

- **The Real Time Interrupt**
- Huang Section 6.6
- CRG Block User Guide
  - Exceptions on the 9S12
  - Using interrupts on the 9S12
  - The Real Time Interrupt on the 9S12

### **Using Interrupts on the MC9S12**

What happens when the MC9S12 receives an unmasked interrupt?

- a. Finish current instruction
- b. Push all registers onto the stack
- c. Set I bit of CCR
- d. Load Program Counter from interrupt vector for highest priority interrupt which is pending

The following (from the MC9S12DP256B Device User Guide) shows the exception priorities. The Reset is the highest priority, the Clock Monitor Fail Reset the next highest, etc.

Table 5-1 lists interrupt sources and vectors in default order of priority.

**Table 5-1 Interrupt Vector Locations**

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFFE, \$FFFF	Reset	None	None	–
\$FFFC, \$FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
\$FFFA, \$FFFB	COP failure reset	None	COP rate select	–
\$FFF8, \$FFF9	Unimplemented instruction trap	None	None	–
\$FFF6, \$FFF7	SWI	None	None	–
\$FFF4, \$FFF5	XIRQ	X-Bit	None	–
\$FFF2, \$FFF3	IRQ	I-Bit	IRQCR (IRQEN)	\$F2
\$FFF0, \$FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	\$F0
\$FFEE, \$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	\$EE
\$FFEC, \$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	\$EC
\$FFEA, \$FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	\$EA
\$FFE8, \$FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	\$E8
\$FFE6, \$FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	\$E6
\$FFE4, \$FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	\$E4
\$FFE2, \$FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	\$E2
\$FFE0, \$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	\$E0
\$FFDE, \$FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	\$DE
\$FFDC, \$FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	\$DC
\$FFDA, \$FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	\$DA
\$FFD8, \$FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	\$D8
\$FFD6, \$FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	\$D6
\$FFD4, \$FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	\$D4
\$FFD2, \$FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	\$D2
\$FFD0, \$FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	\$D0
\$FFCE, \$FFCF	Port J	I-Bit	PTJIF (PTJIE)	\$CE
\$FFCC, \$FFCD	Port H	I-Bit	PTHIF (PTHIE)	\$CC
\$FFCA, \$FFCB	Modulus Down Counter underflow	I-Bit	MCCTL (MCZI)	\$CA

