Freescale Semiconductor
Users Guide

RS08QRUG
Rev. 2, 11/2011

RS08 Peripheral Module Quick Reference

A Compilation of Demonstration Software for RS08 Modules

This collection of code examples, useful tips, and quick reference material has been created to help users with fast development of their applications. Each section within this document contains an example that may be modified to work with RS08 MCU Family members. When you are developing your application, consult your device data sheet for part-specific information, such as which versions of the peripheral modules are on your device.

This book explores the different peripheral modules found in the RS08 Family of MCUs.

Each section of this users guide contains the following topics:

- Programmer’s model register figure for quick reference
- Example code
- Supplemental information supporting the code

All code is available inside a CodeWarrior project, or from Freescale’s Web site in RS08QGUGSW.zip.

In-depth material about using the RS08 modules is also available in Freescale’s application notes. See the Freescale Web site: http://freescale.com

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to: http://freescale.com

Topic Reference

NOTE

- This example code has been developed using the CodeWarrior Development Studio for HC(S)08 v5.1, and expressly made for the MC9RS08KA2 in the 8-pin package. Changes may be needed before the code can be used with other RS08 MCUs.
- This example code is expressly made to operate in an 8-pin package. Therefore, the PTA3 pin shares functionality with background mode. In background mode, the PTA3 pin is not available for application purposes. In run mode, the

Revision History

Date	Revision Level	Description	Page Number(s)
21-May-06	0	Initial public release.	N/A
25-Apr-07	1	Replaced schematic	31
Nov-11	2	Corrected equation	8

Freescale Semiconductor
Users Guide

Using the Analog Comparator (ACMP) for the RS08 Microcontrollers

by: Oscar Luna González and Alan Led Collins Rivera
RTAC Americas
México 2011

Table of Contents

1 Overview

This is a quick reference for using the analog-to-digital comparator (ACMP) module on an MC9RS08KA2 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application. Refer to the data sheet for your device.

The ACMP module provides a circuit for comparing two analog input voltages or for comparing one analog input voltage with an internal reference voltage. Inputs of the ACMP module can operate across the full range of the supply voltage.

ACMP Quick Reference

SIP1	R	0	0	0	KBI	ACMP	MTIM	RTI	LVD
	W								

System Interrupt Pending Register
KBI – pending interrupt from the KBI module
ACMP – pending interrupt from ACMP module
MTIM – pending interrupt from MTIM module
RTI – pending interrupt from RTI module
LVD – pending interrupt from LVD module

ACMPSC	R	ACME	ACBGS	ACF	ACIE	ACO	ACOPE	ACMOD	
	W								

ACMP Status and Control Register
ACME – enable module
ACBGS – selects ACMP+ input pin or internal bandgap reference
ACF – set when event occurs
ACIE – interrupt enable
ACO – read analog comparator output
ACOPE – enable comparator output
ACMOD[1:0] – sets compare mode

The analog comparator (ACMP) module consists of two analog inputs called ACMP+ and ACMP–, and one digital output called ACMPO. ACMP+ serves as a non-inverting analog input. ACMP– serves as an inverting analog input. ACMPO serves as a digital output and can be enabled to drive an external pin. Be aware that interrupts in RS08 MCUs must be polled by software because module hardware interrupts donot exist in RS08 MCUs.

Table 1. ACMP Pin Functionality

Signal	Function	I/O
ACMP–	Inverting analog input to the ACMP (negative input)	I
ACMP+	Non-inverting analog input to the ACMP (positive input)	I
ACMPO	Digital output of the ACMP	O

2 Code Example and Explanation

This project has been developed using CodeWarrior v5.1 and a Softec board.

The project (Analog_Comparator_Module.mpc) implements the ACMP function, selecting a rising- or falling-edge event to trigger an interrupt. Following are the main functions:

- Loop — Endless loop polling and waiting for the ACMP interrupt to occur.
- InitConfig — Configures the MCU to work with the internal oscillator at 8 MHz (bus speed), configures port A, and enables the ACMP module to work with ACMP– and ACMP+ inputs.
- ACMP_Isr — Toggles an LED after a rising- or falling-edge event occurs.

This example compares two different voltages using the ACMP module. Inverted analog input is fed using a potentiometer that is attached to the pin (ACMP–). The ACBGS bit in the ACMPSC register is cleared by writing a 0 to it. This enables the ACMP+ input to be used as the reference input. The ACMP+ signal is fed through a voltage divider set at ~1.6 V. For more detailed information, refer to the RS08 data sheet.

Every time the ACMP– voltage crosses the ACMP+ reference voltage (ACMP– is greater than ACMP+), the ACMP interrupt is triggered turning off a pin at port A (PTA5). Therefore, every time the ACMP– input voltage is lower than ACMP+ input voltage, the PTA5 pin is turned on.

In this application, the ACMP module is demonstrated by lighting and dimming an LED due to an ACMP interrupt triggered by the comparison voltage between the ACMP– and ACMP+ inputs. Please refer to the source code for more details.

Following are the steps to use the ACMP module for this example:

1. Configure ICS module to work with internal oscillator at 8 MHz (bus speed), configure PTA5 as an output and the rest of the PTA pins as inputs, and configure the ACMP module by enabling ACMP– and ACMP+ pins.

```
;CONFIGURES SYSTEM CONTROL
InitConfig:
  IFNE MODE
    mov #HIGH_6_13(SOPT), PAGESEL
    mov #$01, MAP_ADDR_6(SOPT) ; Disables COP and enables RESET (PTA2) pin
  ELSE
    mov #HIGH_6_13(SOPT), PAGESEL
    mov #$03, MAP_ADDR_6(SOPT) ; Disables COP, enables BKGD (PTA3) and RESET (PTA2)
                               ; pins
  ENDIF
    clr ICSC1 ; FLL is selected as Bus Clock
    TRIM_ICS ; call macro to Trim the ICS at ~8 MHz
    clr ICSC2

;CONFIGURES PORT A

  mov #HIGH_6_13(PTAPE), PAGESEL
  mov #$FF, MAP_ADDR_6(PTAPE) ; Enables internal Pulling device

  mov #HIGH_6_13(PTAPUD), PAGESEL
  mov #$00,MAP_ADDR_6(PTAPUD) ; Configures Internal pull up device in PTA

  mov #$30, PTADD ; PTA5(LED2) and PTA4 (LED1) as outputs
  mov #$00, PTAD

;CONFIGURES ANALOG COMPARATOR
  mov #$B3, ACMPSC ; Selects analog comparator between ACMP- and an ACMP+ (external 1.5
                   ; Volts)
                   ; Compares in output falling and rising edge , ACMP enable,
  rts
```

2. Wait in an infinite loop until an ACMP interrupt is triggered. The ACMP interrupt is polling in the loop. After the ACMP interrupt is detected inside the infinite loop, it automatically jumps into a predefined subroutine (ACMP_Isr). The ACMP interrupt is triggered by rising- or falling- edge events.

```
Loop:
      mov #HIGH_6_13(SIP1), PAGE_ADR
       brset 3, MAP_ADDR_6(SIP1),ACMP_Isr     ; branch if ACMP interrupt pending
       bra Loop                              ; Return to main loop
```

3. After the MCU branches into the ACMP_Isr subroutine, the ACF flag is cleared. After clearing the ACF flag, the ACO bit in register ACMPSC is compared with value 1 to assess whether the non-inverting input (ACMP+) is greater than the inverting input (ACMP–). If non-inverting input is greater than the inverting input, the LED2 subroutine is called to light the LED. This indicates that the ACMP– input is below the reference voltage (ACMP+).

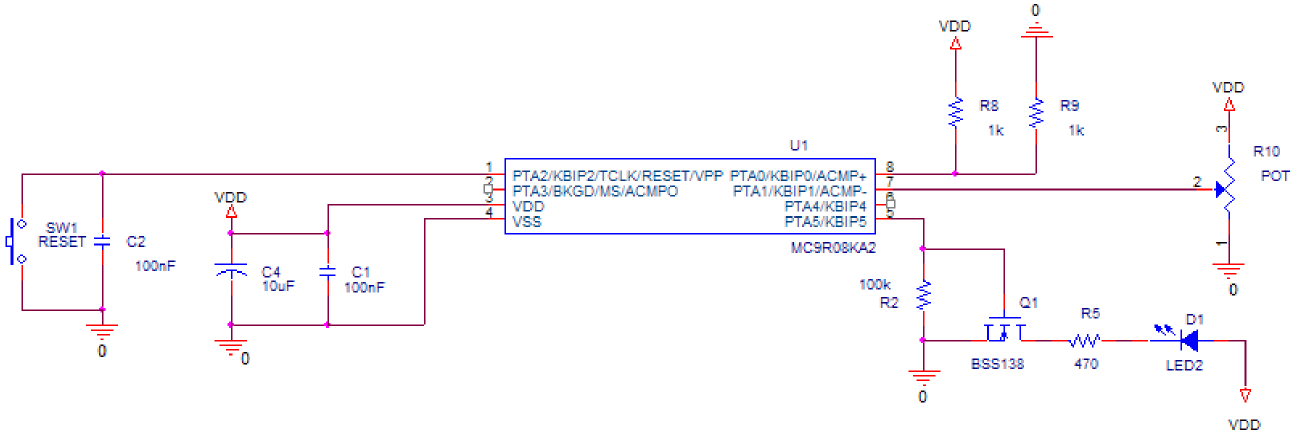
```
ACMP_Isr:
  bset 5,ACMPSC                       ; Clear Compare event flag
  brset 3,ACMPSC,LED2                ; Checks analog comparator output (ACO).
```

```
                                    ; if ACMP+ voltage is greater than ACMP-
                                    ; voltage, ACO will be set with 1
  bclr 5,PTAD
  bra Loop                          ; Return to main loop

LED2:
  bset 5,PTAD                    ; Turn LED on
  bra Loop                          ; Return to main loop
```

3 Hardware Implementation

This schematic shows the hardware used to exercise the code provided.

