CMPE1250 – Real-time Interrupt (sans interrupt)

Your micro uses a crystal oscillator to generate an internal clock. The internal clocks drive and coordinate all activity on the micro. The **16MHz** crystal used on the board is divided by two to provide the Bus Clock at **8MHz**. We learned that with PLL, the system clock may be configured to run at other rates, all derived from the fundamental crystal (**OSCCLK**) rate.

The 9S12XDP microcontroller contains a real-time interrupt module that uses **OSCCLK** as its clock. This means that the RTI hardware runs independent of the bus clock. What advantages would this have?

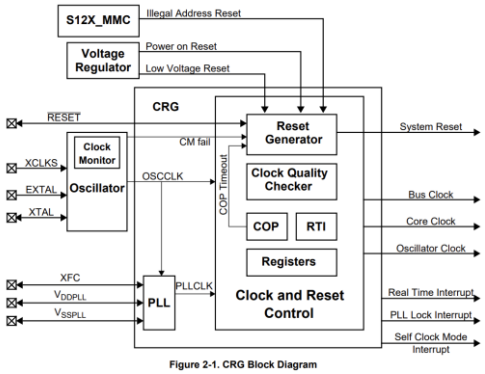

Figure 2-1. CRG Block Diagram

In addition to being independent from the bus clock, the RTI may also be configured to run when the device is pseudo stop mode, but this is a topic for future you.

The RTI module is principally configured with the RTICTL register:

2.3.2.8 CRG RTI Control Register (RTICTL)

This register selects the timeout period for the real time interrupt.

	7	6	5	4	3	2	1	0
R W	RTDEC	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
Reset	0	0	0	0	0	0	0	0

Figure 2-11. CRG RTI Control Register (RTICTL)

The module uses a prescale rate to divide the **OSCCLK** clock down to permit longer timing events. This is configured with the **RTR[6:4]** bits, and is additionally clocked down by a modulus counter through the **RTR[3:0]** bits.

The MSB (**RTDEC**) determines if the main prescaler value is interpreted as decimal or binary, and because the prescale value is somewhat arbitrary for each combination of bits, tables may be used to select the prescale value and modulus values.

For example, if the decimal table is used (**RTDEC = 1**), the following table would determine meaning of the **RTR** bits:

Table 2-8. RTI Frequency Divide Rates for RTDEC = 1

RTR[3:0]	RTR[6:4] =							
	000 ($1x10^3$)	001 ($2x10^3$)	010 ($5x10^3$)	011 ($10x10^3$)	100 ($20x10^3$)	101 ($50x10^3$)	110 ($100x10^3$)	111 ($200x10^3$)
0000 (÷1)	$1x10^3$	$2x10^3$	$5x10^3$	$10x10^3$	$20x10^3$	$50x10^3$	$100x10^3$	$200x10^3$
0001 (÷2)	$2x10^3$	$4x10^3$	$10x10^3$	$20x10^3$	$40x10^3$	$100x10^3$	$200x10^3$	$400x10^3$
0010 (÷3)	$3x10^3$	$6x10^3$	$15x10^3$	$30x10^3$	$60x10^3$	$150x10^3$	$300x10^3$	$600x10^3$
0011 (÷4)	$4x10^3$	$8x10^3$	$20x10^3$	$40x10^3$	$80x10^3$	$200x10^3$	$400x10^3$	$800x10^3$
0100 (÷5)	$5x10^3$	$10x10^3$	$25x10^3$	$50x10^3$	$100x10^3$	$250x10^3$	$500x10^3$	$1x10^6$
0101 (÷6)	$6x10^3$	$12x10^3$	$30x10^3$	$60x10^3$	$120x10^3$	$300x10^3$	$600x10^3$	$1.2x10^6$
0110 (÷7)	$7x10^3$	$14x10^3$	$35x10^3$	$70x10^3$	$140x10^3$	$350x10^3$	$700x10^3$	$1.4x10^6$
0111 (÷8)	$8x10^3$	$16x10^3$	$40x10^3$	$80x10^3$	$160x10^3$	$400x10^3$	$800x10^3$	$1.6x10^6$
1000 (÷9)	$9x10^3$	$18x10^3$	$45x10^3$	$90x10^3$	$180x10^3$	$450x10^3$	$900x10^3$	$1.8x10^6$
1001 (÷10)	$10\ x10^3$	$20x10^3$	$50x10^3$	$100x10^3$	$200x10^3$	$500x10^3$	$1x10^6$	$2x10^6$
1010 (÷11)	$11\ x10^3$	$22x10^3$	$55x10^3$	$110x10^3$	$220x10^3$	$550x10^3$	$1.1x10^6$	$2.2x10^6$
1011 (÷12)	$12x10^3$	$24x10^3$	$60x10^3$	$120x10^3$	$240x10^3$	$600x10^3$	$1.2x10^6$	$2.4x10^6$
1100 (÷13)	$13x10^3$	$26x10^3$	$65x10^3$	$130x10^3$	$260x10^3$	$650x10^3$	$1.3x10^6$	$2.6x10^6$
1101 (÷14)	$14x10^3$	$28x10^3$	$70x10^3$	$140x10^3$	$280x10^3$	$700x10^3$	$1.4x10^6$	$2.8x10^6$
1110 (÷15)	$15x10^3$	$30x10^3$	$75x10^3$	$150x10^3$	$300x10^3$	$750x10^3$	$1.5x10^6$	$3x10^6$
1111 (÷16)	$16x10^3$	$32x10^3$	$80x10^3$	$160x10^3$	$320x10^3$	$800x10^3$	$1.6x10^6$	$3.2x10^6$