\$FFC8, \$FFC9	Pulse Accumulator B Overflow	I-Bit	PBCTL(PBOVI)	\$C8
\$FFC8, \$FFC7	CRG PLL lock	I-Bit	CRGINT(LOCKIE)	\$C8
\$FFC4, \$FFC5	CRG Self Clock Mode	I-Bit	CRGINT (SCMIE)	\$C4
\$FFC2, \$FFC3	BDLC	I-Bit	DLCBCR1(IE)	\$C2
\$FFC0, \$FFC1	IIC Bus	I-Bit	IBCR (IBIE)	\$C0
\$FFBE, \$FFBF	SPI1	I-Bit	SP1CR1 (SPIE, SPTIE)	\$BE
\$FFBC, \$FFBD	SPI2	I-Bit	SP2CR1 (SPIE, SPTIE)	\$BC
\$FFBA, \$FFBB	EEPROM	I-Bit	EECTL(CCIE, CBEIE)	\$BA
\$FFB8, \$FFB9	FLASH	I-Bit	FCTL(CCIE, CBEIE)	\$B8
\$FFB6, \$FFB7	CAN0 wake-up	I-Bit	CAN0RIER (WUPIE)	\$B6
\$FFB4, \$FFB5	CAN0 errors	I-Bit	CAN0RIER (CSCIE, OVRIE)	\$B4
\$FFB2, \$FFB3	CAN0 receive	I-Bit	CAN0RIER (RXFIE)	\$B2
\$FFB0, \$FFB1	CAN0 transmit	I-Bit	CAN0TIER (TXEIE2-TXEIE0)	\$B0
\$FFAE, \$FFAF	CAN1 wake-up	I-Bit	CAN1RIER (WUPIE)	\$AE
\$FFAC, \$FFAD	CAN1 errors	I-Bit	CAN1RIER (CSCIE, OVRIE)	\$AC
\$FFAA, \$FFAB	CAN1 receive	I-Bit	CAN1RIER (RXFIE)	\$AA
\$FFA8, \$FFA9	CAN1 transmit	I-Bit	CAN1TIER (TXEIE2-TXEIE0)	\$A8
\$FFA6, \$FFA7	CAN2 wake-up	I-Bit	CAN2RIER (WUPIE)	\$A6
\$FFA4, \$FFA5	CAN2 errors	I-Bit	CAN2RIER (CSCIE, OVRIE)	\$A4
\$FFA2, \$FFA3	CAN2 receive	I-Bit	CAN2RIER (RXFIE)	\$A2
\$FFA0, \$FFA1	CAN2 transmit	I-Bit	CAN2TIER (TXEIE2-TXEIE0)	\$A0
\$FF9E, \$FF9F	CAN3 wake-up	I-Bit	CAN3RIER (WUPIE)	\$9E
\$FF9C, \$FF9D	CAN3 errors	I-Bit	CAN3RIER (TXEIE2-TXEIE0)	\$9C
\$FF9A, \$FF9B	CAN3 receive	I-Bit	CAN3RIER (RXFIE)	\$9A
\$FF98, \$FF99	CAN3 transmit	I-Bit	CAN3TIER (TXEIE2-TXEIE0)	\$98
\$FF96, \$FF97	CAN4 wake-up	I-Bit	CAN4RIER (WUPIE)	\$96
\$FF94, \$FF95	CAN4 errors	I-Bit	CAN4RIER (CSCIE, OVRIE)	\$94
\$FF92, \$FF93	CAN4 receive	I-Bit	CAN4RIER (RXFIE)	\$92
\$FF90, \$FF91	CAN4 transmit	I-Bit	CAN4TIER (TXEIE2-TXEIE0)	\$90
\$FF8E, \$FF8F	Port P Interrupt	I-Bit	PTPIF (PTPIE)	\$8E
\$FF8C, \$FF8D	PWM Emergency Shutdown	I-Bit	PWMSDN (PWMIE)	\$8C
\$FF80 to \$FF8B	Reserved			

Most interrupts have both a specific mask and a general mask. For most interrupts the general mask is the I bit of the CCR. For the TOF interrupt the specific mask is the TOI bit of the TSCR2 register.

Before using interrupts, make sure to:

a. Load stack pointer

- Done for you in C by the C startup code

b. Write Interrupt Service Routine

- Do whatever needs to be done to service interrupt
  - You cannot pass a variable to an ISR. If the ISR needs to know the value of a variable used in another part of the program, that variable must be **global**
  - You cannot return a variable from an ISR to another part of the program. If the program needs to know the value of a variable set in an ISR, that variable must be **global**

- Clear interrupt flag

- Exit with the RTI instruction
  - Use the interrupt key word in the CodeWarrior compiler
  - Tells compiler to exit function with rti instruction rather than rts instruction

c. Load address of interrupt service routine into interrupt vector

- E.g., `UserTimerOvf = (unsigned short) &toi_isr;`

d. Do any setup needed for interrupt

- For example, for the TOF interrupt, turn on timer and set prescaler

e. Enable specific interrupt

f. Enable interrupts in general (clear I bit of CCR with cli instruction)

- Can disable all (maskable) interrupts with the sei instruction.

- When the MC9S12 is reset, the interrupts are disabled. Some compilers enable interrupts by default, so your code should turn off interrupts before doing setup.

- Can do this with `__asm(sei);`

- Can also do the following:

```
#define disable() __asm(sei)
```

```
#define enable() __asm(cli)
```

and then use more C-like `disable();` and `enable();`

### Example of C program using Timer Overflow Interrupt

```
#include <hidef.h>           /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */
#include "vectors12.h"       /* Dbug12 RAM-based interrupt vectors */
#define enable() __asm(cli)
#define disable() __asm(sei)
interrupt void toi_isr(void); /* Function prototype */

void main(void)
{
    disable();
    DDRB = 0xff;           /* Make Port B output */

    /* Setup for Timer Overflow Interrupt */
    TSCR1 = 0x80;         /* Turn on timer */
    TSCR2 = 0x06;         /* Set prescaler so interrupt period is 175 ms */
    UserTimerOvf = (unsigned short) &toi_isr;
    TSCR2 = TSCR2 | 0x80; /* Enable timer overflow interrupt */
    /* Done with setup */

    enable(); /* Enable interrupts (clear I bit) */
    while (1)
    {
        __asm(wai); /* Do nothing - go into low power mode */
    }
}

interrupt void toi_isr(void)
{
    PORTB = PORTB+1;
    TFLG2 = 0x80;         /* Clear timer interrupt flag */
}
```