Figure 1. Hardware Implementation for ACMP



NOTE

- The ACMP module can compare one analog input to an internal reference. This example code is expressly made to configure the ACMP module to work without using the internal bandgap reference voltage.
- The analog comparator circuit is designed to operate across the full range of the supply voltage. Please see the data sheet for your device.

Freescale Semiconductor
Users Guide

Using the Internal Clock Source (ICS) for the RS08 Microcontrollers

by: Gabriel Sanchez Barba and Sergio García de Alba Garcin
RTAC Americas
México 2011

1 Overview

This is a quick reference for using the internal clock source (ICS) module on an MC9RS08KA2 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following examples may be modified to suit your application. Refer to the data sheet for your device.

Table of Contents

ICS Quick Reference

ICSC1	R	CLKS	0	0	0	0	0	IREFSTEN
	W							

ICS Control Register 1
CLKS – selects the clock source that controls bus frequency
IREFSTEN – controls whether the internal reference clock remains enabled while in stop mode

ICSC2	R	BDIV	0	0	LP	0	0	0
	W							

ICS Control Register 2
BDIV – divides the clock source selected by CLKS bit
LP – disables or enables FLL in bypass mode

ICSTRIM	R	TRIM
	W	

ICS Trim Register
TRIM – controls the internal reference clock frequency

ICSSC	R	0	0	0	0	0	CLKST	0	FTRIM
	W								

ICS Status and Control Register
CLKST – indicates current clock mode
FTRIM – controls the smallest adjustment of the internal reference clock

2 Operating Modes and Examples

The ICS provides several options for clock sources. This allows great flexibility for users choosing among precision, cost, current consumption, and performance. The requirements and characteristics of the application being developed determine the importance of each of these factors.

2.1 FLL Engaged Internal (FEI)

FEI is the default mode of operation out of any reset. The MCU also enters this mode when the CLKS bit is cleared. While in this mode, the bus clock is derived from the FLL clock. The FLL locks the bus frequency as shown in Equation 1.

$$f_{Bus} = (f_{irc} * 512) / 2^{BDIV+1} \quad \textit{Eqn. 1}$$

Where f_{irc} is the frequency of the internal reference clock. If the f_{irc} is trimmed to 31.25 kHz and the BDIV value is 3, the resulting bus frequency is:

$$f_{Bus} = (31250 * 512) / 2^{3+1} = 1000000 \text{ Hz} = 1\text{MHz} \quad \textit{Eqn. 2}$$

Example code:

```
        LDA     #$00
        STA     ICSC1
        LDA     #$C0
        STA     ICSC2
```

2.2 FLL Bypassed Internal (FBI)

The MCU enters FBI mode when the CLKS bit is set and the LP bit is cleared. While in this mode, the bus clock is derived as shown in Equation 3.

$$f_{bus} = f_{irc} / 2^{BDIV+1} \quad \textit{Eqn. 3}$$

Where f_{irc} is the frequency of the internal reference clock. If the f_{irc} is trimmed to 31.25 kHz and the BDIV value is 0, the resulting bus frequency is:

$$f_{bus} = 31250 / 2^{0+1} = 15625 \text{ Hz} = 15.625 \text{ kHz} \quad \textit{Eqn. 4}$$

In this mode, the FLL is active but does not affect the bus clock.

Example code:

```
        LDA     #$40
        STA     ICSC1
        LDA     #$00
        STA     ICSC2
```

2.3 FLL Bypassed Internal Low Power (FBILP)

The MCU enters FBILP when the CLKS and LP bits are set. While in this mode, the bus clock is derived as shown in Equation 5:

$$f_{bus} = f_{irc} / \ 2^{BDIV+1} \qquad \textit{Eqn. 5}$$

Where f_{irc} is the frequency of the internal reference clock. If the f_{irc} is trimmed to 31.25 kHz and the BDIV value is 0, the resulting bus frequency is:

$$f_{bus} = 31250 / \ 2^{0+1} = 15625\ \text{Hz} = 15.625\ \text{kHz}$$

The main difference between this mode and FLL bypassed internal (FBI) is that the FLL is not active, which allows the MCU to consume less current.

Example code:

```
        LDA     #$40
        STA     ICSC1
        LDA     #$08
        STA     ICSC2
```

3 Recommendations

When changing from FBILP to either FEI or FBI, or anytime the trim value is written, you must wait for the FLL acquisition time, $t_{Acquire}$, before FLL is guaranteed to be at the desired frequency.

The BDIV bits can be changed at any time. The change to the new frequency occurs immediately.

The TRIM and FTRIM value is not affected by reset.

4 Conclusion

Although the ICS module has few modes of operation, it allows the flexibility of using many different clock speeds for different applications. It also allows the user to determine the amount of energy that is used by the MCU.

Freescale Semiconductor
Users Guide

Using the Keyboard Interrupt (KBI) for the RS08 Microcontrollers

by: Alan Led Collins Rivera, Oscar Luna González, and Gabriel Sanchez Barba
RTAC Americas
México 2011

1 Overview

This is a quick reference for using the keyboard interrupt (KBI) module on an RS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following example may be modified to suit your application — refer to the data sheet for your device.

Table of Contents

KBI Quick Reference

KBISC	R	0	0	0	0	KBF	0	KBIE	KBMOD
	W						KBACK		

Module Configuration
KBF – set when event occurs
KBACK – clears KBF

KBIE – interrupt enable
KBMOD – mode select

KBIPE	R	0	0	KBIPE5	KBIPE4	0	KBIPE2	KBIPE1	KBIPE0
	W								

KBI Pin Enable
KBIPE[5,4,2:0] – enables the corresponding keyboard interrupt pin.

KBIES	R	0	0	KBEDG5	KBEDG4	0	KBEDG2	KBEDG1	KBEDG0
	W								

KBI Edge Select
KBEDG[5,4,2:0] – selects the falling or rising edge/level function of the corresponding pin.

SIP1	R	0	0	0	KBI	ACMP	MTIM	RTI	LVD
	W								

System Interrupt Pending Register
KBI – pending interrupt from the KBI module
ACMP – pending interrupt from ACMP module
MTIM – pending interrupt from MTIM module

RTI – pending interrupt from RTI module
LVD – pending interrupt from LVD module

2 Code Example and Explanation

In this application, one of the KBI pins is used to trigger a routine that lights one LED, then a second LED, then a third LED, and finally powers off all three LEDs. This occurs every time a keyboard event is detected. The MCU is programmed to do the following tasks:

- Use the KBI pin 1 as the interrupt trigger
- Detect falling edges on the selected pin
- Jump to routines that light the appropriate LEDs

This is the initialization code for the keyboard interrupt using the MC9RS08KA2. For this example, two KBI registers (KBISC and KBIPE) are used to customize the module as mentioned above. (For KBIES, use the default value of the register after reset.) During the initialization phase, the interrupt flag is cleared in case there were any false interrupts and the keyboard interrupt is unmasked.

```
    mov #$00,KBIES                  ; Select only falling edges or low-level condition

    mov #$02,KBIPE                  ; PTA1 as KBI

    mov #$06,KBISC                  ; Clear any false interrupts and unmask KBI
```

After the keyboard module is set, the MCU goes into a loop that loads the SIP1 register and checks the KBI bit. If the KBI bit is set, the program jumps to the routines that determine which LEDs to power on. If the KBI bit is not set, then the program branchs to the start of the loop until the KBI bit is set.

```
Loop:

        mov #HIGH_6_13(SIP1), PAGE_ADR

        brset 4, MAP_ADDR_6(SIP1),Led2  ; Branch if KB interrupt pending

        bra Loop
```

After the program branches to the Led2 label, it acknowledges the KBI interrupt by writing a 1 to the KBIACK bit. The program then checks whether LED2 is already set. If it is set, the program branches to test the next LED (LED1). If LED2 is not set, the program toggles the port pin to light LED2 and returns to the beginning of the main loop.

```
Led2:

        bset 2,KBISC            ; Acknowledge to KBI

        brset 5,PTAD,Led1       ; Branch if LED2 set

        lda PTAD

        eor #$20                ; Toggles LED2

        sta PTAD

        bra Loop
```

The other labels (Led1 and Led0) also check whether the current LED is lit. If it is, the program continues to the next LED. If it is not, the program lights the LED. After all LEDs are lit, the program toggles all three port pins so none of the three LEDs is lit and then returns to the beginning of the main loop.

