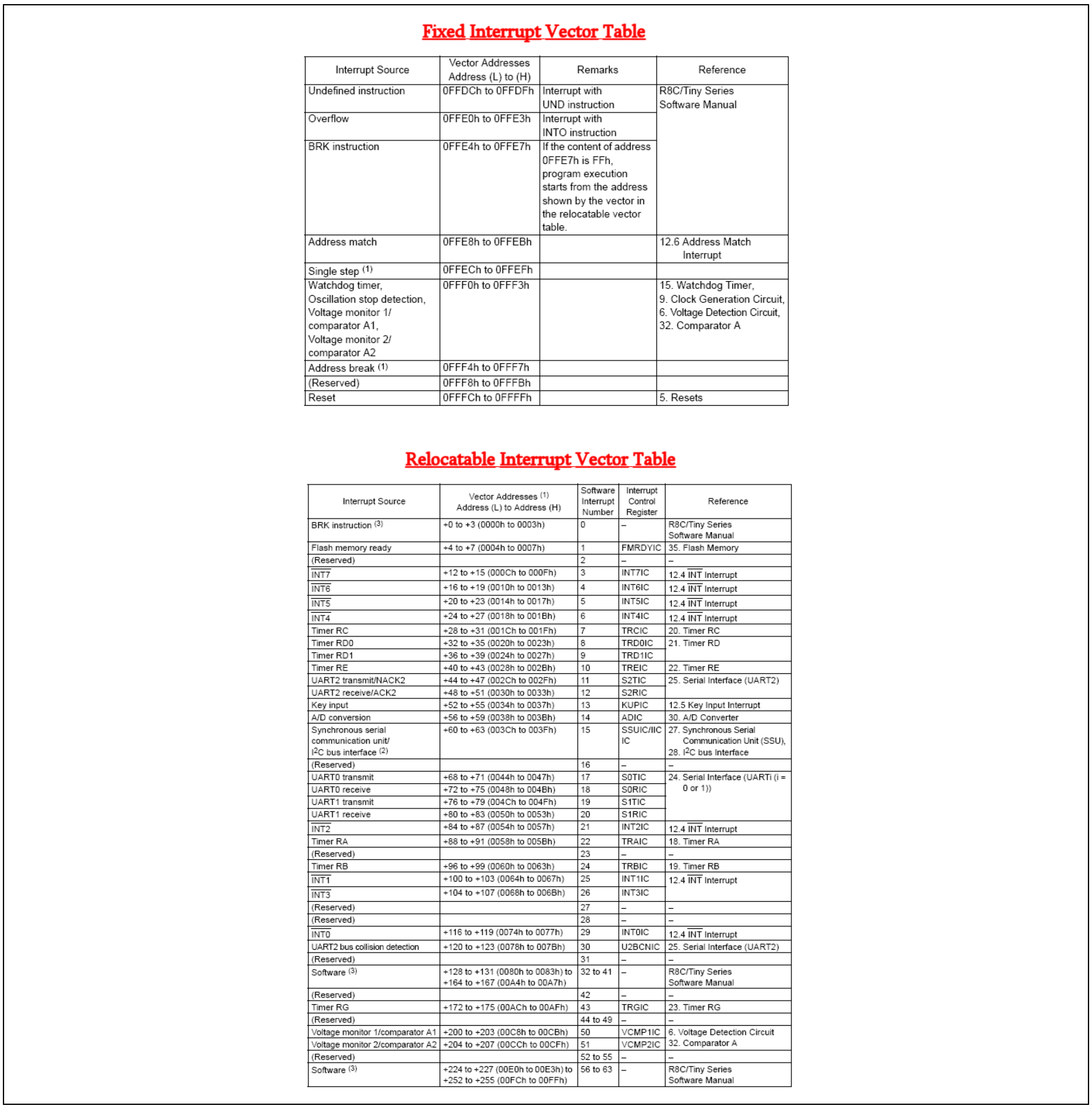
PIC is a device that accept and prioritize multiple interrupt sources so that the highest priority interrupt is directed to core CPU for processing at any point of time. In addition, PIC helps core CPU to determine an interrupt's exact source.

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.



Fixed Interrupt Vector Table

Interrupt Source	Vector Addresses Address (L) to (H)	Remarks	Reference
Undefined instruction	0FFDCh to 0FFDFh	Interrupt with UND instruction	R8C/Tiny Series Software Manual
Overflow	0FFE0h to 0FFE3h	Interrupt with INTO instruction	
BRK instruction	0FFE4h to 0FFE7h	If the content of address 0FFE7h is FFh, program execution starts from the address shown by the vector in the relocatable vector table.	
Address match	0FFE8h to 0FFEBh		12.6 Address Match Interrupt
Single step (1)	0FFECh to 0FFEFh		
Watchdog timer, Oscillation stop detection, Voltage monitor 1/ comparator A1, Voltage monitor 2/ comparator A2	0FFF0h to 0FFF3h		15. Watchdog Timer, 9. Clock Generation Circuit, 6. Voltage Detection Circuit, 32. Comparator A
Address break (1)	0FFF4h to 0FFF7h		
(Reserved)	0FFF8h to 0FFFBh		
Reset	0FFFCh to 0FFFFh		5. Resets

Relocatable Interrupt Vector Table

Interrupt Source	Vector Addresses (1) Address (L) to Address (H)	Software Interrupt Number	Interrupt Control Register	Reference
BRK instruction (3)	+0 to +3 (0000h to 0003h)	0	–	R8C/Tiny Series Software Manual
Flash memory ready	+4 to +7 (0004h to 0007h)	1	FMRDYIC	35. Flash Memory
(Reserved)		2	–	–
$\overline{INT7}$	+12 to +15 (000Ch to 000Fh)	3	INT7IC	12.4 $\overline{INT}$ Interrupt
$\overline{INT6}$	+16 to +19 (0010h to 0013h)	4	INT6IC	12.4 $\overline{INT}$ Interrupt
$\overline{INT5}$	+20 to +23 (0014h to 0017h)	5	INT5IC	12.4 $\overline{INT}$ Interrupt
$\overline{INT4}$	+24 to +27 (0018h to 001Bh)	6	INT4IC	12.4 $\overline{INT}$ Interrupt
Timer RC	+28 to +31 (001Ch to 001Fh)	7	TRCIC	20. Timer RC
Timer RD0	+32 to +35 (0020h to 0023h)	8	TRD0IC	21. Timer RD
Timer RD1	+36 to +39 (0024h to 0027h)	9	TRD1IC	
Timer RE	+40 to +43 (0028h to 002Bh)	10	TREIC	22. Timer RE
UART2 transmit/NACK2	+44 to +47 (002Ch to 002Fh)	11	S2TIC	25. Serial Interface (UART2)
UART2 receive/ACK2	+48 to +51 (0030h to 0033h)	12	S2RIC	
Key input	+52 to +55 (0034h to 0037h)	13	KUPIC	12.5 Key Input Interrupt
A/D conversion	+56 to +59 (0038h to 003Bh)	14	ADIC	30. A/D Converter
Synchronous serial communication unit/ I^2C bus interface (2)	+60 to +63 (003Ch to 003Fh)	15	SSUIC/IIC IC	27. Synchronous Serial Communication Unit (SSU), 28. I^2C bus Interface
(Reserved)		16	–	–
UART0 transmit	+68 to +71 (0044h to 0047h)	17	S0TIC	24. Serial Interface (UARTi (i = 0 or 1))
UART0 receive	+72 to +75 (0048h to 004Bh)	18	S0RIC	
UART1 transmit	+76 to +79 (004Ch to 004Fh)	19	S1TIC	
UART1 receive	+80 to +83 (0050h to 0053h)	20	S1RIC	
$\overline{INT2}$	+84 to +87 (0054h to 0057h)	21	INT2IC	12.4 $\overline{INT}$ Interrupt
Timer RA	+88 to +91 (0058h to 005Bh)	22	TRAIC	18. Timer RA
(Reserved)		23	–	–
Timer RB	+96 to +99 (0060h to 0063h)	24	TRBIC	19. Timer RB
$\overline{INT1}$	+100 to +103 (0064h to 0067h)	25	INT1IC	12.4 $\overline{INT}$ Interrupt
$\overline{INT3}$	+104 to +107 (0068h to 006Bh)	26	INT3IC	
(Reserved)		27	–	–
(Reserved)		28	–	–
$\overline{INT0}$	+116 to +119 (0074h to 0077h)	29	INT0IC	12.4 $\overline{INT}$ Interrupt
UART2 bus collision detection	+120 to +123 (0078h to 007Bh)	30	U2BCNIC	25. Serial Interface (UART2)
(Reserved)		31	–	–
Software (3)	+128 to +131 (0080h to 0083h) to +164 to +167 (00A4h to 00A7h)	32 to 41	–	R8C/Tiny Series Software Manual
(Reserved)		42	–	–
Timer RG	+172 to +175 (00ACh to 00AFh)	43	TRGIC	23. Timer RG
(Reserved)		44 to 49	–	–
Voltage monitor 1/comparator A1	+200 to +203 (00C8h to 00CBh)	50	VCMP1IC	6. Voltage Detection Circuit 32. Comparator A
Voltage monitor 2/comparator A2	+204 to +207 (00CCh to 00CFh)	51	VCMP2IC	
(Reserved)		52 to 55	–	–
Software (3)	+224 to +227 (00E0h to 00E3h) to +252 to +255 (00FCh to 00FFh)	56 to 63	–	R8C/Tiny Series Software Manual

Figure 1 Interrupts Vector Tables of R8C/Lx Devices

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 represents 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).

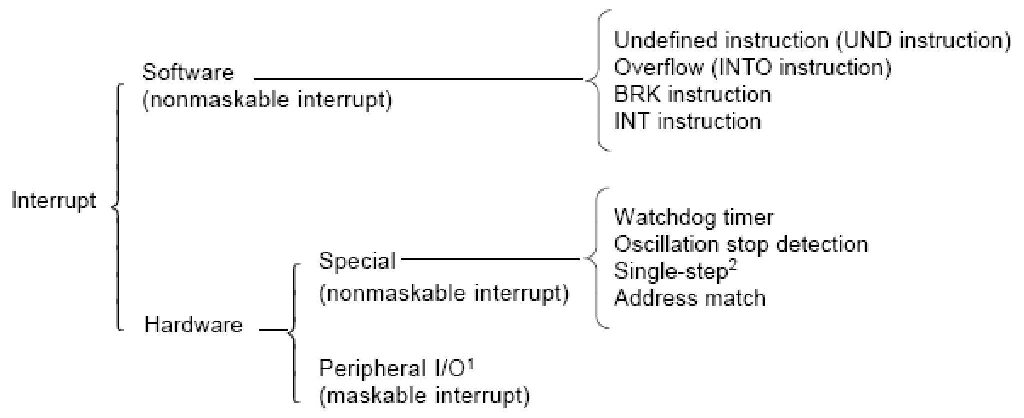
IRQs are interrupts that can be disabled to let higher priority ISRs be executed uninterruptedly. Interrupt masking is usually done by clearing the Interrupt Enable Flag.

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.



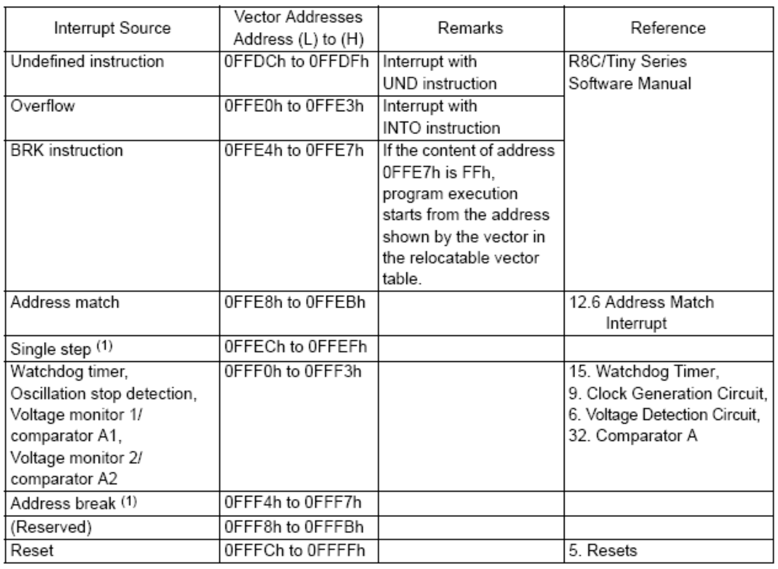
Notes: 1. Peripheral function interrupts are generated by the peripheral functions built into the microcomputer system.
2. This is a dedicated interrupt for development support tools. Do not use this interrupt.