**An example of the MC9S12 registers and stack when a TOF interrupt is received**

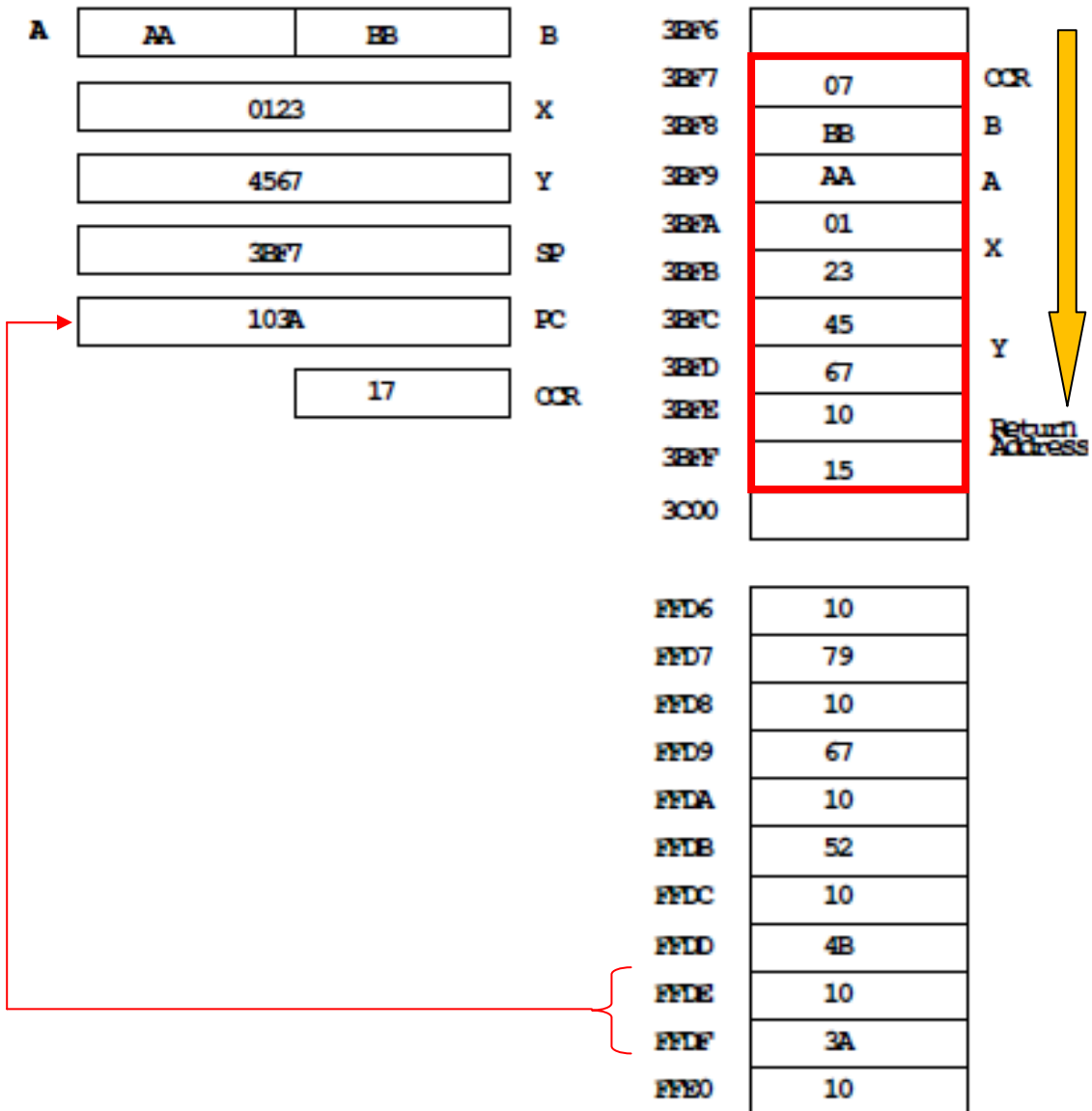
**HC12 STATE BEFORE RECEIVING TOF INTERRUPT**