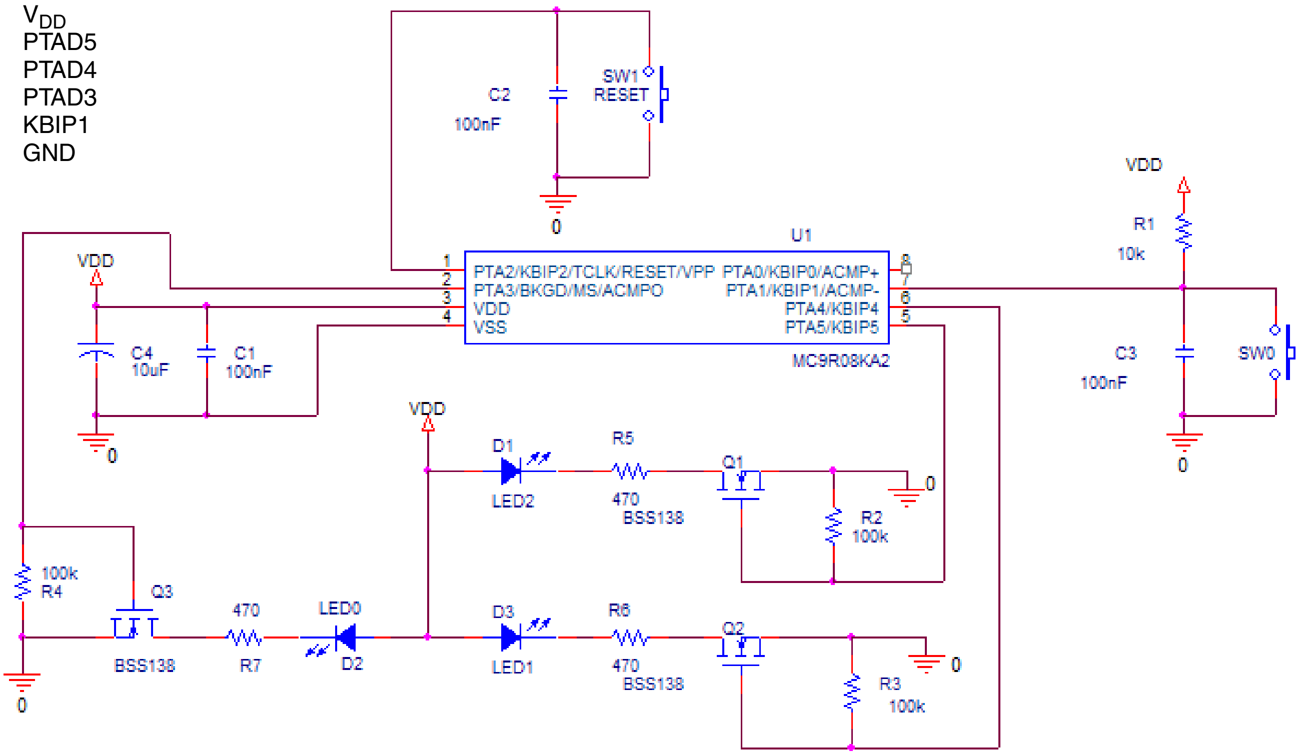
```
Off:

               clr PTAD               ; Powers off all LEDs

bra Loop
```

3 Hardware Implementation

For this example the hardware implementation is fairly simple because we are using only PTA7 as a KBI input. Only the following six pins of the MCU are needed:

- Supply voltage pin
- Ground reference pin
- KBI pin as interrupt input
- Three I/O pins as output; LEDs are used as visual display of the KBI module proper function

Figure 1. Hardware Implementation for KBI



Freescale Semiconductor
Users Guide

Using the Modulo Timer (MTIM) for the RS08 Microcontrollers

by: Oscar Luna González and Alan Led Collins Rivera
RTAC Americas
México 2011

1 Overview

This is a quick reference for using the modulo timer (MTIM) module on an MC9RS08KA2 microcontroller (MCU). The basic functional description and configuration options are provided. The following examples may be modified to suit your applications. Refer to the data sheet for your device.

Table of Contents

MTIM Quick Reference

SIP1	R	0	0	0	KBI	ACMP	MTIM	RTI	LVD
	W								

System Interrupt Pending Register
KBI – pending interrupt from the KBI module
ACMP – pending interrupt from ACMP module
MTIM – pending interrupt from MTIM module
RTI – pending interrupt from RTI module
LVD – pending interrupt from LVD module

MTIMSC	R	TOF	TOIE		TSTP	0	0	0	0
	W			TRST					

MTIM Status and Control Register
TOF – set Overflow flag when counter
TOIE – enables MTIM overflow interrupts
TRST – reset MTIM counter if it set with 1
TSTP – run or stop MTIMs counter

MTIMCLK	R	0	0	CLKS	PS
	W				

MTIM Clock Configuration Register
CLKS – select MTIM clock source
PS – select one of nine prescaler outputs

MTIMCNT	R	COUNT
	W	

MTIM Counter
MTIM count – current value of the 8-bit counter

MTIMMOD	R	MOD
	W	

MTIM Modulo Register
MOD – modulo value

The MTIM comprises an 8-bit up-counter with a source clock selector and a prescaler block that allows the counter to generate larger time bases. The prescaler block helps to divide the selected source clock frequency by 1, 2, 4, 8, 16, 32, 64, 128, and 256.

2 Code Example and Explanation

This project has been developed using CodeWarrior v5.1 and the DEMO9RS08KA2 board from Softec.

The project (MTIM_Module.mpc) uses the MTIM module of the RS08 MCU by selecting reference source clock from bus clock. Following are the main functions:

- Loop — Endless loop polling and waiting for the MTIM overflow interrupt to occur.
- InitConfig — Configures the MCU to work with the internal oscillator at 8 MHz (bus speed), configure port A pins 3, 4, and 5 as outputs, and enables the MTIM module to work with an operation mode called modulo.
- MTIM_Isr — Toggles an LED after a rising- or falling-edge event occurs.

This example shows how to configure the MTIM module to use one of four clock source selecting choices that the module has. The clock source option in this example is taken directly from the bus clock. Also, the MTIM module is configured to set the prescaler value at 256. Configuring the prescaler with a value of 256 divides the selectable clock source (bus clock, in this case) by 256. Doing this, every 32 µs (8 MHz/256), a count in MTIM modulus counter is increased, so every 8.192 ms (32 µs * 256), an MTIM overflow interrupt is generated and attended by polling SIP1 register to branch into the proper subroutine.

The MTIM module has been configured to work with modulo operation mode, so every time the modulus counter reaches the overflow compare value, an interrupt is triggered. After the overflow interrupt triggers, an infinite loop in the code polls the MTIM interrupt flag in SIP1 register to compare and branch into a subroutine called MTIM_Isr (if MTIM flag is set).

In this application, the MTIM module is demonstrated by lighting and dimming three LEDs (in run mode) approximately every second. Because, the application is running at 8 MHz (bus clock) and the MTIM prescaler is set at the maximum divider value (256), a 1 second interrupt cannot be reached. To solve this problem, a subroutine called 'Count' counts 122 times to generate an approximate base time of 1 second. After the 1 second base time has elapsed, the application follows a sequence pattern by lighting LED0, then lighting LED1, and then lighting LED2 every second. After the fourth second, all LEDs are dimmed.

Please refer to the source code for more details.

Following are the steps to use the ACMP module as in this example:

1. Configure the ICS module to work with internal oscillator at 8 MHz (bus speed), configure PTA4 and PTA5 as outputs and the rest of the PTA pins as inputs, and configure MTIM module in modulo mode.

```
InitConfig:
      IFNE MODE
        mov #HIGH_6_13(SOPT), PAGESEL
        mov #$01, MAP_ADDR_6(SOPT) ; Disables COP and enables RESET (PTA2) pin
      ELSE
        mov #HIGH_6_13(SOPT), PAGESEL
        mov #$03, MAP_ADDR_6(SOPT) ; Disables COP, enables BKGD (PTA3) and RESET (PTA2)
                                   ; pins
        ENDIF
        clr ICSC1 ; FLL is selected as Bus Clock (8MHz)
        TRIM_ICS ; call macro to Trim the ICS at ~8 MHz
        clr ICSC2
        rts

;CONFIGURES PORT A
   mov #$30, PTADD             ; PTA4(LED1),PTA5(LED0) as outputs
   clr PTAD                    ; Clears PTA

;CONFIGURES TIMER
   mov #$70, MTIMSC              ; Enables interrupt, stops and resets timer counter
   mov #$FF, MTIMMOD             ; MTIM modulo with 256 counts before
                                 ; interrupt.
   mov #$08, MTIMCLK             ; Selects internal clock as reference bus
                                 ; clock (8 MHz) with prescaler 256

                                 ;         Bus Clk
                                 ;   --------------------- = Timer interrupt
                                 ;   (preescaler)*(MTIMMOD)

                                 ; (increments timer counter every 32
                                ;  us)(flag interrupt every 8.192ms)

   bclr 4,MTIMSC               ; MTIM counter is Active
    rts
```

2. Wait in an infinite loop until the MTIM overflow flag is triggered. The MTIM interrupt is polling in the loop. After the MTIM interrupt is detected polling the SIP1 register, it automatically jumps into a predefined subroutine called MTIM_Isr.

```
Loop:
   mov #HIGH_6_13(SIP1), PAGE_ADR
   brset 2, MAP_ADDR_6(SIP1),Count  ; branch if timer interrupt pending
   bra Loop
```