Figure 2 Classifications of Interrupts

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

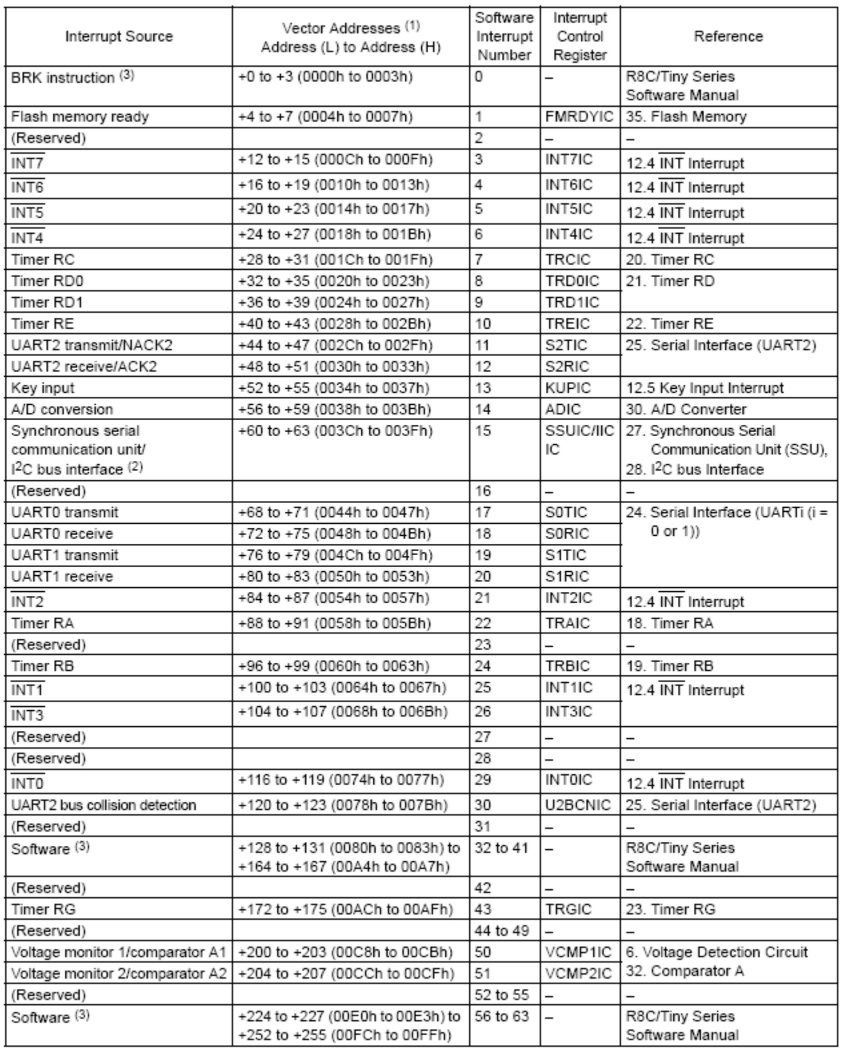
As the names imply, “Fixed Vector Table” resides in allocated addresses (e.g. 0FFDCh to 0FFFFh). Whereas “Relocatable Vector Table” can be allocated at any desired location within the devices’ entire memory space by INTB register relative addressing.

Figure 3 and Figure 4 provide the “Fixed Vector Table” and “Relocatable Vector Table” of R8C/Lx devices respectively.



Interrupt Source	Vector Addresses Address (L) to (H)	Remarks	Reference
Undefined instruction	0FFDCh to 0FFDFh	Interrupt with UND instruction	R8C/Tiny Series Software Manual
Overflow	0FFE0h to 0FFE3h	Interrupt with INTO instruction	
BRK instruction	0FFE4h to 0FFE7h	If the content of address 0FFE7h is FFh, program execution starts from the address shown by the vector in the relocatable vector table.	
Address match	0FFE8h to 0FFEBh		12.6 Address Match Interrupt
Single step (1)	0FFECh to 0FFEFh		
Watchdog timer, Oscillation stop detection, Voltage monitor 1/ comparator A1, Voltage monitor 2/ comparator A2	0FFF0h to 0FFF3h		15. Watchdog Timer, 9. Clock Generation Circuit, 6. Voltage Detection Circuit, 32. Comparator A
Address break (1)	0FFF4h to 0FFF7h		
(Reserved)	0FFF8h to 0FFFBh		
Reset	0FFFCh to 0FFFFh		5. Resets

Figure 3 Fixed Vector Tables of R8C/Lx Devices



Interrupt Source	Vector Addresses (1) Address (L) to Address (H)	Software Interrupt Number	Interrupt Control Register	Reference
BRK instruction (3)	+0 to +3 (0000h to 0003h)	0	–	R8C/Tiny Series Software Manual
Flash memory ready	+4 to +7 (0004h to 0007h)	1	FMRDYIC	35. Flash Memory
(Reserved)		2	–	–
$\overline{\text{INT7}}$	+12 to +15 (000Ch to 000Fh)	3	INT7IC	12.4 $\overline{\text{INT}}$ Interrupt
$\overline{\text{INT6}}$	+16 to +19 (0010h to 0013h)	4	INT6IC	12.4 $\overline{\text{INT}}$ Interrupt
$\overline{\text{INT5}}$	+20 to +23 (0014h to 0017h)	5	INT5IC	12.4 $\overline{\text{INT}}$ Interrupt
$\overline{\text{INT4}}$	+24 to +27 (0018h to 001Bh)	6	INT4IC	12.4 $\overline{\text{INT}}$ Interrupt
Timer RC	+28 to +31 (001Ch to 001Fh)	7	TRCIC	20. Timer RC
Timer RD0	+32 to +35 (0020h to 0023h)	8	TRD0IC	21. Timer RD
Timer RD1	+36 to +39 (0024h to 0027h)	9	TRD1IC	
Timer RE	+40 to +43 (0028h to 002Bh)	10	TREIC	22. Timer RE
UART2 transmit/NACK2	+44 to +47 (002Ch to 002Fh)	11	S2TIC	25. Serial Interface (UART2)
UART2 receive/ACK2	+48 to +51 (0030h to 0033h)	12	S2RIC	
Key input	+52 to +55 (0034h to 0037h)	13	KUPIC	12.5 Key Input Interrupt
A/D conversion	+56 to +59 (0038h to 003Bh)	14	ADIC	30. A/D Converter
Synchronous serial communication unit/ I2C bus interface (2)	+60 to +63 (003Ch to 003Fh)	15	SSUIC/IICIC	27. Synchronous Serial Communication Unit (SSU), 28. I2C bus Interface
(Reserved)		16	–	–
UART0 transmit	+68 to +71 (0044h to 0047h)	17	S0TIC	24. Serial Interface (UARTi (i = 0 or 1))
UART0 receive	+72 to +75 (0048h to 004Bh)	18	S0RIC	
UART1 transmit	+76 to +79 (004Ch to 004Fh)	19	S1TIC	
UART1 receive	+80 to +83 (0050h to 0053h)	20	S1RIC	
$\overline{\text{INT2}}$	+84 to +87 (0054h to 0057h)	21	INT2IC	12.4 $\overline{\text{INT}}$ Interrupt
Timer RA	+88 to +91 (0058h to 005Bh)	22	TRAIC	18. Timer RA
(Reserved)		23	–	–
Timer RB	+96 to +99 (0060h to 0063h)	24	TRBIC	19. Timer RB
$\overline{\text{INT1}}$	+100 to +103 (0064h to 0067h)	25	INT1IC	12.4 $\overline{\text{INT}}$ Interrupt
$\overline{\text{INT3}}$	+104 to +107 (0068h to 006Bh)	26	INT3IC	
(Reserved)		27	–	–
(Reserved)		28	–	–
$\overline{\text{INT0}}$	+116 to +119 (0074h to 0077h)	29	INT0IC	12.4 $\overline{\text{INT}}$ Interrupt
UART2 bus collision detection	+120 to +123 (0078h to 007Bh)	30	U2BCNIC	25. Serial Interface (UART2)
(Reserved)		31	–	–
Software (3)	+128 to +131 (0080h to 0083h) to +164 to +167 (00A4h to 00A7h)	32 to 41	–	R8C/Tiny Series Software Manual
(Reserved)		42	–	–
Timer RG	+172 to +175 (00ACh to 00AFh)	43	TRGIC	23. Timer RG
(Reserved)		44 to 49	–	–
Voltage monitor 1/comparator A1	+200 to +203 (00C8h to 00CBh)	50	VCMP1IC	6. Voltage Detection Circuit 32. Comparator A
Voltage monitor 2/comparator A2	+204 to +207 (00CCh to 00CFh)	51	VCMP2IC	
(Reserved)		52 to 55	–	–
Software (3)	+224 to +227 (00E0h to 00E3h) to +252 to +255 (00FCh to 00FFh)	56 to 63	–	R8C/Tiny Series Software Manual

Figure 4 Relocatable Vector Tables of R8C/Lx Devices

Figure 5 illustrates an example of defining the starting address locations of “Fixed Vector Table” and “Relocatable Vector Table” for a R8C23 device.

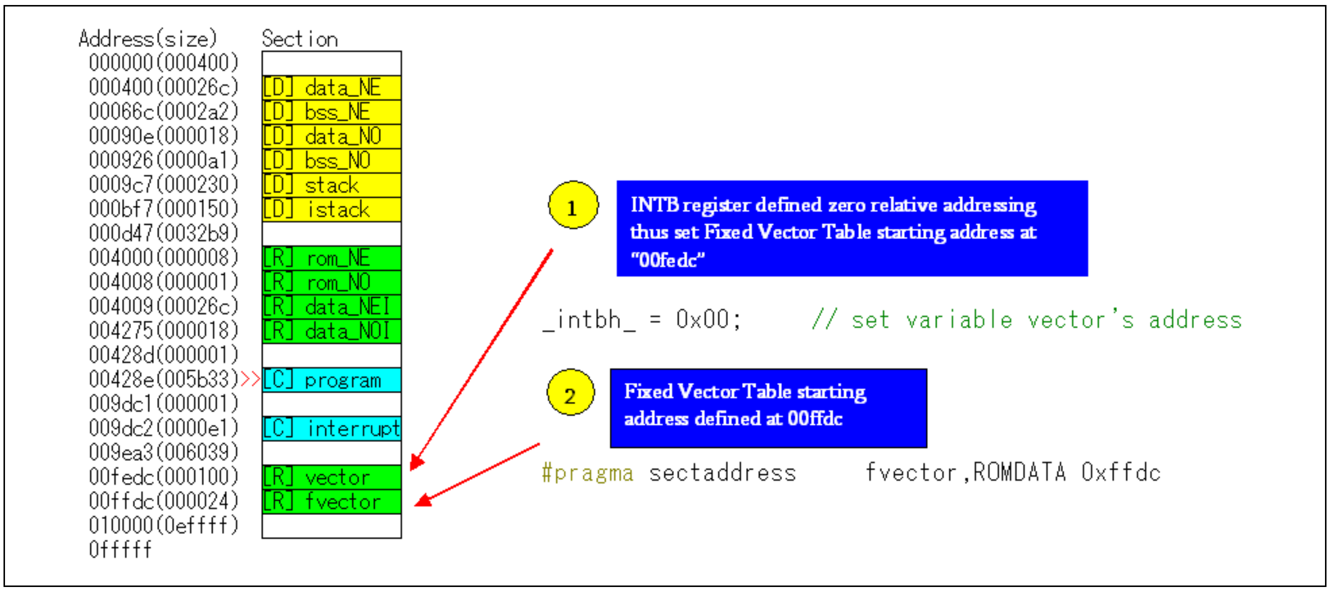
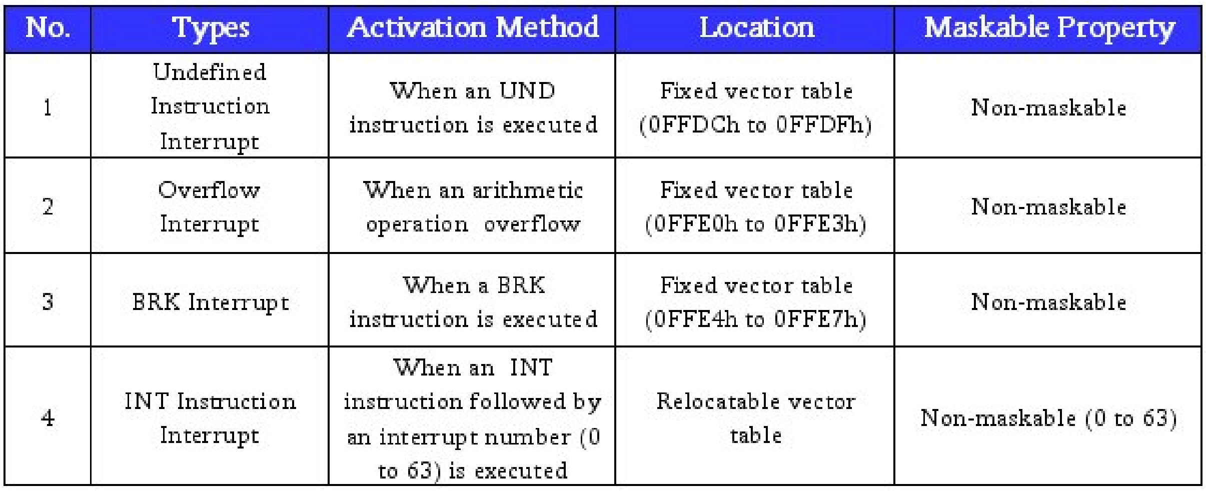