<b>A</b>	<b>AA</b>	<b>BB</b>	<b>B</b>	<b>3BF6</b>	
	0123		<b>X</b>	<b>3BF7</b>	
	4567		<b>Y</b>	<b>3BF8</b>	
	3C00		<b>SP</b>	<b>3BF9</b>	
	1015		<b>PC</b>	<b>3BFA</b>	
				<b>3BFB</b>	
			<b>OCR</b>	<b>3BFC</b>	
				<b>3BFD</b>	
				<b>3BFE</b>	
				<b>3BF7</b>	
				<b>3C00</b>	
				<b>FFD6</b>	10
				<b>FFD7</b>	79
				<b>FFD8</b>	10
				<b>FFD9</b>	67
				<b>FFDA</b>	10
				<b>FFDB</b>	52
				<b>FFDC</b>	10
				<b>FFDD</b>	4B
				<b>FFDE</b>	10
				<b>FFDF</b>	3A
				<b>FFE0</b>	10

**An example of the MC9S12 registers and stack just after a TOF interrupt is received**

- All of the MC9S12 registers are pushed onto the stack, the PC is loaded with the contents of the Interrupt Vector, and the I bit of the CCR is set

**HC12 STATE AFTER RECEIVING TOF INTERRUPT**





### **Interrupt vectors for the HCS12**

- The interrupt vectors for the MC9S12DG256 are located in memory from **0xFF80 to 0xFFFF**.
- These vectors are programmed into Flash EEPROM and are very difficult to change
- DBug12 redirects the interrupts to a region of RAM where they are easy to change
- For example, when the HCS12 gets a TOF interrupt:
  - It loads the PC with the contents of **0xFFDE and 0xFFDF**.
  - The program at that address tells the HCS12 to look at address **0x3E5E and 0x3E5F**.
  - If there is a 0x0000 at these two addresses, DBug12 gives an error stating that the interrupt vector is uninitialized.
  - If there is anything else at these two addresses, DBug12 loads this data into the PC and executes the routine located there.
  - To use the TOF interrupt you need to put the address of your TOF ISR at addresses **0x3E5E and 0x3E5F**.
- The location of the vectors is defined in include files so you don't have to remember them or look them up in the reference manual.
  - For Assembly programs, the vectors are defined in the file hcs12.inc

```
UserTimerOvf equ $3E5E
```
  - For C programs, the vectors are defined in the file vectors12.h

```
#define UserTimerOvf _VEC16(47) /* Maps to 0x3E5E */
```