3. After an MTIM overflow has been detected by the infinite loop, the MTIM_Isr subroutine is called and the overflow flag (TOF) is cleared. A variable called Counter stores 122 iterations to generate an approximate base time of 1 second. After 1 second has elapsed, LED0 is lit; then one second later, LED1 is lit; then one second later, LED2 is lit. After three seconds all three LEDs are dimmed. This sequence is generated forever in the application.

```
MTIM_Isr:
   lda  MTIMSC                  ; Clear overflow interrupt flag
   mov  #$60,MTIMSC              ; Reset MTIM Counter, Clear  overflow flag
   lda  Counter                  ; Store new value in the Accumulator
   cbeqa #122,Led2               ; (8.192ms*122) =~ 1 seg
    inc  Counter                 ; Increase counter value
    bra  Loop

Led2:
      clr Counter               ; Reset Counter Value
      brset 5,PTAD,Led1       ; Branch if LED2 is set
      lda PTAD
      eor #$20                  ; Toggles LED2
      sta PTAD
      bra Loop

Led1:
      brset 4,PTAD,Led0         ; Branch if LED1 is set
      lda PTAD
      eor #$10                  ; Toggles LED1
      sta PTAD
      bra Loop

Led0:
      brset 3,PTAD,Off          ; Branch if LED0 is set
      lda PTAD
      eor #$08                   ; Toggles LED0 (not available in background mode)
      sta PTAD
      bra Loop

Off:
      clr PTAD                  ; Turn Off all LEDs
      bra Loop
```

3 Hardware Implementation

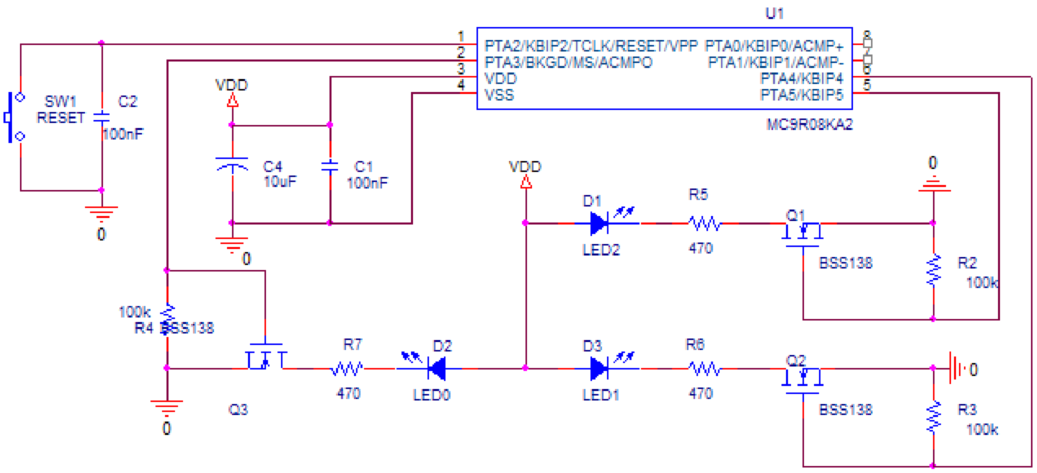
This schematic shows the hardware used to exercise the code provided. For this example, the hardware implementation is fairly simple, because the application itself uses only six of the pins available in the MCU.

Following six pins of the MCU are needed:

- Supply voltage pin
- Ground reference pin
- Reset pin

- Three pins configured as outputs to turn on-off the three LEDs. The LEDs are the visual display that shows the MTIM is functioning correctly.

Figure 1. Hardware Implementation for MTIM



NOTE

- The MTIM module can operate using two more operating modes (stop mode and free-running mode). This example code is made to configure the MTIM module to work in modulo operating mode. Please refer to data sheet for more details.

Freescale Semiconductor
Users Guide

Using the Real-Time Interrupt (RTI) for the RS08 Microcontrollers

by: Oscar Luna González
RTAC Americas
México 2011

1 Overview

This is a quick reference for using the real-time interrupt (RTI) module on an MCRS08KA2 microcontroller. Basic information about the functional description and configuration options is provided. The following example may be modified to suit your application. Refer to the data sheet for your device.

Table of Contents

RTI Quick Reference

SIP1	R	0	0	0	KBI	ACMP	MTIM	RTI	LVD
	W								

System Interrupt Pending Register
KBI – pending interrupt from the KBI module
ACMP – pending interrupt from ACMP module
MTIM – pending interrupt from MTIM module
RTI – pending interrupt from RTI module
LVD – pending interrupt from LVD module

MTIMSC	R	RTIF	0	RTICLKS	RTIE	0	RTIS
	W		RTIACK				

MTIM Status and Control Register
RTIF – RTI interrupt flag
RTIACK – bit used to acknowledge RTI interrupt request
RTICLKS – RTI clock source
RTIE – RTI interrupt
RTIS – RTI interrupt period

The real-time interrupt module allows the generation of seven different interrupt time bases. The RTI has two different sources to be driven: the 1-kHz internal clock reference or the trimmed 32-kHz internal clock reference. The 32-kHz internal clock reference is divided by 32 to generate a more accurate 1-kHz clock. When an application requires a more accurate base time, run the RTI module from 32-kHz internal source after the internal oscillator has been trimmed.

Seven different options are available in the RTI module to configure a periodic interrupt base time. These options are: 1024 ms, 512 ms, 256 ms, 128 ms, 64 ms, 32 ms, and 8 ms.

RTI module remains active in run, wait, or stop mode and also wakes the MCU from wait or stop modes.

2 Code Example and Explanation

This project has been developed using CodeWarrior v5.1 and DEMO9RS08KA2 board from Softec.

The project (RTI.mpc) implements the use of the RTI module of the RS08 MCU by selecting the reference source clock from the 32-kHz internal trimmed source clock. The main functions are:

- Loop — Endless loop polling and waiting for the RTI interrupt to occur.
- InitConfig — Configures the MCU to work with the internal oscillator at 8 MHz (bus speed), sets a trim value to the internal oscillator, configures port A pin 5 as output.
- InitRTI — Initializes the RTI module to work with the 32-kHz trimmed internal source clock and set an initial interrupt period of 1024 ms.
- RTI_Isr — Checks whether a counter variable has reached a value of 0 to change the periodic blinking time of an LED. A status variable is also checked to determine whether the 1024 ms or 256 ms periodic base time must be set to change the periodic interrupt base time of the RTI module after 10 counts. Further detailed explanation is given below:

This example shows how to configure the RTI module using one of the two possible clock source references available. For this document, the 32-kHz trimmed internal source clock has been used instead of the 1-kHz internal source clock that the RTI module has. This 32-kHz trimmed internal reference is divided by 32 to generate a more accurate 1-kHz clock. For more detail, refer to RS08 datasheet.

This example code blinks an LED (PTA5) ten times every 1024 ms. After the tenth time, the periodic interrupt time of the RTI module changes to generate a periodic interrupt time every 256 ms. After the tenth time of generating an interrupt every 256 ms, the RTI changes again to generate a 1024 ms interrupt time. It continues to change from 1024 ms interrupt time to 256 ms interrupt time every ten counts.

To accomplish the requirements of this example, the RTI module is initially configured to generate a 1024 ms interrupt using the internal 32-kHz trimmed oscillator frequency. After the initial configuration, the periodic interrupt time of the RTI is changed from one interrupt period to another every ten times.

Refer to the source code for more details.

Following are steps to use the RTI module as in this example:

1. Configure ICS module to work with the internal oscillator at 8 MHz (bus speed), trim the internal oscillator to generate the fewest possible errors, configure PTA5 as an output and the rest of the PTA pins as inputs, and initialize a counter variable (which changes from one interrupt frequency to another interrupt frequency).

```
InitConfig:
;CONFIGURES SYSTEM CONTROL
  IFNE MODE
    mov #HIGH_6_13(SOPT), PAGESEL
    mov #$01, MAP_ADDR_6(SOPT) ; Disables COP and enables RESET (PTA2) pin
  ELSE
    mov #HIGH_6_13(SOPT), PAGESEL
    mov #$03, MAP_ADDR_6(SOPT) ; Disables COP, enables BKGD (PTA3) and RESET (PTA2)
                               ; pins
  ENDIF
    clr ICSC1 ; FLL is selected as Bus Clock
    TRIM_ICS ; call macro to Trim the ICS at ~8 MHz
    clr ICSC2

;CONFIGURES PORT A
  mov #$30, PTADD ; PTA4(LED1),PTA5(LED0) as outputs
  clr PTAD ; Clears PTA
  mov #20,Counter ; Initialize Counter variable
  rts
```

2. Configure RTI module to work with the internal 32-kHz trimmed frequency and set the interrupt time every 1024 ms.

```
   mov   #$37,MAP_ADDR_6(SRTISC)      ; 32-kHz trimmed internal source selected,
                                      ; Interrupt enabled, 1.024s interrupt period base
   rts
```

3. Wait in an infinite loop until the RTI overflow flag is triggered. RTI interrupt is polling in the loop. After the RTI interrupt is detected polling the SIP1 register, it automatically jumps into a predefined subroutine called RTI_Isr.

```
Loop:
   wait                               ; Enter into wait mode until an interrupt arrives
   mov   #HIGH_6_13(SIP1), PAGESEL
   brset 1, MAP_ADDR_6(SIP1),RTI_Isr  ; branch if RTI interrupt pending
   bra Loop
```