Address(size) Section
000000(000400)
000400(00026c) [D] data_NE
00066c(0002a2) [D] bss_NE
00090e(000018) [D] data_NO
000926(0000a1) [D] bss_NO
0009c7(000230) [D] stack
000bf7(000150) [D] istack
000d47(0032b9)
004000(000008) [R] rom_NE
004008(000001) [R] rom_NO
004009(00026c) [R] data_NEI
004275(000018) [R] data_NOI
00428d(000001)
00428e(005b33)>> [C] program
009dc1(000001)
009dc2(0000e1) [C] interrupt
009ea3(006039)
00fedc(000100) [R] vector
00ffdc(000024) [R] fvector
010000(0effff)
0fffff
1
INTB register defined zero relative addressing thus set Fixed Vector Table starting address at "00fedc"
intbh = 0x00; // set variable vector's address
2
Fixed Vector Table starting address defined at 00ffdc
#pragma sectaddress fvector,ROMDATA 0xffdc

Figure 5 Vector Tables Definition for 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.



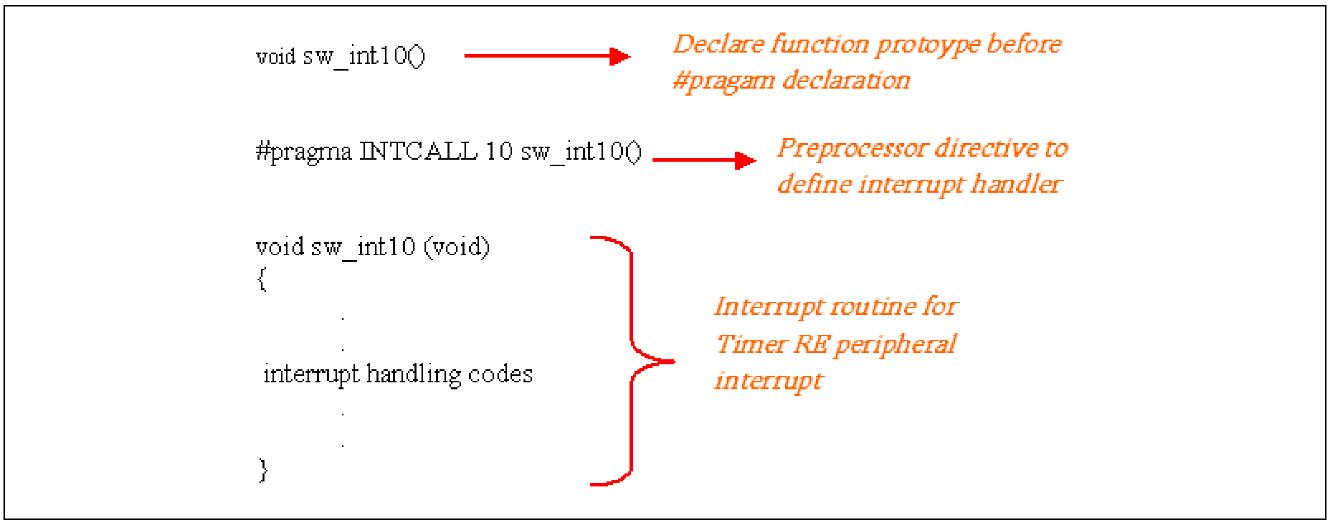
No.	Types	Activation Method	Location	Maskable Property
1	Undefined Instruction Interrupt	When an UND instruction is executed	Fixed vector table (0FFDCh to 0FFDFh)	Non-maskable
2	Overflow Interrupt	When an arithmetic operation overflow	Fixed vector table (0FFE0h to 0FFE3h)	Non-maskable
3	BRK Interrupt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

Figure 6 Types of Software Interrupts

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.



```
void sw_int10()

#pragma INTCALL 10 sw_int10()

void sw_int10 (void)
{
        .
        .
 interrupt handling codes
        .
        .
}
```

Declare function protoype before #pragam declaration
Preprocessor directive to define interrupt handler
Interrupt routine for Timer RE peripheral interrupt

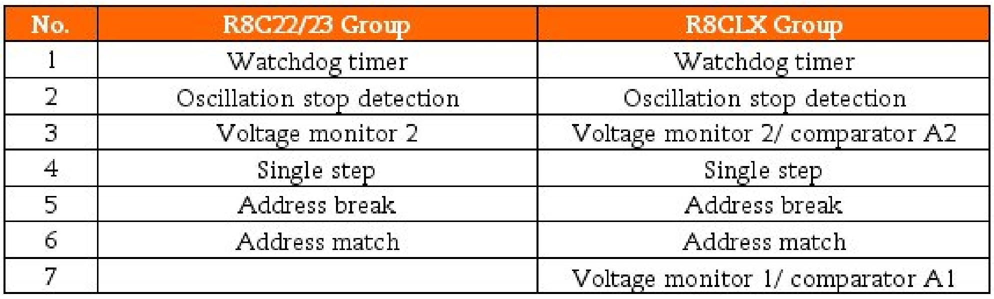
Figure 7 Writing Software Interrupts

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.



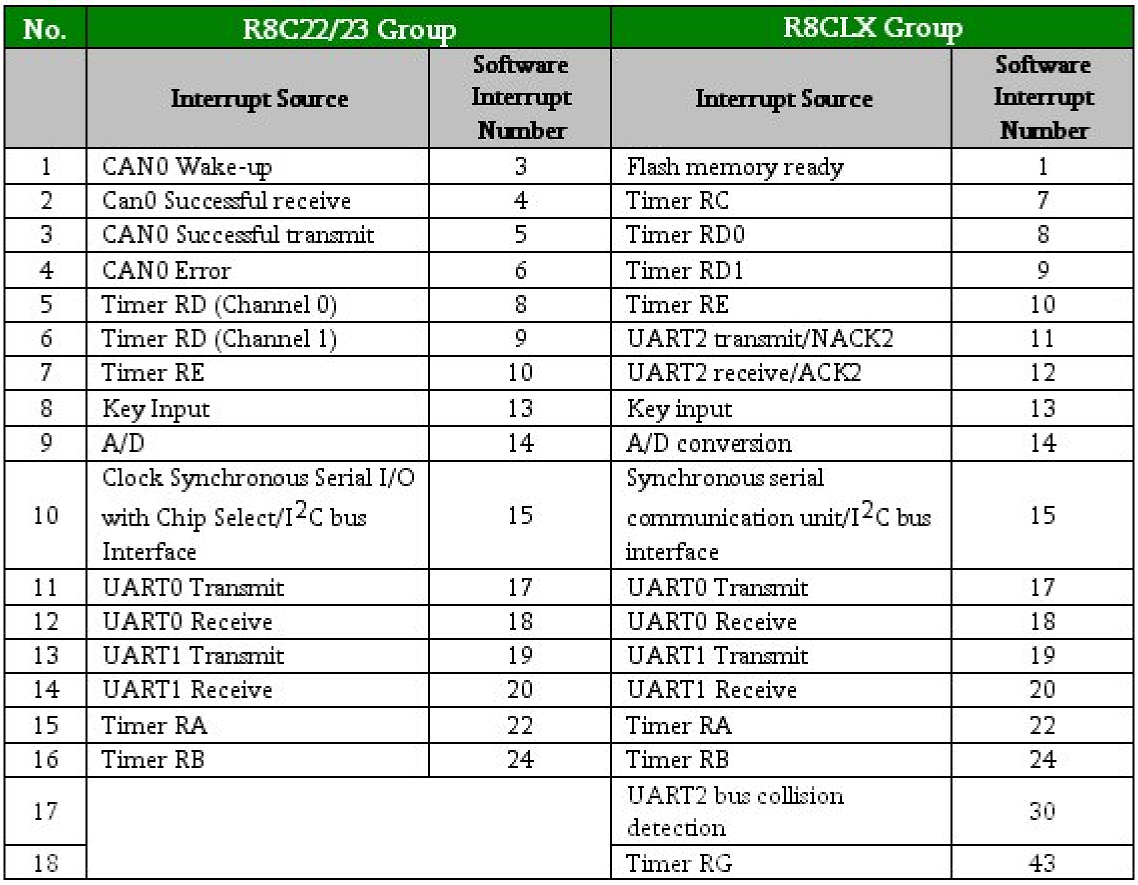
No.	R8C22/23 Group	R8CLX Group
1	Watchdog timer	Watchdog timer
2	Oscillation stop detection	Oscillation stop detection
3	Voltage monitor 2	Voltage monitor 2/ comparator A2
4	Single step	Single step
5	Address break	Address break
6	Address match	Address match
7		Voltage monitor 1/ comparator A1

Figure 8 Comparing Special Interrupts of R8C22/23 and R8C/Lx Devices

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.