**Commonly Used Interrupt Vectors for the MC9S12Dp256**

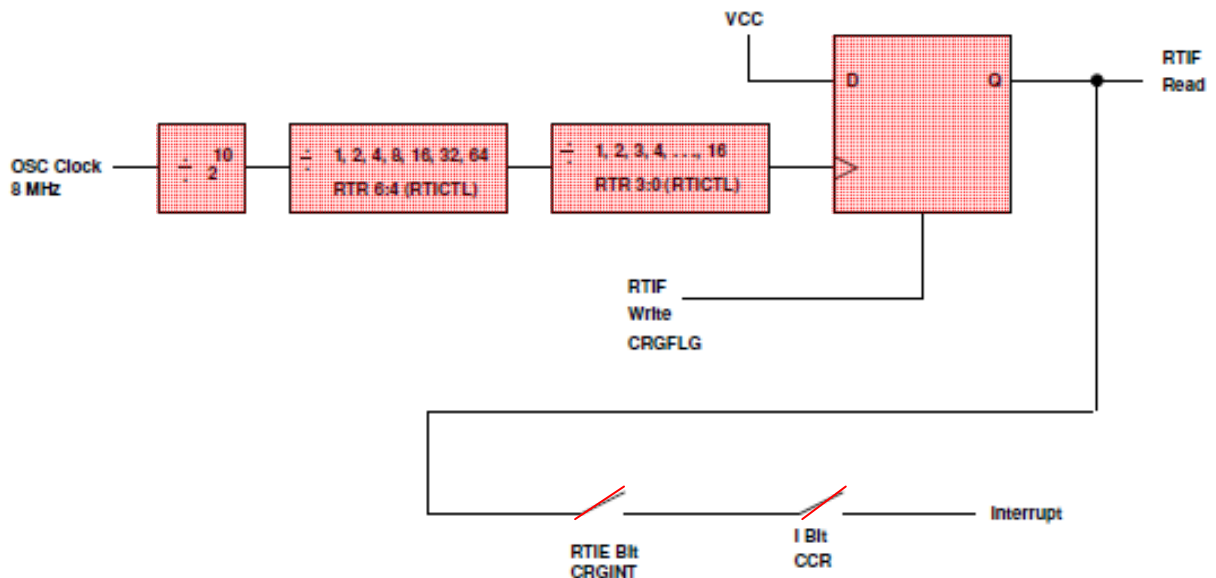
Interrupt	Specific Mask	General Mask	Normal Vector	DBug-12 Vector
SPI2	SP2CR1 (SPIE, SPTIE)	I	FFBC, FFBD	3E3C, 3E3D
SPI1	SP1CR1 (SPIE, SPTIE)	I	FFBE, FFBF	3E3E, 3E3F
IIC	IBCR (IBIR)	I	FFC0, FFC1	3E40, 3E41
BDLC	DLCBCR (IE)	I	FFC2, FFC3	3E42, 3E43
CRG Self Clock Mode	CRGINT (SCMIE)	I	FFC4, FFC5	3E44, 3E45
CRG Lock	CRGINT (LOCKIE)	I	FFC6, FFC7	3E46, 3E47
Pulse Acc B Overflow	PBCTL (PBOVI)	I	FFC8, FFC9	3E48, 3E49
Mod Down Ctr UnderFlow	MCCTL (MCZI)	I	FFCA, FFCB	3E4A, 3E4B
Port H	PTHIF (PTHIE)	I	FFCC, FFCD	3E4C, 3E4D
Port J	PTJIF (PTJIE)	I	FFCE, FFCF	3E4E, 3E4F
ATD1	ATD1CTL2 (ASCIE)	I	FFD0, FFD1	3E50, 3E51
ATD0	ATD0CTL2 (ASCIE)	I	FFD2, FFD3	3E52, 3E53
SCI1	SC1CR2 (TIE, TCIE, RIE, ILIE)	I	FFD4, FFD5	3E54, 3E55
SCIO	SC0CR2 (TIE, TCIE, RIE, ILIE)	I	FFD6, FFD7	3E56, 3E57
SPIO	SPOCR1 (SPIE)	I	FFD8, FFD9	3E58, 3E59
Pulse Acc A Edge	PACTL (PAI)	I	FFDA, FFDB	3E5A, 3E5B
Pulse Acc A Overflow	PACTL (PAOVI)	I	FFDC, FFDD	3E5C, 3E5D
Enh Capt Timer Overflow	TSCR2 (TOI)	I	FFDE, FFDF	3E5E, 3E5F
Enh Capt Timer Channel 7	TIE (C7I)	I	FFE0, FFE1	3E60, 3E61
Enh Capt Timer Channel 6	TIE (C6I)	I	FFE2, FFE3	3E62, 3E63
Enh Capt Timer Channel 5	TIE (C5I)	I	FFE4, FFE5	3E64, 3E65
Enh Capt Timer Channel 4	TIE (C4I)	I	FFE6, FFE7	3E66, 3E67
Enh Capt Timer Channel 3	TIE (C3I)	I	FFE8, FFE9	3E68, 3E69
Enh Capt Timer Channel 2	TIE (C2I)	I	FFEA, FFEB	3E6A, 3E6B
Enh Capt Timer Channel 1	TIE (C1I)	I	FFEC, FFED	3E6C, 3E6D
Enh Capt Timer Channel 0	TIE (C0I)	I	FFEE, FFEF	3E6E, 3E6F
Real Time	CRGINT (RTIE)	I	FFF0, FFF1	3E70, 3E71
IRQ	IRQCR (IRQEN)	I	FFF2, FFF3	3E72, 3E73
XIRQ	(None)	X	FFFF, FFFF	3E74, 3E75
SWI	(None)	(None)	FFF6, FFF7	3E76, 3E77
Unimplemented Instruction	(None)	(None)	FFF8, FFF9	3E78, 3E79
COP Failure	COPCTL (CR2-CR0 COP Rate Select)	(None)	FFFA, FFFB	3E7A, 3E7B
COP Clock Moniotr Fail	PLLCTL (CME, SCME)	(None)	FFFC, FFFD	3E7C, 3E7D
Reset	(None)	(None)	FFFE, FFFF	3E7E, 3E7F