4. After RTI interrupt has been detected by the infinite loop, the RTI_Isr subroutine is called and the acknowledge flag (ACK) is cleared. A variable called Counter stores one iteration every RTI interrupt event. Every iteration is shown by turning on/off an LED (PTA5). After 10 interrupt times, the RTI periodic interrupt is changed from 1024 ms to 256 ms and the LED blinks faster. This process continues to change the periodic interrupt from 1024 ms to 256 ms and vice versa.

```
RTI_Isr:
   bset 6,MAP_ADDR_6(SRTISC)     ; Clear RTI interrupt flag (ACK)
   dbnz Counter,Blink_Led        ; Check if Counter = 0 in order to change RTI interrupt period
   bsr  RTIPeriod                ; branch to blink subroutine
   bra  Loop

RTIPeriod:
```

Using the Real-Time Interrupt (RTI) for the RS08 Microcontrollers

```
    brset 0,StatusPeriod,_1024msPeriod  ; if StatusPeriod = 1, Configure RTI with 1024ms period
    brclr 0,StatusPeriod,_256msPeriod   ; if StatusPeriod = 0, Configure RTI with 256ms period
    bra Loop

_256msPeriod:
    mov #RTI_256,Period
    ChangeRTIPeriod                     ; call macro
    bra Loop

_1024msPeriod:
    mov #RTI_1024,Period
    ChangeRTIPeriod                     ; call macro
    bra Loop

Blink_Led:
    lda PTAD
    eor #$20                            ; Toggles LED2 (PTA5)
    sta PTAD                            ; blink LED2
    bra Loop
```

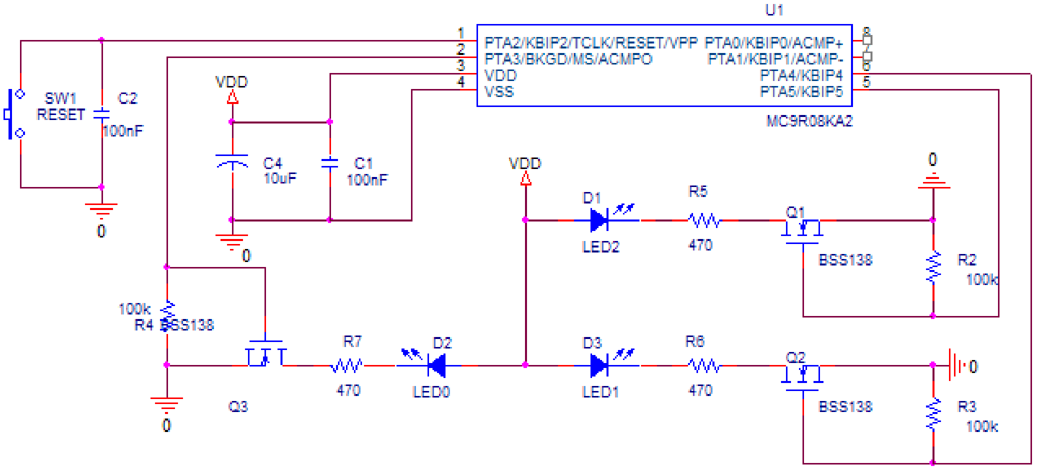
3 Hardware Implementation

This schematic shows the hardware used to exercise the code. For this example, the hardware implementation is fairly simple because the application itself uses only four of the eight pins available in the MCU.

Follwoing four pins of the MCU are needed:

- Supply voltage pin
- Ground reference pin
- Reset pin
- One available pin configured as output to turn on-off one LED. The LED is the visual display that shows the RTI is functioning correctly).

Figure 1. Hardware Implementation for RTI



Freescale Semiconductor
Users Guide

RS08 Addressing Modes

by: José Ruiz Juárez
Oscar Luna González
RTAC Americas
México 2011

1 Overview

This is a quick reference to the addressing modes in the MC9RS08KA2 microcontroller. The RS08 family introduces a paging scheme to allow the CPU to access the whole 16K memory space. Basic information about addressing modes is given to introduce you to the new features on the MCU.

Table of Contents

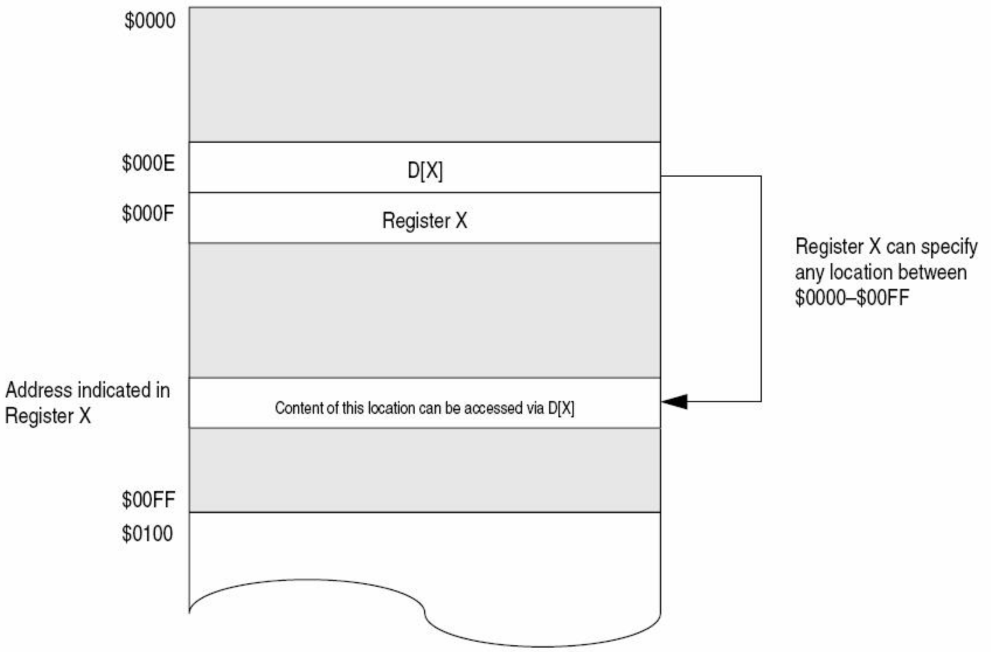
2 RS08 Memory Map

The RS08 family introduces a standard paging scheme that allows the CPU to access the whole 16K memory space using 64-byte paged windows. The page window is located from $C0 to $FF. 256 pages already segmented in 64-bytes per page window exist in the RS08 family to segment the full 16-KB memory map of the microcontroller.

To access every memory space in the microcontroller, the RS08 MCU implements a memory mapped indirect memory access using two registers:

Register	Description
X	Contains the address of memory to be accessed when accessing register D[X]. It is mapped in address location $000F
D[X]	Address memory pointed to by register X. The address location of D[X] register is $000E

Using these registers the user can have easy access to the first 256 bytes of the memory map. This is an advantage of tiny and short addressing modes and a powerful solution for indexed access.


$0000
$000E
D[X]
$000F
Register X
Register X can specify any location between $0000–$00FF
Address indicated in Register X
Content of this location can be accessed via D[X]
$00FF
$0100

The RS08 MCUs have pseudo instructions to implement all X inherent instructions with a single instruction.

```
LDX          LDA $0F
LDX #SC0     MOV #SC0,$0F
LDX $44      MOV $44,$0F
```

Also, all zero byte indexed operations ‘,X’ can be implemented simply by using the address $0E in direct addressing instructions.

```
STA  ,X      STA $0E
EOR ,X       EOR $0E
MOV ,X,$C0   MOV $0E,$C0
```

2.1 Addressing Modes

2.1.1 Inherent

The CPU knows everything it needs. No addressing information is required because only internal registers can be accessed with this addressing mode; there are no operands.

```
inca
clra
```

2.1.2 Immediate

The operands is located immediately after the opcode in the instruction stream. Immediate addressing is used when the programmer wants to use an explicit value. The # symbol is the operator that indicates immediate addressing mode.

```
Lda #$C0
```

2.1.3 Tiny

This mode can accessed with only the first 16 bytes in the address map ($0000–$000F) A system can be optimized by placing the most computation-intensive data in this area memory. This addressing mode is available for INC, DEC, ADD, and SUB instructions.

```
Tax
```

2.1.4 Short

CLR, LDA, and STA are some instructions that short addressing mode supports. Short addressing mode is capable of addressing only the first 32 bytes in the address map.

```
Sta PAGESEL
```

2.1.5 Direct

This addressing mode is used to access operands located in direct address space ($0000 through $00FF).

```
lda $C0
```

2.1.6 Extended

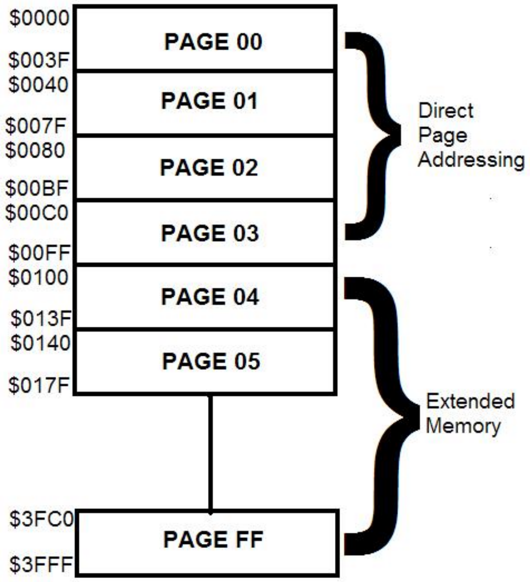
There is only one instruction for extended addressing mode: JMP can jump to all of the memory map. The destination address is given by two bytes next the opcode.

```
Indexed

Jmp $3800
```

2.2 Page Mapping

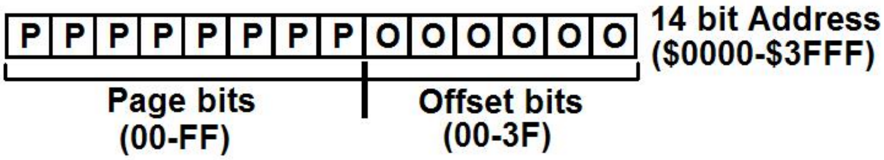
The memory in the RS08 MCU is organized in 64 byte pages. The first page (0) starts in $0000 and ends in $003F. Direct addressing mode works in the address range $0000 to $00FF.