No.	R8C22/23 Group		R8CLX Group	
	Interrupt Source	Software Interrupt Number	Interrupt Source	Software Interrupt Number
1	CAN0 Wake-up	3	Flash memory ready	1
2	Can0 Successful receive	4	Timer RC	7
3	CAN0 Successful transmit	5	Timer RD0	8
4	CAN0 Error	6	Timer RD1	9
5	Timer RD (Channel 0)	8	Timer RE	10
6	Timer RD (Channel 1)	9	UART2 transmit/NACK2	11
7	Timer RE	10	UART2 receive/ACK2	12
8	Key Input	13	Key input	13
9	A/D	14	A/D conversion	14
10	Clock Synchronous Serial I/O with Chip Select/I^{2}C bus Interface	15	Synchronous serial communication unit/I^{2}C bus interface	15
11	UART0 Transmit	17	UART0 Transmit	17
12	UART0 Receive	18	UART0 Receive	18
13	UART1 Transmit	19	UART1 Transmit	19
14	UART1 Receive	20	UART1 Receive	20
15	Timer RA	22	Timer RA	22
16	Timer RB	24	Timer RB	24
17			UART2 bus collision detection	30
18			Timer RG	43

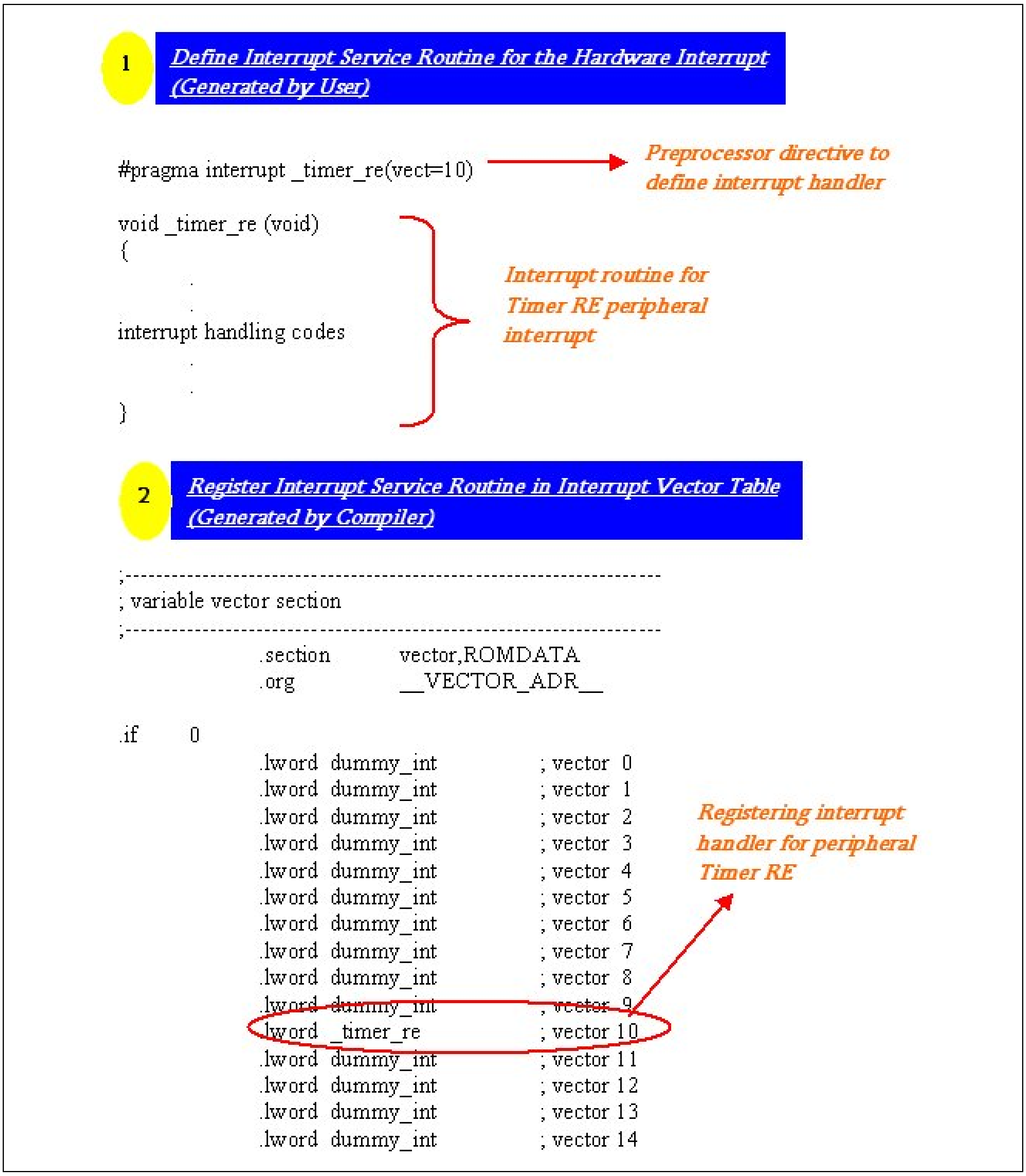
Figure 9 Comparing Peripheral I/O Interrupts of R8C22/23 and R8C/Lx Devices

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)


1
Define Interrupt Service Routine for the Hardware Interrupt (Generated by User)
Preprocessor directive to define interrupt handler
Interrupt routine for Timer RE peripheral interrupt
2
Register Interrupt Service Routine in Interrupt Vector Table (Generated by Compiler)
Registering interrupt handler for peripheral Timer RE

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

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

```
;------------------------------------------------------------------
; 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
```

Figure 10 Writing Hardware Interrupts

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).

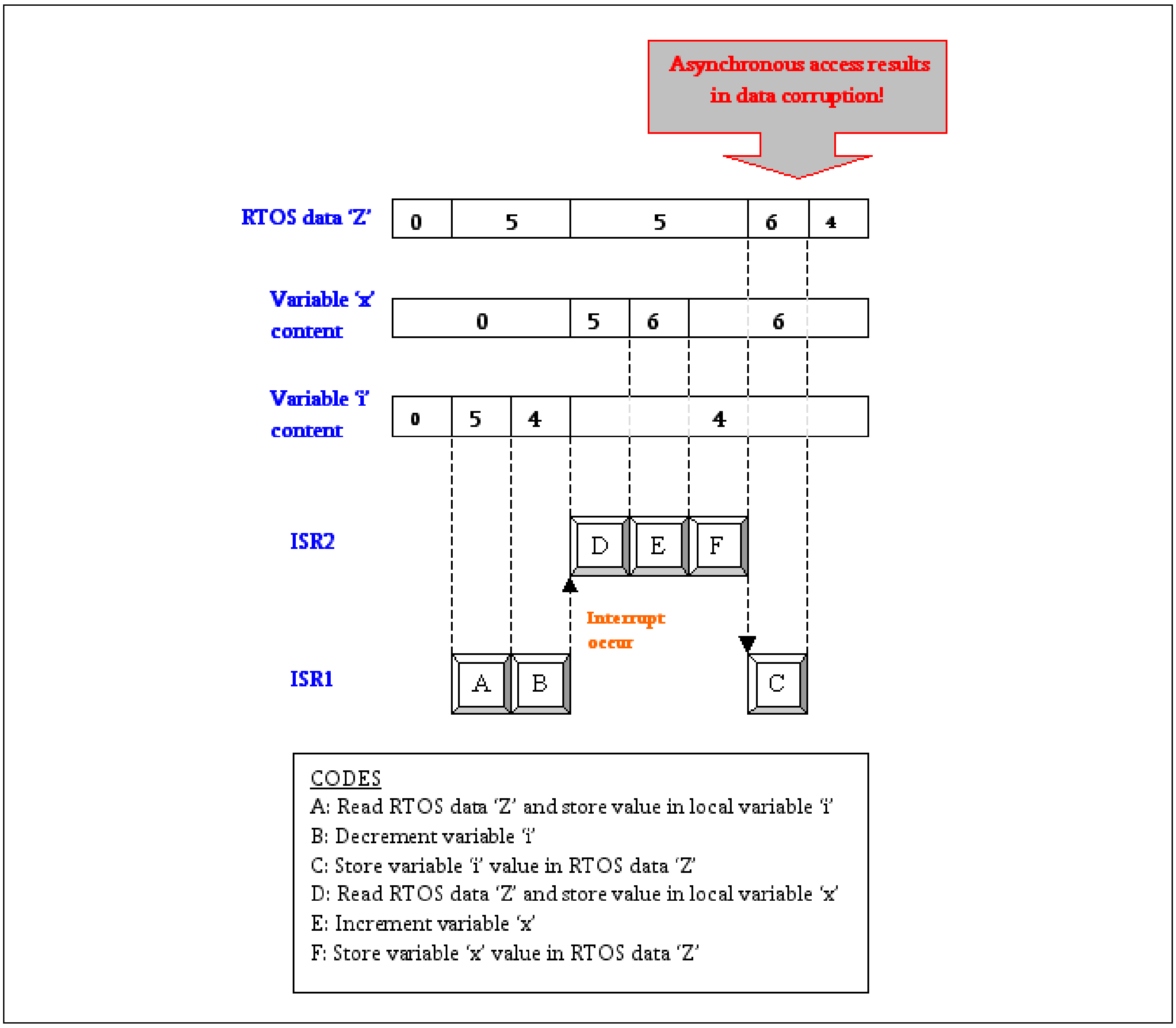
The RTOS layer, being the first layer, functions as an interrupt handler in accepting the control of an interrupt when it occurred. Upon receiving the control, the RTOS interrupts handler calls the application layer interrupt service routine corresponding to the interrupt for processing.

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.



CODES
A: Read RTOS data 'Z' and store value in local variable 'i'
B: Decrement variable 'i'
C: Store variable 'i' value in RTOS data 'Z'
D: Read RTOS data 'Z' and store value in local variable 'x'
E: Increment variable 'x'
F: Store variable 'x' value in RTOS data 'Z'

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.

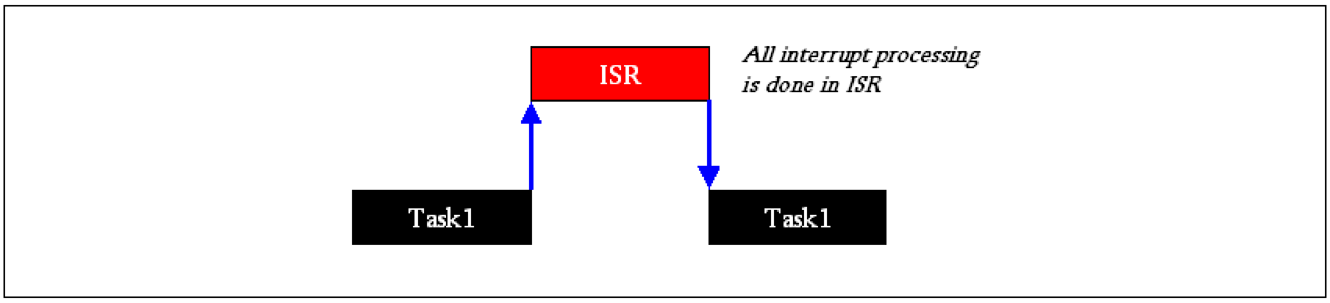


Figure 12 Unified Interrupt Architecture

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.

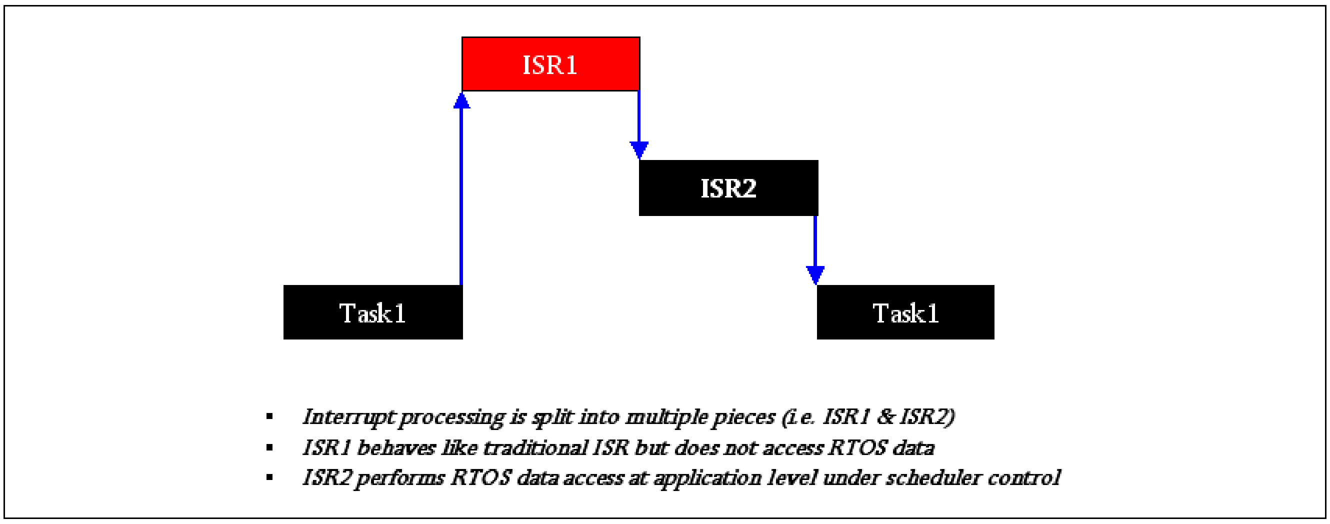

ISR1
ISR2
Task1
Task1
Interrupt processing is split into multiple pieces (i.e. ISR1 & ISR2)
ISR1 behaves like traditional ISR but does not access RTOS data
ISR2 performs RTOS data access at application level under scheduler control