As an example, if we used a value of `0b10010111` for `RTICTL`, the first bit would mean use the decimal table (the one above), with `001` for the prescale (`2 x 10`^{3} `(2000)`), and `0111` for the modulus counter (divide by `8`).

This would provide a total division of `2000 x 8`, or `16000`. The `OSCCLK` clock is `16MHz`, so `16MHz / 16000` = `1000Hz`, or `1ms`.

Once the `RTICTL` register is set to anything other than zero for both the prescale and modulus counter, the RTI is running, like it or not.

With a 16MHz crystal, the RTI can be configured to run with intervals as long as `200ms` (`16MHz / 3.2 x 10`^{6}).

The timeout event for the RTI is determined with the `RTIF` bit of the `CRGFLG` register – if this bit is found to be set, the RTI period is complete, and the flag needs to be reset.

2.3.2.4 CRG Flags Register (CRGFLG)

This register provides CRG status bits and flags.

	7	6	5	4	3	2	1	0
R	RTIF	PORF	LVRF	LOCKIF	LOCK	TRACK	SCMIF	SCM
W								
Reset	0	1	2	0	0	0	0	0

1. PORF is set to 1 when a power on reset occurs. Unaffected by system reset.
2. LVRF is set to 1 when a low-voltage reset occurs. Unaffected by system reset.

= Unimplemented or Reserved

Figure 2-7. CRG Flags Register (CRGFLG)

Field	Description
7 RTIF	**Real Time Interrupt Flag** — RTIF is set to 1 at the end of the RTI period. This flag can only be cleared by writing a 1. Writing a 0 has no effect. If enabled (RTIE = 1), RTIF causes an interrupt request. 0 RTI time-out has not yet occurred. 1 RTI time-out has occurred.

As discussed in the 'Mighty Derivative File' document, this flag is subject to a special clearing procedure (you write a `1` to clear the flag, writing a `0` has no effect on the bits in this register).

Testing the RTI:

To prove that this works as expected, create a program with PLL and switch/LED support and include the following code:

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  PLL_To20MHz();
  SWL_Init();

  RTICTL = 0b10010111;

  for (;;)
  {
    if (CRGFLG_RTIF) // RTI period over?
    {
      CRGFLG = CRGFLG_RTIF_MASK;    //clear flag, VERY IMPORTANT
      SWL_TOG (SWL_RED);
    }
  }
}
```

Discussion:

What are the characteristics of the waveform that this program produces on Pin 82?

Does enabling or disabling the PLL code have any effect on the output waveform?

How would you document the `RTICTL` configuration line of code?

Creating an RTI library

You should create a new library to support the RTI.

The following header would be a good starting point, and you can provide the appropriate implementation:

```
////////////////////////////////////////////////////////////////////////
// Processor:     MC9S12XDP512
// Bus Speed:     Does not matter, uses 16MHz OSCCLK
// Author:        Simon Walker
// Details:       Utility methods that use the RTI module
// Date:          Feb 27/2024
// Revision History :
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Library Prototypes
////////////////////////////////////////////////////////////////////////

// perform a blocking delay for n x 1ms intervals
void RTI_Delay (unsigned long Intervalms);
```

The **`RTI_Delay`** method will be a blocking function, meaning it does not return (blocks) until it has completed the delay.

In the future you will use more sophisticated techniques to manage delays and timing such that you don't need to wait around for a blocking method to complete. Blocking delays do have some niche utility, particularly in initialization sequences where specific timing delays are required, and you don't *want* anything else to be happening.

The implementation of the **`RTI_Delay`** function should perform the following steps:

- Stop the RTI (it shouldn't be running).
- Check the **`RTIF`** flag of **`CRGFLG`** register, if it is set, clear it.
- Start the RTI with a **`1ms`** period.
- For `Intervalms` times, wait for the **`RTIF`** flag, then clear it.
- Stop the RTI (you are now done with it, no need to keep it running).

Your function should be tolerant of **`0ms`** delays (immediately returns, with RTI off).

Because the parameter to the function is an **`unsigned long`**, the range of delay is **`0ms`** to just under 50 days, with a step size of 1ms.

NOTE: Blocking delays don't need to be perfect* – just calling the method adds overhead to the timing. For accurate timing, we will use other techniques.

Test your new library with various values, ensuring that it works as expected. Proceed to the ICA when you are happy with your (functional) library.

*You will find that for 1ms intervals, the delay is still *very* accurate.