RS08 does not directly support extended addressing; access to extended address is available through the paging window.



The paging window is $00C0 and ends at $00FF (page 03). These 64 bytes contains a mirror of the page selected by the page register ($001F).

Accessing extended memory through the page window involves simply setting the page number from the upper eight bits of the address and the offset within the page window from the lower six bits of the extended address.



In other words, shift right six times the 14 bit address and store the result in the page register. The six least significant bits of the 14-bit address are the offset in the window page.

For example, to access to the $0142 memory location, load a 0x05 ($0142 shifted right 6 times) in the page register. The offset value is two but to accomplish the 8-bit address, set the two most significant bits to 1. That results in 0xC2, which is the 3rd value of the mirror page.

2.3 Implementation Example

Having a table with 256 elements (4 pages of 64 bytes), the program automatically calculates the page and the offset values depending the position that want to access.

The table starts at 0x3E00 in the memory (page F8 and offset 0).

```
Table
ORG $3E00
 dc.b 0,5,12,20,27,34,41,47,53,59,65,71,77,82,87,92
 dc.b 97,102,107,111,116,120,124,128,132,135,139,142,146,149,152,155
 dc.b 158,161,164,167,170,172,175,177,179,182,184,186,188,190,192,194
 dc.b 196,197,199,201,202,204,206,207,208,210,211,212,214,215,216,217
 dc.b 218,219,221,222,223,223,224,225,226,227,228,229,229,230,231,232
 dc.b 232,233,234,234,235,235,236,236,237,238,238,238,239,239,240,240
 dc.b 241,241,241,242,242,243,243,243,244,244,244,244,245,245,245,246
 dc.b 246,246,246,247,247,247,247,247,248,248,248,248,248,248,249,249
 dc.b 249,249,249,249,250,250,250,250,250,250,250,250,250,251,251,251
 dc.b 251,251,251,251,251,251,251,251,252,252,252,252,252,252,252,252
 dc.b 252,252,252,252,252,252,252,252,252,252,253,253,253,253,253,253
 dc.b 253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253
 dc.b 253,253,253,253,253,253,253,253,253,253,253,253,253,253,253,253
 dc.b 253,253,253,253,253,253,254,254,254,254,254,254,254,254,254,254
 dc.b 254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254
 dc.b 254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254
```

2.4 Search Routine

The CounterValue has the position of the table data that contains the value that needs to be accessed. Because, CounterValue is an 8-bit variable, only the two MSB bits are needed to calculate the page. The page register is loaded with the eight most significant bits of origin table address plus the two more significant bits of CounterValue.

```
lda CounterValue            ; load the search value in the accumulator
rola                        ; getting 2 MSB
rola
rola
and #$03                    ; make a logical AND
add #(Table_Data>>6)        ; address of table data>>6 + calculated page
sta PAGESEL                 ; load Page in Page Register (Now page is set)
```

Next, reload the counter value on the accumulator and extract the six LSBs with a logical AND, add a 0xC0 (page window) and store it in x register.

Now, D[x] contains the required value.

```
lda CounterValue            ; Reload search value on accumulator
and #$3F                    ; Extract 6 LSB
add #$C0                    ; Add offset to the first page window address
tax                         ; Store address in X register

lda ,x                      ; Load table result
sta ConvertedValue          ; Store result on Convertedvalue variable
```

Freescale Semiconductor
Users Guide

Interrupt Handling on RS08 MCUs

by: Alan Led Collins Rivera and Oscar Luna González
RTAC Americas
México 2011

1 Overview

Table of Contents

This is a quick reference for using interrupts on an RS08 microcontroller (MCU). There are no interrupt vectors on the MCU; every module interrupt request must be attended by polling flags.

Because the RS08 does not have interrupt vectors, it is necessary to poll every enabled interrupt except the reset interrupt. When a reset occurs, the program counter starts at $3FFD. A jump instruction must be placed in this location for correct reset operation.

Interrupts Quick Reference

SIP1	R	0	0	0	KBI	ACMP	MTIM	RTI	LVD
	W								

System Interrupt Pending Register
KBI – pending interrupt from the KBI module
ACMP – pending interrupt from ACMP module
MTIM – pending interrupt from MTIM module
RTI – pending interrupt from RTI module
LVD – pending interrupt from LVD module

MTIMSC	R	TOF	TOIE		TSTP	0	0	0	0
	W			TRST					

MTIM Status and Control Register
TOF – Set Overflow flag when counter
TOIE – Enables MTIM overflow interrupts
TRST – Reset MTIM counter if it set with 1
TSTP – Run or Stop MTIM counter

KBISC	R	0	0	0	0	KBF	0	KBIE	KBMOD
	W						KBACK		

KBI Status and Control Register
KBF – Indicates that a keyboard interrupt is detected
KBACK – Writing a 1 clears KBF
KBMOD – Controls the detection mode of the keyboard interrupt pins (0-detects edges only, 1-Detects both edges and levels).
KBIE – Enables keyboard Interrupt

The interrupt request for every module may be checked using two different registers:

- System interrupts pending register (SIP)
- The module registers

The programmer chooses between these two ways of polling the interrupt events. Using the SIP1 register provides the advantage of setting priority to interrupts, but this register is mapped in the $0202 memory address, so indexed addressing is needed to access it. One more instruction is needed for writing in the page register the page of the SIP1 register address.

Using the module register flag for polling may provide faster acknowledgement of the event that interrupted the MCU.

You can determine how to poll the interrupts based on your preference and on the requirements of the application.

2 Code Example and Explanation

This project has been developed using CodeWarrior v5.1 and the DEMO9RS08KA2 board from Softec.

The project shows how to use interrupts in the MTIM and KBI modules. Following are the main functions:

- Entry — Configures the system control, ports, and modules that will be used (ICS, KBI, MTIM)
- Loop — Endless loop polling and waiting for the MTIM or KBI overflow interrupt to occur
- Timer — Toggles an LED every 0.5 seconds.
- Kboard — Chooses whether LED2 or LED1 will toggle.

This example shows how to manage interrupts requests by polling flags. The clock source option in this example is taken directly from the bus clock that is trimmed to 8 MHz. Configuring the MTIM prescaler with a value of 256 divides the selectable clock source (bus clock, in this case) by 256. Doing this, every 32 s (8 MHz/256), a count in MTIM modulus counter is increased, so every 8.192 ms (32 s * 256), an MTIM overflow interrupt is generated. When this event occurs, a counter is checked, so every 0.5 seconds an LED (LED2) will toggle.

The keyboard interrupt is also enabled in this example. When a keyboard event occurs, the dim toggling LED2, and another LED (LED1) will replace its function of toggling every 0.5 seconds. If the keyboard interrupt is received again, LED1 dims, and LED2 toggles again.

The interrupts generated by MTIM will be attended by polling SIP1 or MTIMSC register to branch into the proper subroutine. KBI interrupts may be attended checking SIP1 or KBISC register. Please refer to the source code for more details.

1. Configure ICS module to work with internal oscillator at 8 MHz (bus speed), configure PTA4 and PTA5 as outputs, configure KBI detecting edges only in PTA1, and configure the MTIM module in free running mode.

```
Entry:
;---------------------------------------
; CONFIGURES SYSTEM CONTROL
;---------------------------------------
 IFNE MODE
```

```
   mov #HIGH_6_13(SOPT), PAGE_ADR
   mov #$01, MAP_ADDR_6(SOPT)      ;Disables COP and enables RESET pin

 ELSE
   mov #HIGH_6_13(SOPT), PAGE_ADR
   mov #$03, MAP_ADDR_6(SOPT)      ;Disables COP, enables BKGD and RESET pins

 ENDIF
;--------------------------------------
; CONFIGURES CLOCK (FEI Operation Mode)
;--------------------------------------
   clr ICSC1                 ; Selects FLL as clock source and disables it in stop mode
 mov #$98,ICSTRM
 clr ICSC2                ; ICSOUT = DCO output frequency
 mov #$04,ICSSC

;--------------------------------------
; CONFIGURES PORT A
;--------------------------------------
   mov #$30, PTADD     ;PTA4(LED1),PTA5(LED2) as outputs
   clr PTAD         ; Clears PTA
;--------------------------------------
; CONFIGURES KEYBOARD INTERRUPT MODULE
;--------------------------------------
   mov #HIGH_6_13(PTAPE), PAGE_ADR
   mov #$FF, MAP_ADDR_6(PTAPE); Enables internal Pulling device
   mov #HIGH_6_13(PTAPUD), PAGE_ADR
   clr MAP_ADDR_6(PTAPUD)  ; Configures Internal pull up device in PTA
   mov #$02,KBIPE; PTA1 as KBI
   mov #$06,KBISC; KBI interrupt request enable
;-------------------------------------
; CONFIGURES TIMER
;-------------------------------------
   mov #$70, MTIMSC              ; Enables interrupt, stops and resets timer counter
   mov #$00, MTIMMOD          ; MTIM modulo in free running mode (256 counts).
      mov #$08, MTIMCLK             ; Selects internal clock as reference bus clock
                              ; (8 MHz) ; with prescaler 256
                                  ;          Bus Clk
                              ; ----------------------- = Timer interrupt
                              ; (preescaler)*(MTIMMOD)
                              ; (increments timer counter every 32 us)(flag interrupt
                                  ; every 8.192ms)

bclr 4,MTIMSC        ; MTIM counter is Active
```