Figure 13 Segmented Interrupt Architecture

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)

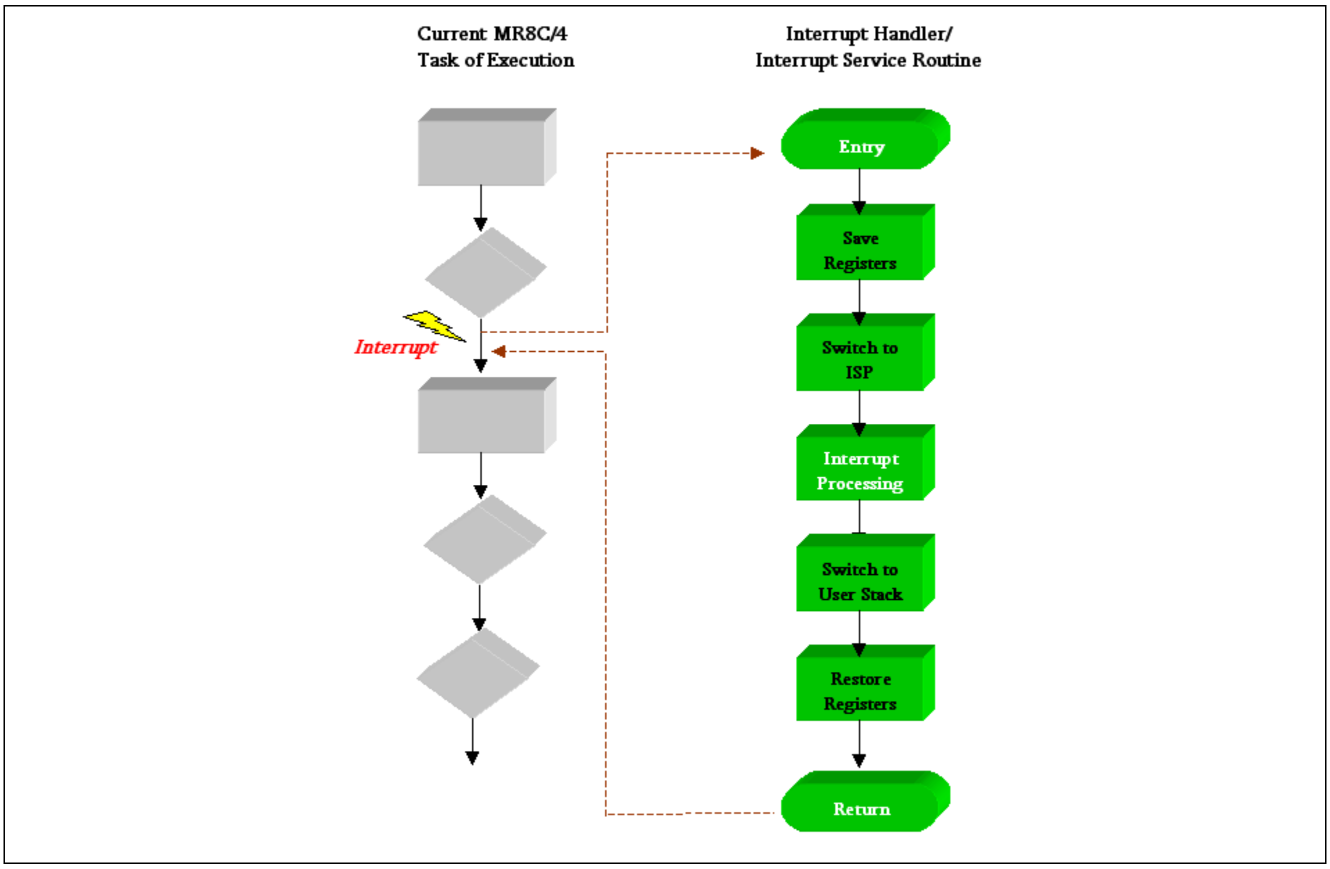


Figure 14 Interrupt Occurring when Processing Task

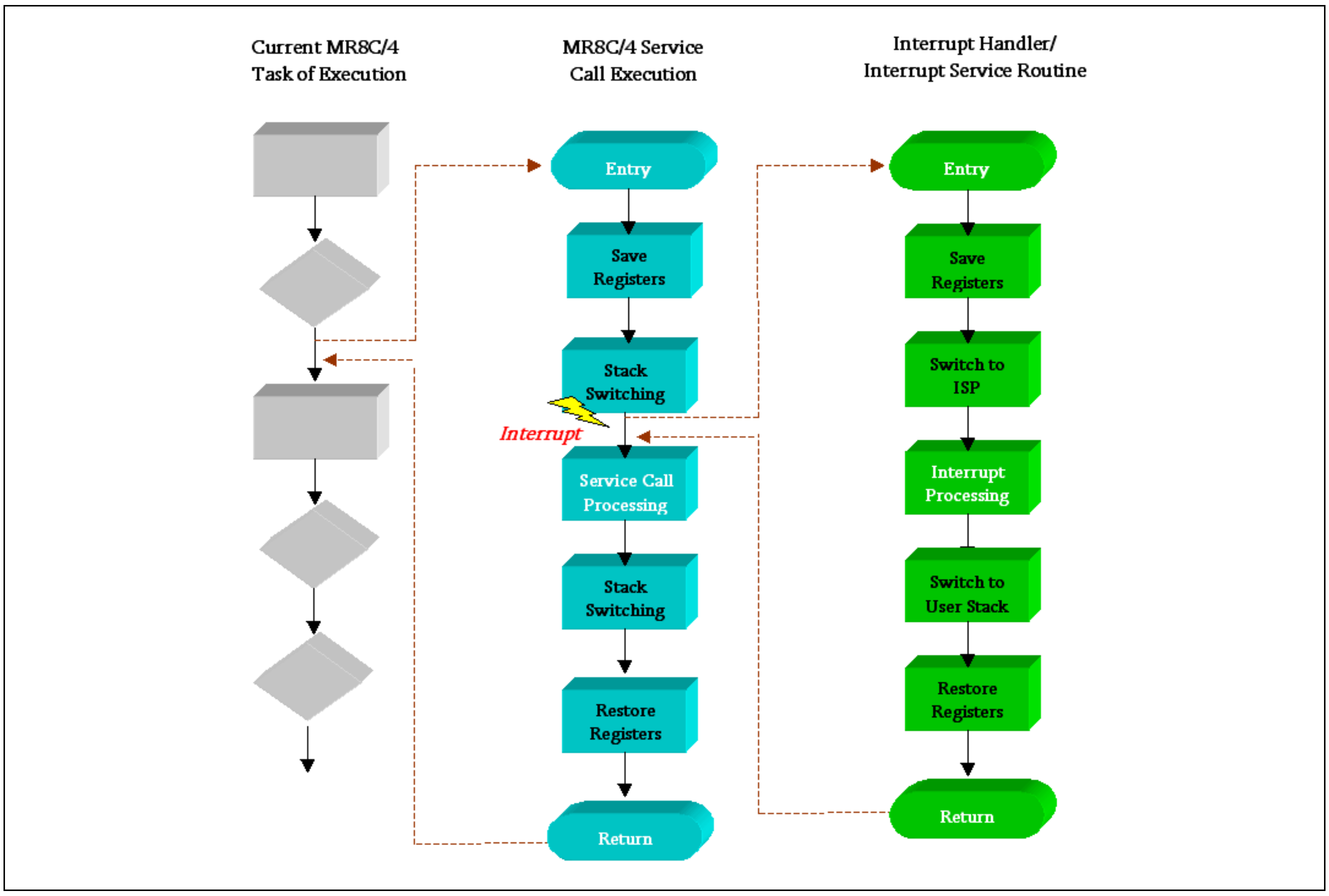


Figure 15 Interrupt Occurring when Processing Service Call

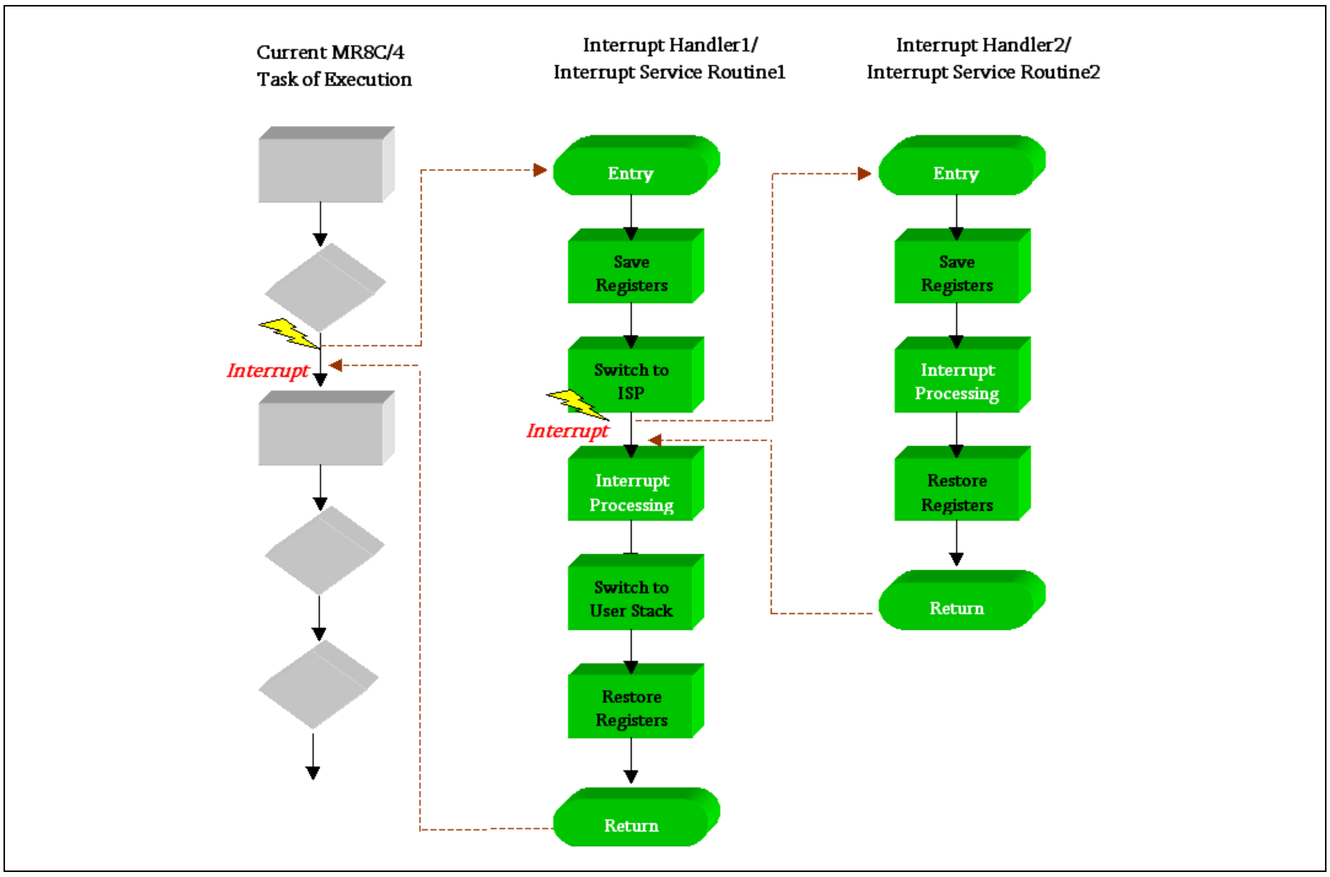


Figure 16 Interrupt Occurring within Interrupt Handler

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.