### **Exceptions on the MC9S12**

- Exceptions are the way a processor responds to things other than the normal sequence of instructions in memory.
- Exceptions consist of such things as Reset and Interrupts.
- Interrupts allow a processor to respond to an event without constantly polling to see whether the event has occurred.
- On the MC9S12 some interrupts cannot be masked — these are the Unimplemented Instruction Trap and the Software Interrupt (SWI instruction).
- XIRQ interrupt is masked with the X bit of the Condition Code Register. Once the X bit is cleared to enable the XIRQ interrupt, it cannot be set to disable it.
  - The XIRQ interrupt is for external events such as power fail which must be responded to.
- The rest of the MC9S12 interrupts are masked with the I bit of the CCR.
  - All these other interrupts are also masked with a specific interrupt mask. For example, the Timer Overflow Interrupt is masked with the TOI bit of the TSCR2 register.
  - This allows you to enable any of these other interrupts you want.
  - The I bit can be set to 1 to disable all of these interrupts if needed.

### The Real Time Interrupt

- Like the Timer Overflow Interrupt, the Real Time Interrupt allows you to interrupt the processor at a regular interval.
- Information on the Real Time Interrupt is in the CRG Block User Guide
- There are two clock sources for MC9S12 hardware.
  - Some hardware uses the Oscillator Clock. The RTI system uses this clock.
    - \* For our MC9S12, the oscillator clock is 8 MHz.
  - Some hardware uses the Bus Clock. The Timer system (including the Timer Overflow Interrupt) use this clock.
    - \* For our MC9S12, the bus clock is 24 MHz.



- The specific **interrupt mask for the Real Time Interrupt** is the **RTIE bit** of the **CRGINT** register.
- When the **Real Time Interrupt occurs**, the **RTIF bit** of the **CRGFLG** register is set.
  - To clear the Real Time Interrupt write a 1 to the RTIF bit of the CRGFLG register.
- The interrupt rate is set by the **RTR 6:4** and **RTR 2:0** bits of the **RTICTL** register. The RTR 6:4 bits are the Prescale Rate Select bits for the RTI, and the RTR 2:0 bits are the Modulus Counter Select bits to provide additional granularity.

<b>RTIF</b>	<b>PORF</b>	<b>0</b>	<b>LOCKIF</b>	<b>LOCK</b>	<b>TRACK</b>	<b>SMIF</b>	<b>SM</b>	<b>0x0037</b>	<b>CRGFLG</b>
-------------	-------------	----------	---------------	-------------	--------------	-------------	-----------	---------------	---------------

<b>RTIE</b>	<b>0</b>	<b>0</b>	<b>LOCKIE</b>	<b>0</b>	<b>0</b>	<b>SMIE</b>	<b>0</b>	<b>0x0038</b>	<b>CRGINT</b>
-------------	----------	----------	---------------	----------	----------	-------------	----------	---------------	---------------

<b>0</b>	<b>RTR6</b>	<b>RTR5</b>	<b>RTR4</b>	<b>RTR3</b>	<b>RTR2</b>	<b>RTR1</b>	<b>RTR0</b>	<b>0x003B</b>	<b>RTICTL</b>
----------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	---------------	---------------

- To use the Real Time Interrupt, set the rate by writing to the RTR 6:4 and the RTR 3:0 bits of the RTICTL, and enable the interrupt by setting the RTIE bit of the CRGINT register
  - In the Real Time Interrupt ISR, you need to clear the RTIF flag by writing a 1 to the RTIF bit of the CRGFLG register.