2. The MCU waits until an interrupt is generated. The source of the interrupt event may be checked in two ways:
   — Using the SIP1 register: The MCU has a system interrupt pending register (SIP1) that contains all the modules’ interrupt flags.

```
Loop:
    wait                                          ; MCU in low voltage mode
    mov #HIGH_6_13(SIP1), PAGE_ADR
    brset 4, MAP_ADDR_6(SIP1),Kboard        ; Branch if KB interrupt pending
    brset 2, MAP_ADDR_6(SIP1),Timer         ; branch if timer interrupt pending
    bra Loop
```

```
Using the modules registers: Every module that use interrupts, has a flag interrupt in
its control register. If an interrupt occurs because of a specific module, that exactly
module will set a flag interrupt.  Checking the interrupt event by the module register is
one instruction faster than using the SIP1 register, because of the location in memory of
the SIP1 register.
Loop:
    wait                                    ; MCU in low voltage mode
    brset 3,KBISC,Kboard                    ; Branch if KB interrupts pending
    brset 7,MTIMSC,Timer                    ; branch if timer interrupt pending
    bra Loop
```

— If the interrupt detected by the infinite loop is from MTIM, then branches to the Timer label. The overflow flag (TOF) is cleared. A variable called Counter stores 61 interactions to generate a base time of approximately 0.5 seconds. Every half second, an LED toggles. The LED that toggles is chosen by the Kboard function.

```
Timer:
   bset 5,MTIMSC         ; Acknowledge to Timer
   lda count
   cbeqa #61,Toggle      ; (8.192ms*61) =~ 0.5 seg
   inc count
   bra Loop

Toggle:
  clr count
  brset 0,change,LED1    ; Checks which LED will toggle.
  lda PTAD
  eor #$20               ; Toggles LED2
  sta PTAD
  bra Loop

LED1:
  lda PTAD
  eor #$10               ; Toggles LED1
  sta PTAD
  bra Loop
```

3. If the interrupt detected by the infinite loop is from KBI, then branches to the Kboard label. The keyboard interrupt flag (KBF) is cleared and both LEDs are dimmed. A variable called change is toggled so the next time the MTIM interrupts the MCU, the LED that was not toggling will toggle every 0.5 seconds.

```
Kboard:
 bset 2,KBISC                ; Acknowledge to KBI
 clr PTAD
 lda change                  ;
 eor #1
 sta change
 bra Loop
```

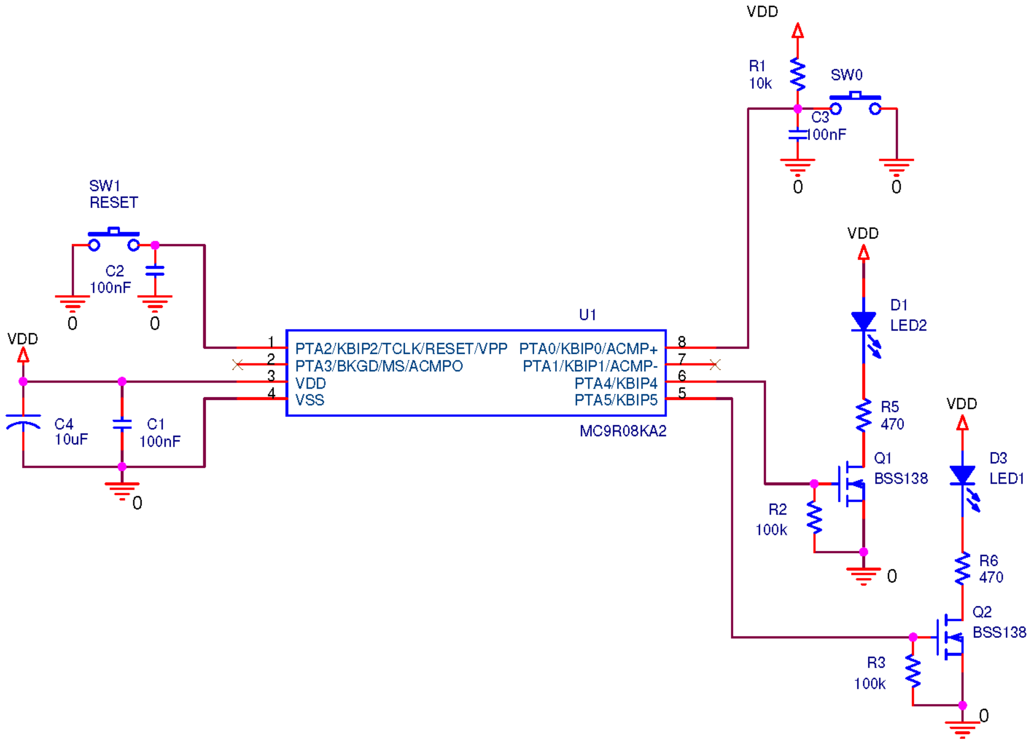
3 Hardware Implementation

This schematic shows the hardware used to exercise the code provided. For this example, the hardware implementation is fairly simple because the application itself uses only six pins.

Following six pins of the MCU are needed:

- Supply voltage pin
- Ground reference pin
- Reset pin
- Three available pins — two configured as outputs to toggle on-off two LEDs and one configured as an input for the keyboard interrupt

Figure 1. Hardware Implementation Interrupt Handling



Freescale Semiconductor
Users Guide

Using the Low-Power Modes for the RS08 Family Microcontrollers

by: Gabriel Sanchez Barba
RTAC Americas
México 2011

1 Overview

This is a quick reference for using the low-power modes on an HCRS08 microcontroller (MCU). Basic information about the functional description and configuration options is provided. The following example may be modified to suit your application. Refer to the data sheet for your device.

Table of Contents

Low-Power Stop Mode Quick Reference

SOPT	R	COPE	COPT	STOPE	0	0	0	BGDPE	RSTPE
	W								

System Options Register
COPE – COP watchdog enable
COPT – COP watchdog timeout
STOPE – stop mode enable
BKGDPE – background debug mode pin enable
RSTPE – reset pin enable

2 Low-Power Modes

The MC9RS08KA2 has two modes of low-power consumption. These modes offer flexibility for the user and may be used to lower the power consumption for many types of applications. These modes are the wait mode and the stop mode.

This table summarizes the behavior of the MCU in low-power modes

Mode	CPU	MTIM	ICS	ACMP	Regulator	I/O Pins	RTI
Wait	Standby	Optionally on	On	Optionally on	On	States held	Optionally On
Stop	Standby	Standby	Optionally On[1]	Optionally On[2]	Standby	States held	Optionally On

[1] ICS requires IREFSTEN=1 and LVDE and LVDSE must be set to allow operation in stop.

[2] If bandgap reference is required, the LVDE and LVDSE bits in the SPMSC1 must be set before entering stop to allow the 32-kHz clock source to run during stop.

2.1 Wait Mode Overview

Wait mode is entered whenever a WAIT instruction is executed. When executed, the CPU will enter a low-power state in which it is not clocked. The program counter (PC) is halted, but all the peripherals on the chip continue to work if they were enabled before the WAIT instruction was executed. The state of all internal registers and logic, as well as the RAM content, is maintained as all are I/O pin states. The CPU remains in this state until either a reset or interrupt occurs. If a reset occurs, the PC will fetch the address in the reset vector and begin to process instructions at that address. If an interrupt occurs, the MCU will exit wait mode and process the next instruction to be processed. If the MCU exits wait mode by an interrupt, then it is the user program’s responsibility to check the source of the interrupt, and then take whichever actions that are necessary.

2.2 Stop Mode Overview

Stop mode is entered whenever a STOP instruction is executed if the STOPE bit in the system options register is set. If the STOPE bit is not set, then an illegal opcode reset is forced. In stop mode, the voltage regulator is put in standby, and all internal clocks to the CPU and the modules are halted. The ICS is turned off when the IREFSTEN bit is cleared. The state of all internal registers and logic, as well as the RAM content is maintained as are all I/O pin states.

Exit from stop mode is done after either a reset or interrupt occurs. If a reset occurs, the PC will fetch the address in the reset vector and begin to process instructions at that address. If an interrupt occurs, the MCU will increment the PC, which halted at the location where the STOP instruction was executed, and the next instruction will be fetched and executed accordingly. It is the user program’s responsibility to check the source of the interrupt, and then take whichever actions are necessary.

3 Code Example and Explanation

In this application, the MCU switches from one low-power mode to another after a small delay. To trigger the exit from one mode and let the MCU enter the other, one of the KBI pins is used through a pushbutton. There is a LED that lights while the MCU is running, and dims when the MCU enters a low-power mode to let the user know whether the MCU is running or in a low-power mode. Every time the pushbutton is pressed, the MCU will exit low-power mode, delay, and re-enter a low-power mode.