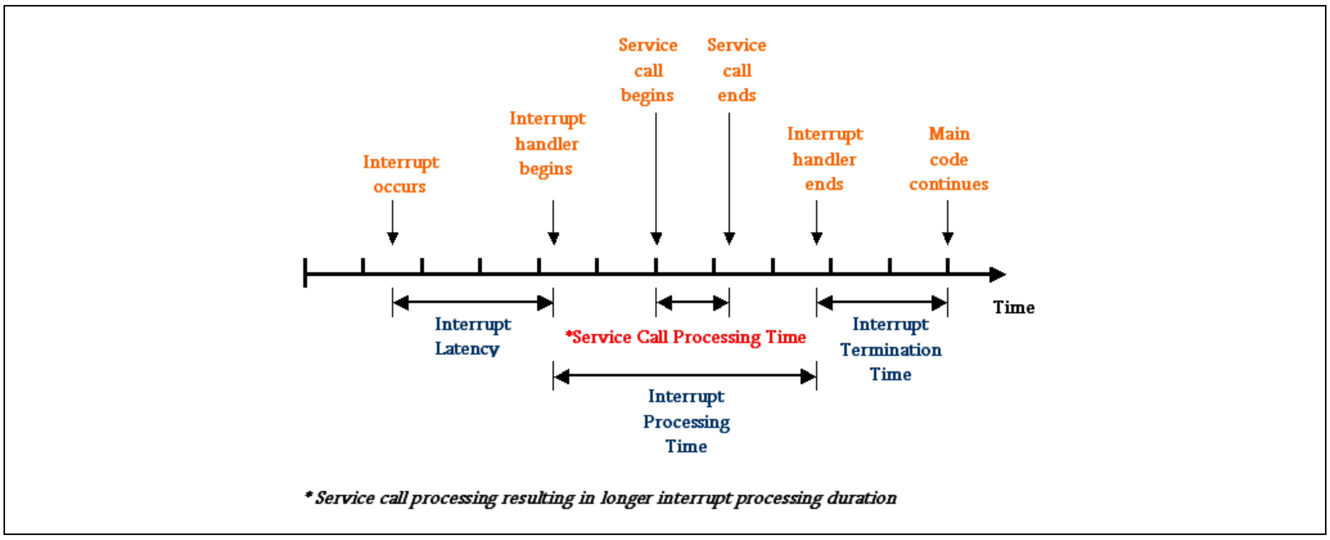

Service call begins
Service call ends
Interrupt handler begins
Interrupt occurs
Interrupt handler ends
Main code continues
Time
Interrupt Latency
*Service Call Processing Time
Interrupt Termination Time
Interrupt Processing Time
* Service call processing resulting in longer interrupt processing duration

Figure 17 Kernel Interrupt Service Routine with Service Call

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).

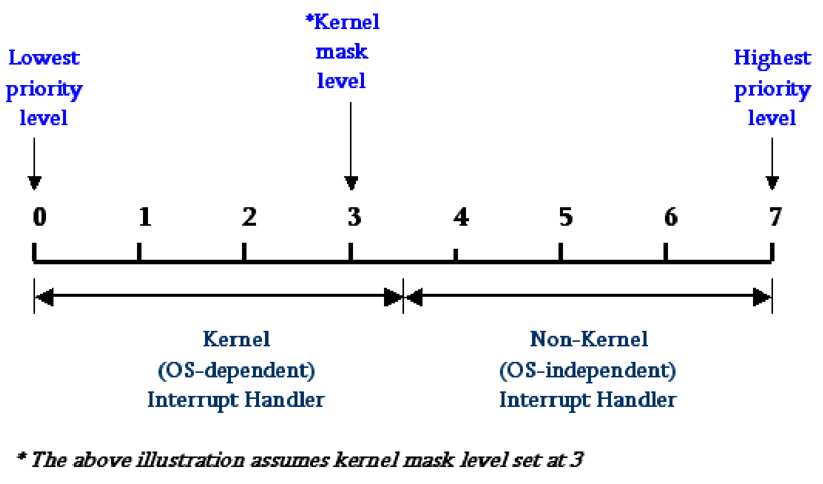
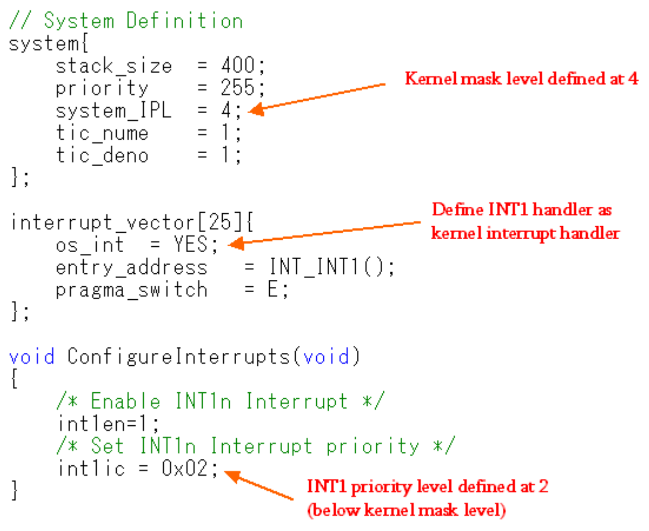

*Kernel mask level
Lowest priority level
Highest priority level
0 1 2 3 4 5 6 7
Kernel (OS-dependent) Interrupt Handler
Non-Kernel (OS-independent) Interrupt Handler
* The above illustration assumes kernel mask level set at 3

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_nume    = 1;
    tic_deno    = 1;
];

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

void ConfigureInterrupts(void)
[
    /* Enable INT1n Interrupt */
    int1en=1;
    /* Set INT1n Interrupt priority */
    int1ic = 0x02;
]
```

Kernel mask level defined at 4
Define INT1 handler as kernel interrupt handler
INT1 priority level defined at 2 (below kernel mask level)

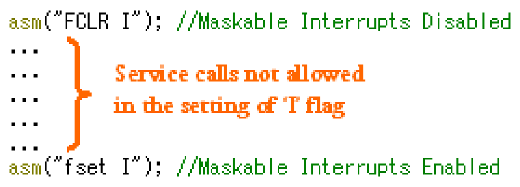
Figure 19 Defining INT1 as Kernel Interrupt Handler

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. Inaddition, 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.



```
asm("FCLR I"); //Maskable Interrupts Disabled
...
...
...
...
...
asm("fset 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

```
// System Definition
system{
    stack_size  = 400;
    priority    = 255;
    system_IPL  = 4;
    tic_nume    = 1;
    tic_deno    = 1;
};

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

2 TRBIC interrupt control register of R8CLX

Bit	b7	b6	b5	b4	b3	b2	b1	b0
Symbol	—	—	—	—	IR	ILVL2	ILVL1	ILVL0
After Reset	X	X	X	X	X	0	0	0

Bit	Symbol	Bit Name	Function	R/W
b0	ILVL0	Interrupt priority level select bit	b2 b1 b0 0 0 0: Level 0 (interrupt disabled) 0 0 1: Level 1 0 1 0: Level 2 0 1 1: Level 3 1 0 0: Level 4 1 0 1: Level 5 1 1 0: Level 6 1 1 1: Level 7	R/W
b1	ILVL1			R/W
b2	ILVL2			R/W
b3	IR	Interrupt request bit	0: No interrupt requested 1: Interrupt requested	R/W (1)
b4	—	Nothing is assigned. If necessary, set to 0. When read, the content is undefined.		—
b5	—			
b6	—			
b7	—			

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

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

void task1(VP_INT stacd)
{
    trbic = 0;      ← Timer RB interrupt disabled
    ...
    ...
    ..
    ...
    trbic = 5;      ← Time 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

```
// System Definition
system{
    stack_size  = 400;
    priority    = 255;
    system_IPL  = 4;
    tic_nume    = 1;
    tic_deno    = 1;
};

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

2 INT1 interrupt control register of R8CLX

Bit	b7	b6	b5	b4	b3	b2	b1	b0
Symbol	—	—	—	POL	IR	ILVL2	ILVL1	ILVL0
After Reset	X	X	0	0	X	0	0	0

Bit	Symbol	Bit Name	Function	R/W
b0	ILVL0	Interrupt priority level select bit	b2 b1 b0 0 0 0: Level 0 (interrupt disabled) 0 0 1: Level 1 0 1 0: Level 2 0 1 1: Level 3 1 0 0: Level 4 1 0 1: Level 5 1 1 0: Level 6 1 1 1: Level 7	R/W
b1	ILVL1			R/W
b2	ILVL2			R/W
b3	IR	Interrupt request bit	0: No interrupt requested 1: Interrupt requested	R/W (1)
b4	POL	Polarity switch bit (3)	0: Falling edge selected 1: Rising edge selected (2)	R/W
b5	—	Reserved bit	Set to 0.	R/W
b6	—	Nothing is assigned. If necessary, set to 0. When read, the content is undefined.		—
b7	—			

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

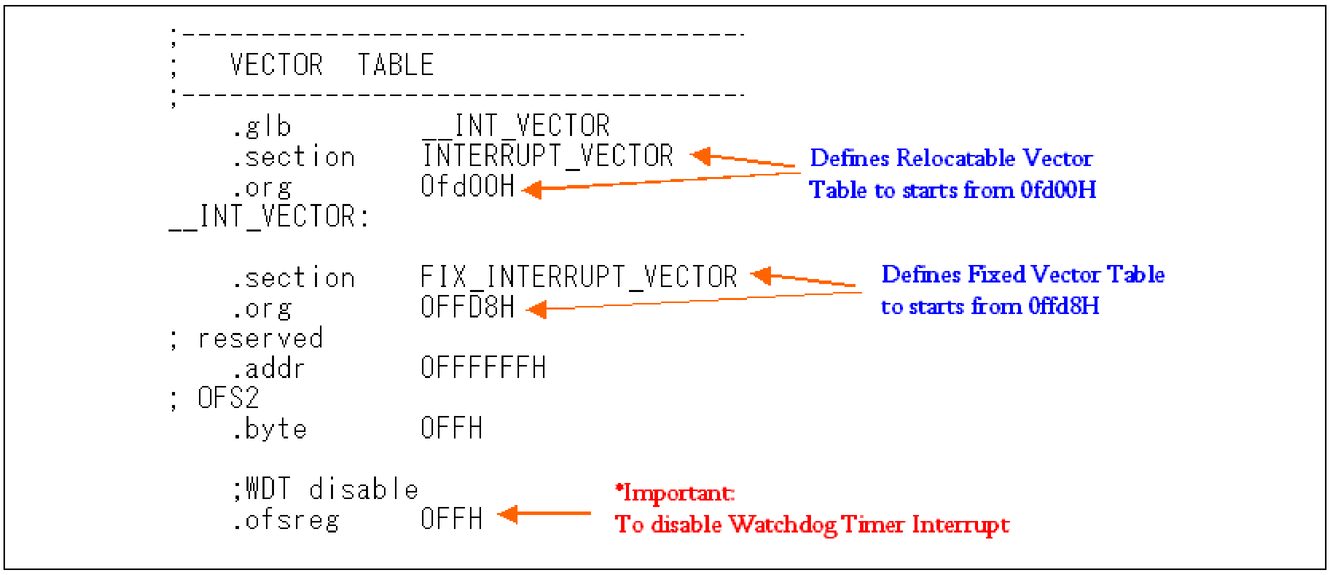
```
#include <itron.h>
#include <kernel.h>
#include "kernel_id.h"

void task1(VP_INT stacd)
{
    int1ic = 0x03;     /*Define INT 1 priority 3 */
    ercd = loc_cpu();  ←—— Kernel interrupts disabled
    ...
    ...
    ...
    ercd = 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.