- The following table shows all possible values, in ms, selectable by the RTICTL register (assuming the system uses a 8 MHz oscillator):

RTR 3:0	RTR 6:4							
	000 (0)	001 (1)	010 (2)	011 (3)	100 (4)	101 (5)	110 (6)	111 (7)
0000 (0)	Off	0.128	0.256	0.512	1.024	2.048	4.096	8.192
0001 (1)	Off	0.256	0.512	1.024	2.048	4.096	8.192	16.384
0010 (2)	Off	0.384	0.768	1.536	3.072	6.144	12.288	24.576
0011 (3)	Off	0.512	1.024	2.048	4.096	8.192	16.384	32.768
0100 (4)	Off	0.640	1.280	2.560	5.120	10.240	20.480	40.960
0101 (5)	Off	0.768	1.536	3.072	6.144	12.288	24.570	49.152
0110 (6)	Off	0.896	1.792	3.584	7.168	14.336	28.672	57.344
0111 (7)	Off	1.024	2.048	4.096	8.192	16.384	32.768	65.536
1000 (8)	Off	1.152	2.304	4.608	9.216	18.432	36.864	73.728
1001 (9)	Off	1.280	2.560	5.120	10.240	20.480	40.960	81.920
1010 (A)	Off	1.408	2.816	5.632	11.264	22.528	45.056	90.112
1011 (B)	Off	1.536	3.072	6.144	12.288	24.576	49.152	98.304
1100 (C)	Off	1.664	3.328	6.656	13.312	26.624	53.248	106.496
1101 (D)	Off	1.729	3.584	7.168	14.336	28.672	57.344	114.688
1110 (E)	Off	1.920	3.840	7.680	15.360	30.720	61.440	122.880
1111 (F)	Off	2.048	4.096	8.192	16.384	32.768	65.536	131.072

- Here is a C program which uses the Real Time Interrupt:

```
#include <hdef.h>           /* common defines and macros */
#include "derivative.h"     /* derivative-specific definitions */
#include "vectors12.h"     /* Dbug12 RAM-based interrupt vectors */
#define enable() __asm(cli)
#define disable() __asm(sei)
interrupt void rti_isr(void);

void main(void)
{
    disable();
    DDRB = 0xff;
    PORTB = 0;
    RTICTL = 0x63;        /* Set rate to 16.384 ms */
    CRGINT = 0x80;       /* Enable RTI interrupts */
    CRGFLG = 0x80;       /* Clear RTI Flag */
    UserRTI = (unsigned short) &rti_isr;
    enable();
    while (1)
    {
        __asm(wai);     /* Do nothing -- wait for interrupt */
    }
}

interrupt void rti_isr(void)
{
    PORTB = PORTB + 1;
    CRGFLG = 0x80;
}
```

RTI interrupt service routine to display a global 16-bit variable called value on the seven-segment display

```
interrupt void rti_isr(void)
{
    static unsigned char digit=0;
    const char hex2seven_seg[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
                                   0x7D, 0x07, 0x7F, 0x6F, 0x77, 0x7c,
                                   0x58, 0x5e, 0x79, 0x71};

    switch (digit) {
        case 0: PTP = 0x0E;
                PTJ |= 0x02;
                PORTB = hex2seven_seg[(value>>12)&0x0F];
                break;
        case 1: PTP = 0x0D;
                PTJ |= 0x02;
                PORTB = hex2seven_seg[(value>>8)&0x0F];
                break;
        case 2: PTP = 0x0B;
                PTJ |= 0x02;
                PORTB = hex2seven_seg[(value>>4)&0x0F];
                break;
        case 3: PTP = 0x07;
                PTJ |= 0x02;
                PORTB = hex2seven_seg[(value)&0x0F];
                break;
    }
    if (++digit >= 4) digit = 0;
    CRGFLG = 0x80;
}
```

- digit is declared to be static so its value remains between entries into RTI\_isr
- You cannot pass a value to an interrupt service routine, so any variable from another part of the program used by the ISR must be declared as global
- You cannot pass a value out of an ISR, so if another part of the program needs a value determined inside an ISR, you must use a global variable. It must also be



declared as volatile so the compiler knows that its value may change outside the regular program flow.