```
;------------------------------------
;   VECTOR  TABLE
;------------------------------------
    .glb          __INT_VECTOR
    .section      INTERRUPT_VECTOR
    .org          0fd00H
__INT_VECTOR:

    .section      FIX_INTERRUPT_VECTOR
    .org          0FFD8H
; reserved
    .addr         0FFFFFFH
; OFS2
    .byte         0FFH

    ;WDT disable
    .ofsreg       0FFH
```

Defines Relocatable Vector Table to starts from 0fd00H

Defines Fixed Vector Table to starts from 0ffd8H

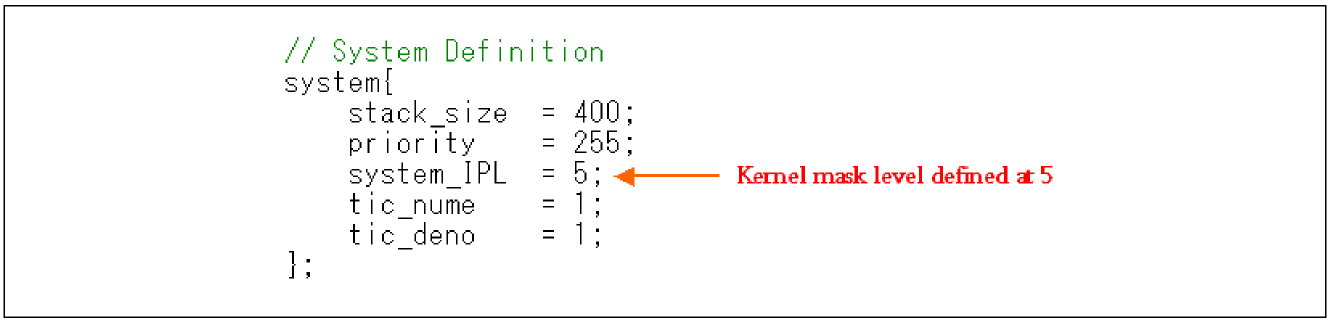
*Important: To disable Watchdog Timer Interrupt

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.



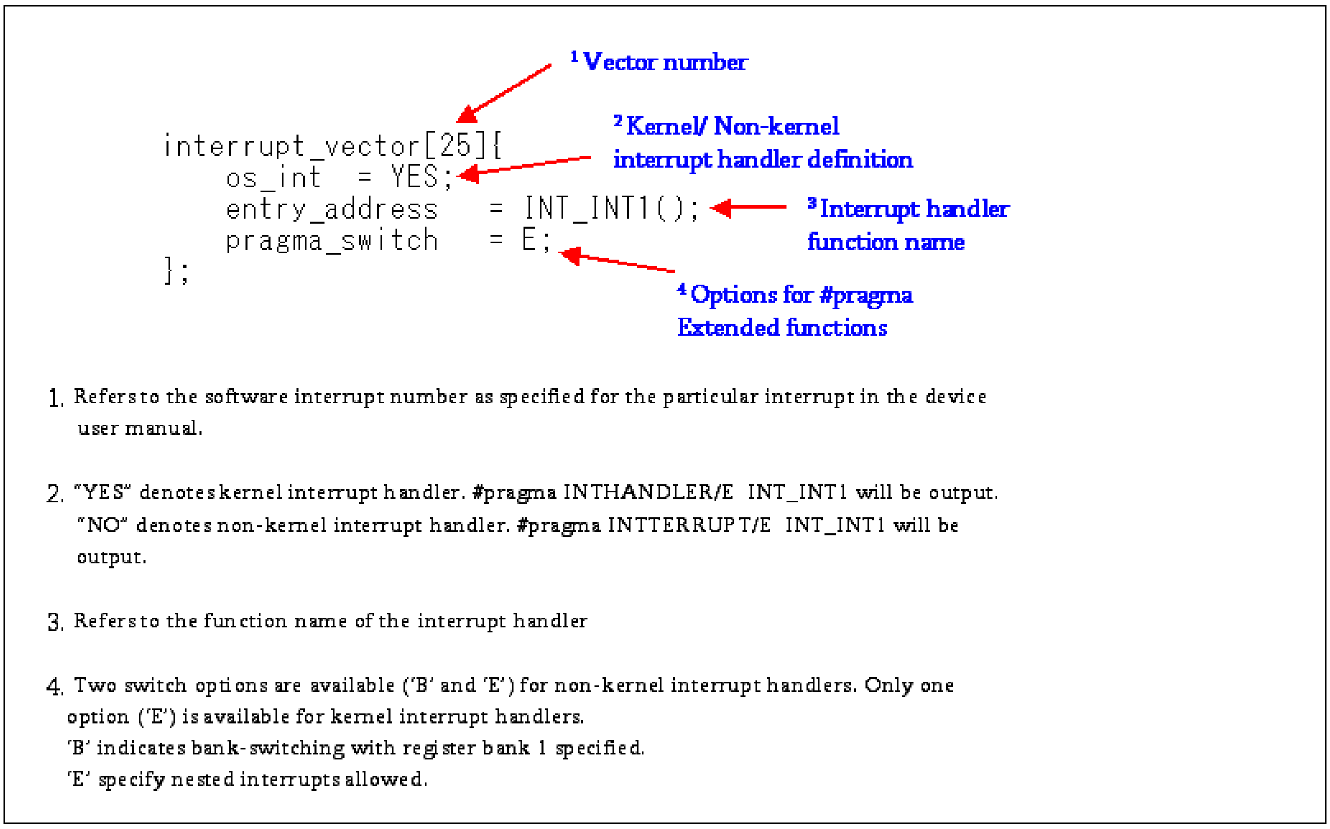
```
// System Definition
system{
    stack_size  = 400;
    priority    = 255;
    system_IPL  = 5;
    tic_nume    = 1;
    tic_deno    = 1;
};
```

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.



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

[1] Vector number

[2] Kernel/ Non-kernel interrupt handler definition

[3] Interrupt handler function name

[4] Options for #pragma Extended functions

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/E INT_INT1 will be output.
   "NO" denotes non-kernel interrupt handler. #pragma INTTERRUPT/E INT_INT1 will be output.

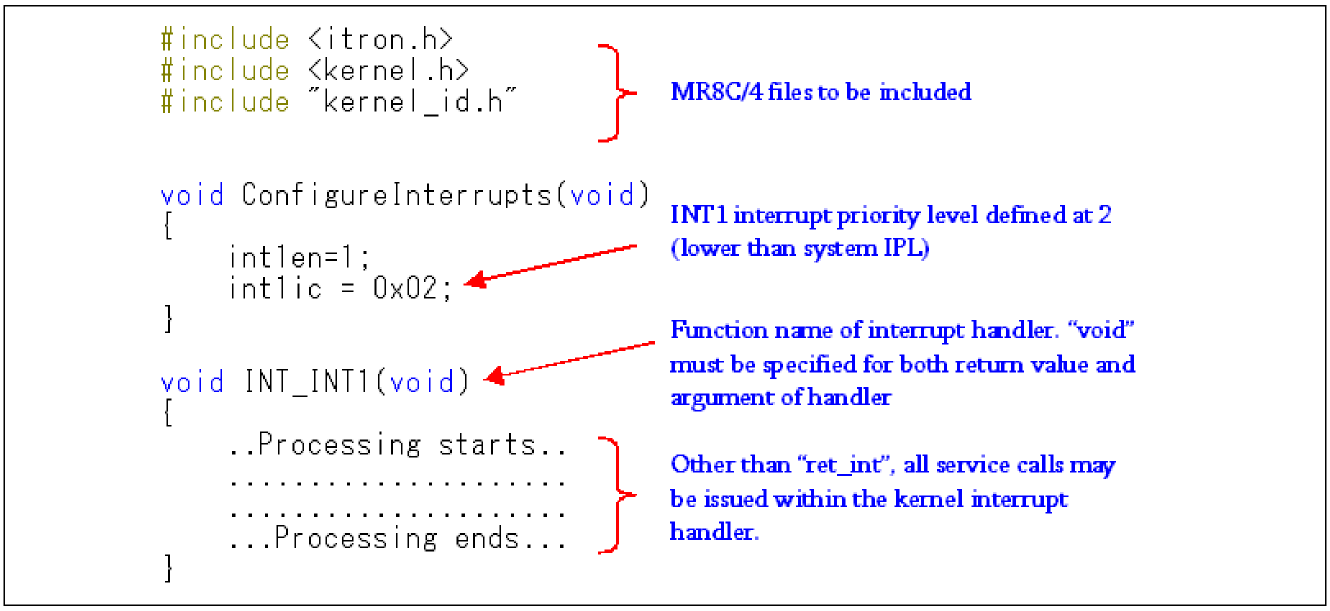
3. Refers to the function name of the interrupt handler

4. Two switch options are available ('B' and 'E') for non-kernel interrupt handlers. Only one option ('E') is available for kernel interrupt handlers.
   'B' indicates bank-switching with register bank 1 specified.
   'E' specify nested interrupts allowed.

Figure 25 Defining Interrupt Vector

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.



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

void ConfigureInterrupts(void)
{
     int1en=1;
     int1ic = 0x02;
}

void INT_INT1(void)
{
     ..Processing starts..
     .....................
     .....................
     ...Processing ends...
}
```

MR8C/4 files to be included

INT1 interrupt priority level defined at 2 (lower than system IPL)

Function name of interrupt handler. "void" must be specified for both return value and argument of handler

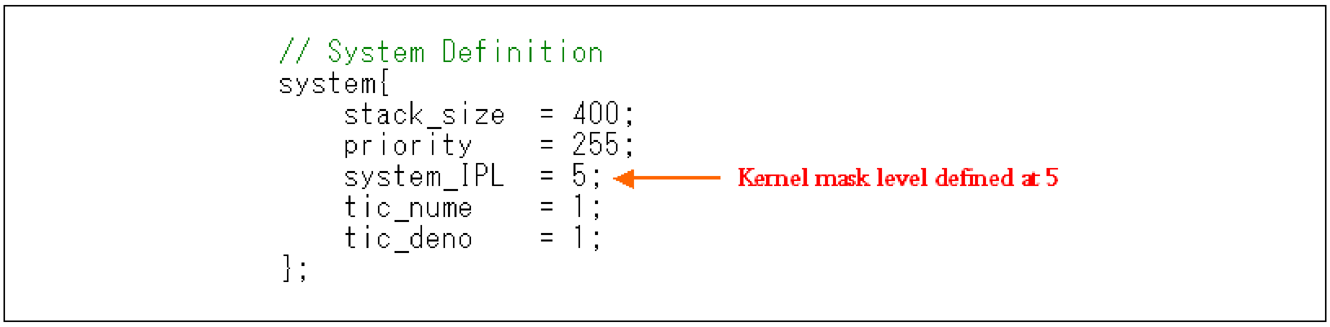
Other than "ret_int", all service calls may be issued within the kernel interrupt handler.

Figure 26 Defining Interrupt Handler

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.

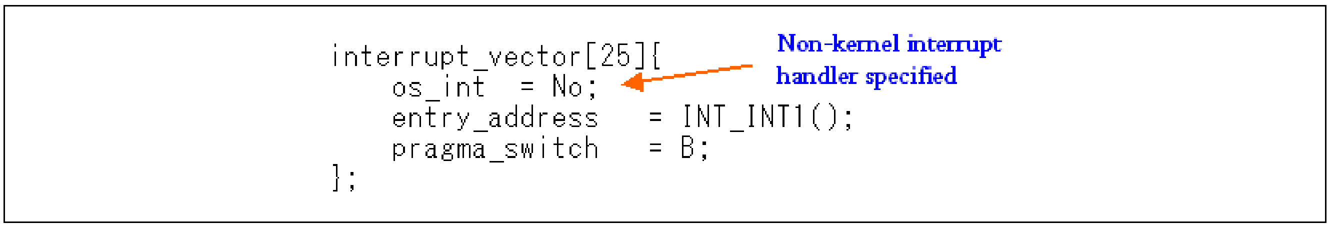


```
// System Definition
system{
    stack_size  = 400;
    priority    = 255;
    system_IPL  = 5;
    tic_nume    = 1;
    tic_deno    = 1;
};
```

Kernel mask level defined at 5

Figure 27 Defining Kernel Mask Level

Step 2: Define interrupt vector



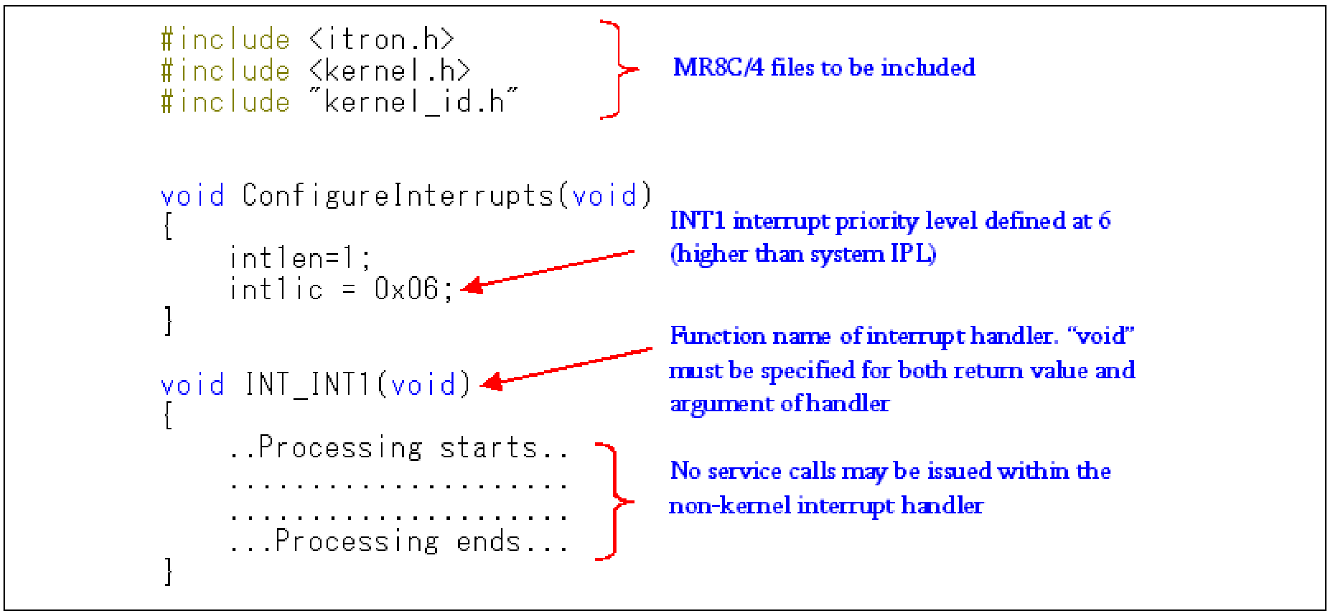
```
interrupt_vector[25]{
    os_int  = No;
    entry_address   = INT_INT1();
    pragma_switch   = B;
};
```

Non-kernel interrupt handler specified

Figure 28 Defining Interrupt Vector

Step 3: Define interrupt handler

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



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


void ConfigureInterrupts(void)
{
    int1en=1;
    int1ic = 0x06;
}

void INT_INT1(void)
{
    ..Processing starts..
    .....................
    .....................
    ...Processing ends...
}
```

MR8C/4 files to be included

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

Function name of interrupt handler. "void" must be specified for both return value and argument of handler

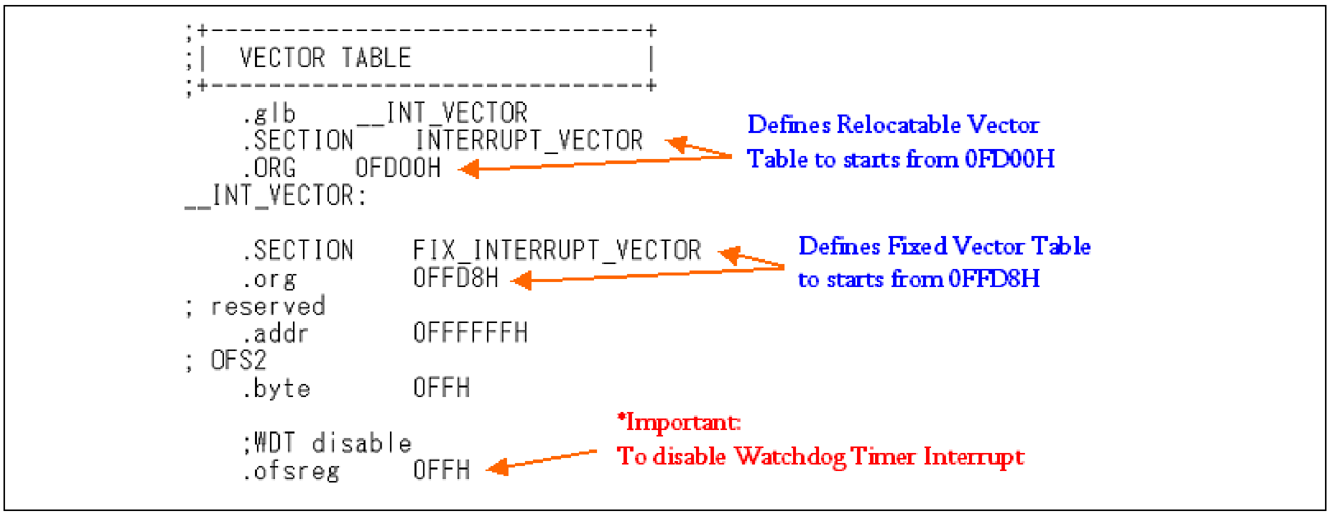
No service calls may be issued within the non-kernel interrupt handler

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         0FFD8H
; reserved
    .addr        0FFFFFFH
; OFS2
    .byte        0FFH

    ;WDT disable
    .ofsreg      0FFH
```

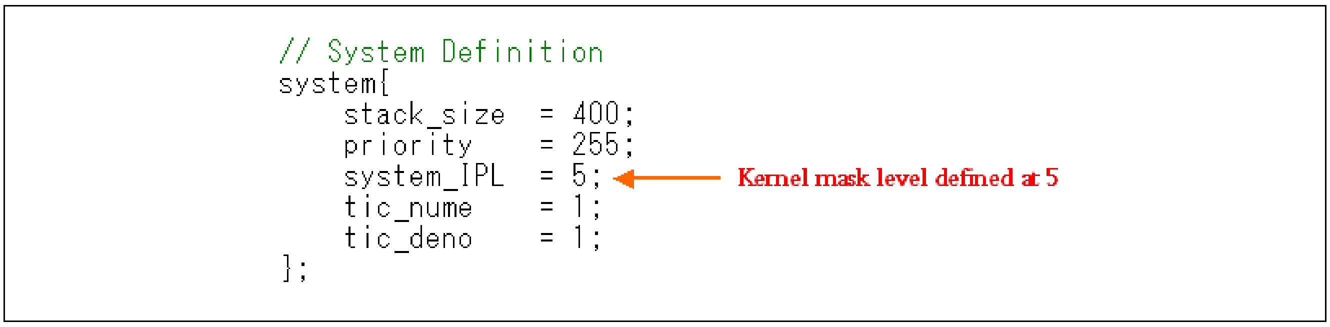
Defines Relocatable Vector Table to starts from 0FD00H
Defines Fixed Vector Table to starts from 0FFD8H
*Important:
To disable Watchdog Timer Interrupt

Figure 30 Defining Vector Tables in “asm_sec.inc”

5.5.2 Writing Kernel Interrupt Handler

Step 1: Define kernel mask level

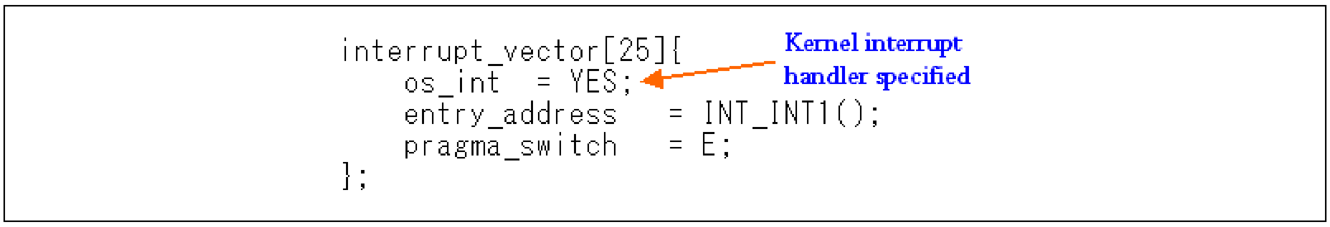
The kernel mask level is defined in configurator file (e.g. template.cfg) under the system definition. This step is identical to step 1 of coding interrupts in C language.



```
// System Definition
system{
    stack_size   = 400;
    priority     = 255;
    system_IPL   = 5;
    tic_nume     = 1;
    tic_deno     = 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          intlen
          MOV.B         #02H,intlic

          .INCLUDE mr8c.inc
          .GLB          INT_INT1

     INT_INT1:

          PUSHM         R0,A0
          irel_wai      #ID_task1
          POPM          R0,A0
          ret_int
```

INT1 interrupt priority level defined at 2 (lower than system IPL)

Global declaration of interrupt handler function name

Save registers to stack

Service calls may be issued within the kernel interrupt handler

Registers are restored

"ret_int" must be used to return from interrupt handler

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     = 255;
    system_IPL   = 5;
    tic_nume     = 1;
    tic_deno     = 1;
};
```

Kernel mask level defined at 5

Figure 34 Defining Kernel Mask Level

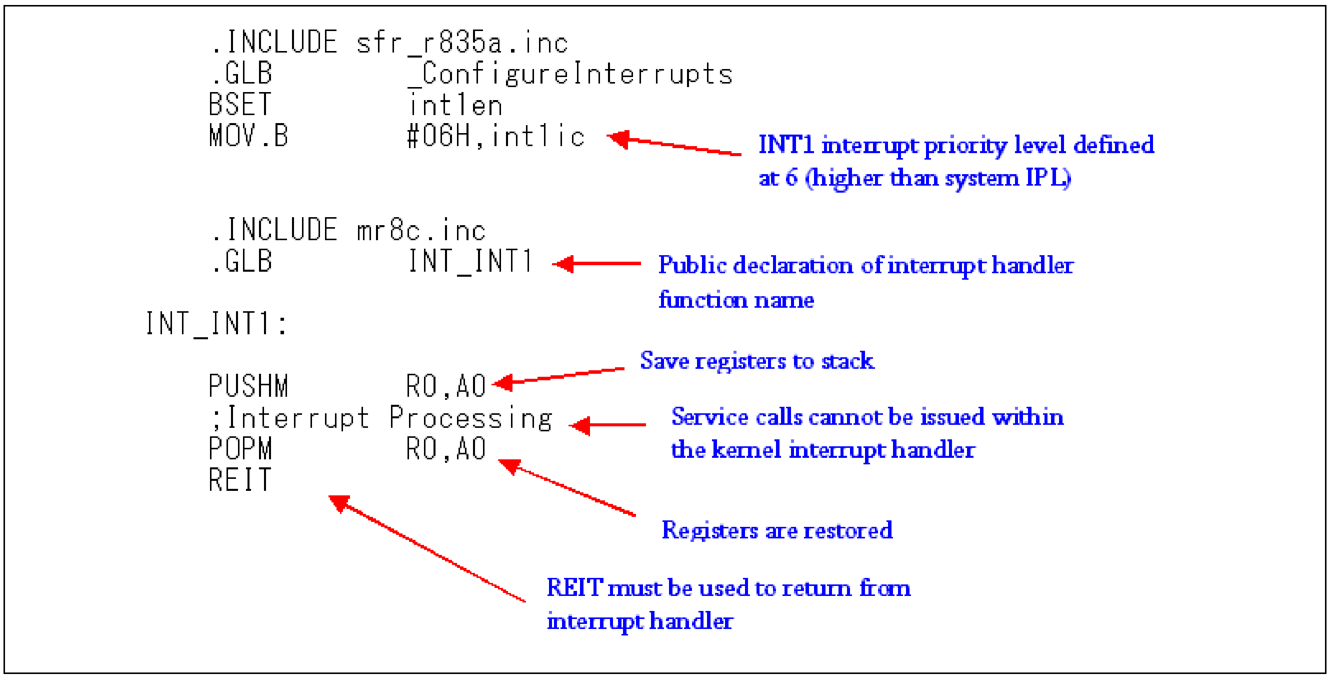
Step 2: Define interrupt vector

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

Non-kernel interrupt handler specified

Figure 35 Defining Interrupt Vector

Step 3: Define interrupt handler



```
        .INCLUDE sfr_r835a.inc
        .GLB         _ConfigureInterrupts
        BSET         int1en
        MOV.B        #06H,int1ic

        .INCLUDE mr8c.inc
        .GLB         INT_INT1

    INT_INT1:

        PUSHM        R0,A0
        ;Interrupt Processing
        POPM         R0,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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	March.01.10	—	First edition issued

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

RENESAS

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

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

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

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

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

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

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan, R.O.C.
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

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

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.0