The code initializes the MCU first. Here, we disable the COP, enable stop modes, the BKGD pin, and the reset pin. Then the hardware is setup; PTAD4 is setup as an output and turns on the LED, and the KBI1 pin is enabled.

```
MOV     #HIGH_6_13(SOPT), PAGESEL
MOV     #$23, MAP_ADDR_6(SOPT)              ; Disables COP, enables BKGD,RESET & STOP
MOV     #$10,PTADD                          ; PTA4 as Output
MOV     #$02,KBIPE                          ; PTA1 as KBI
MOV     #$06,KBISC                          ; Clear any false interrupts and unmask KBI
```

Next, the MCU is set to run in FEI mode:

```
LDA     #$00
STA     ICSC1
LDA     #$C0
STA     ICSC2
```

The code then initializes the two variables used in the delay:

```
MOV     #0, COUNTER2
MOV     #0, COUNTER1
```

The code then goes in and does a delay, which will turn off the LED when done, and finally enters wait mode.

```
mainLoop:
        JSR Delay
        WAIT
```

When the pushbutton is pressed, the MCU exits wait mode, lights the LED, and clears the KBI flag:

```
LDA     PTAD                                ; Toggle LED
EOR     #$10
STA     PTAD
BSET    KBISC_KBACK,KBISC                   ; Clear KBI interrupt
```

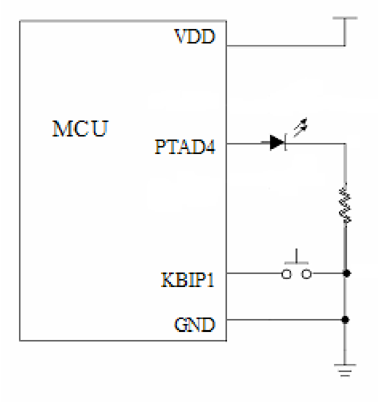
Then the code enters the delay routine, which dims the LED, and then enters stop mode:

```
JSR    Delay
STOP
```

When the pushbutton is pressed, the MCU exits wait mode and lights the LED, clears the KBI flag, and branches to the start:

```
LDA     PTAD                                ; Toggle LED
EOR     #$10
STA     PTAD
BSET    KBISC_KBACK,KBISC                   ; Clear KBI interrupt
BRA     mainLoop
```

4 Hardware Configuration



Freescale Semiconductor
Users Guide

Nesting Subroutines in the RS08 Microcontrollers

by: José Ruiz Juarez
RTAC Americas
México 2011

Table of Contents

1 Overview

This is a quick reference for using nested subroutines on an MC9RS08KA2 microcontroller.

The MC9RS08KA2 does not have stack support, but it does have a single level of subroutine that is implemented by using a shadow program counter. Upon calling a subroutine, the PC is saved in the shadow program counter before the jump to the subroutine is made. On returning from a subroutine, the program counter is loaded from the saved return address in the shadow program counter.

When calling a subroutine from another subroutine, the present program counter is stored in the shadow program counter but the previous PC that was stored on the shadow program counter is lost, which means that the CPU cannot go back to the main program.

This reference shows how subroutine nesting can be implemented in software by accessing the shadow program counter through the new instructions added to instruction set.

2 Code Example and Explanation

In this application, the implementation of nested subroutines will be demonstrated by turning on and off three LEDs. Each LED is turned on in a different subroutine and the main program turns all LEDs off. The single level subroutine is used to make the direct jump to the subroutine. Before it makes the jump, the shadow program counter must be backed up in RAM. The program counter is restored when the program returns from the subroutine. These procedures must be implemented in software. This application uses two macros to do that procedure.

Please refer to the source code for more details

1. Initialization. Configure PTA3, PTA4 and PTA5 as outputs

```
InitConfig:                                 ;CONFIGURES SYSTEM CONTROL

IFNE  MODE
       mov #HIGH_6_13(SOPT), PAGESEL
       mov #$01, MAP_ADDR_6(SOPT)           ; Disables COP and enables RESET (PTA2) pin
       mov #$34, PTADD                      ; PTA4(LED2),PTA5(LED1), PTA1 (LED3) as outputs
ELSE
       mov #HIGH_6_13(SOPT), PAGESEL
       mov #$03, MAP_ADDR_6(SOPT)           ; Disables COP, enables BKGD (PTA3) and RESET
(PTA2) pins
       mov #$30, PTADD                      ; PTA4(LED2),PTA5(LED1), PTA1 (LED3) as outputs
ENDIF
                                            ; configure PORT A
       clr PTAD                             ; Clears PTA
       rts
```

2. Call the LED1 subroutine in an infinite loop and after that, clear all ports.

```
_Startup:
       bsr InitConfig
loop:  clr PTAD
       jsr sal1
       jsr led1
       jsr sal1
       bra loop
```

3. Macro declarations. There are two macros. ENTRY_SUB is for backing up the shadow program counter into RAM. EXIT_SUB is to restore the shadow program counter from the RAM.
   — ENTRY_SUB — SHA instruction swaps the high byte of the shadow program counter with the accumulator. Then, STA stores the accumulator in a RAM location. Then it swaps the high byte of the shadow program counter with the accumulator. Now, the low byte of the shadow program counter must be backed up. To do this, the SLA instruction is used to swap the low byte of the shadow program counter with the accumulator. Then the accumulator is stored into the next available RAM, followed by a swap of the low byte of the shadow program counter with the accumulator.

```
ENTRY_SUB: MACRO                        ; Macro for "stacking" SPC
          SHA
          STA pcBuffer + 2*(\1)
          SHA
          SLA
          STA pcBUFFER + 2*(\1) +1
          SLA
ENDM
```

— EXIT_SUB — This subroutine does the exact opposite of what happened in the ENTRY_SUB routine. Here, the high byte of shadow program counter is swapped with the accumulator. Then the accumulator loads what is in RAM. This value is then swapped into the shadow program counter with an SHA. Next, the low byte of the shadow program counter is swapped with the accumulator, and the accumulator is loaded with the next available RAM location. This value is then swapped into the shadow program counter, and with that, the original values are restored.

```
EXIT_SUB: MACRO                         ; Macro for restore SPC
          SHA
          LDA pcBuffer + 2*(\1)
          SHA
          SLA
          LDA pcBUFFER + 2*(\1) +1
          SLA
ENDM
```

4. Subroutines. The three subroutines turn on a single LED and call a nested subroutine; LED3 is the last level subroutine of this application.

```
led1:  bset 5,PTAD
       ENTRY_SUB 0
       jsr sal1                    ;Call for a nested subroutine
       EXIT_SUB  0

       ENTRY_SUB 0
       jsr led2                    ;Call for a nested subroutine
       EXIT_SUB  0
       rts

led2:  bset 4,PTAD
       ENTRY_SUB 1
       jsr sal1                    ;Call for a nested subroutine
       EXIT_SUB  1

       ENTRY_SUB  1
       jsr led3                    ;Call for a nested subroutine
       EXIT_SUB   1
       rts

led3:  bset 3,PTAD
       ENTRY_SUB  2
       jsr sal1                    ;Call for a nested subroutine
       EXIT_SUB   2
       rts

sal1:                              ;delay by software
sal2:  dbnz COUNTER, sal2
       dbnz COUNTER2,sal1
       rts
```

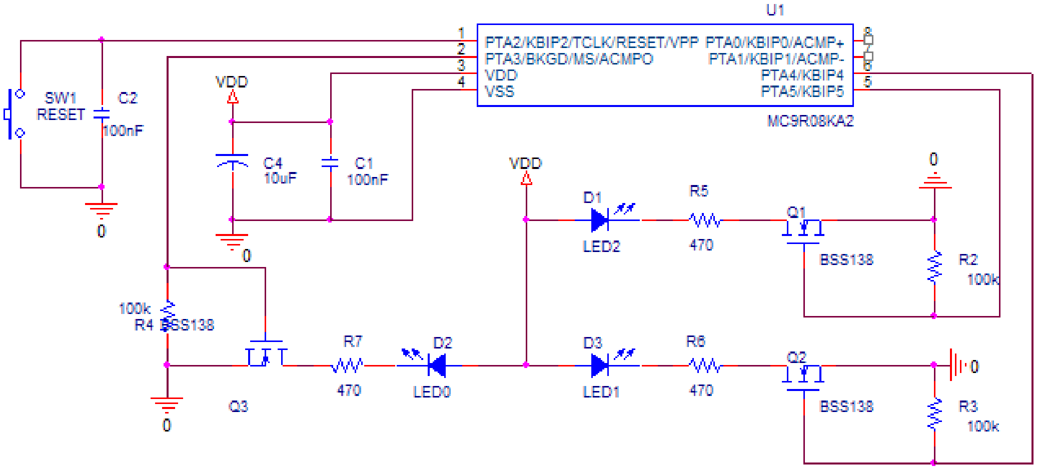
3 Hardware Implementation

This schematic show the hardware used to exercise the code provided. For this example the hardware implementation is fairly simple, because the application itself uses only six of the pins available in the MCU.

The six pins of the MCU that will be needed are:

- Supply voltage pin
- Ground reference pin
- Reset pin
- Three available pins configured as outputs to turn on-off three LEDs.

Figure 1. Hardware Implementation for nested subroutines



Freescale Semiconductor
Users Guide

Implementing an Analog-to-Digital Converter (ADC) on the MC9RS08KA2

by: Oscar Luna González, Alan Led Collins Rivera, and José Ruiz Juárez
RTAC Americas
México 2011

Table of Contents

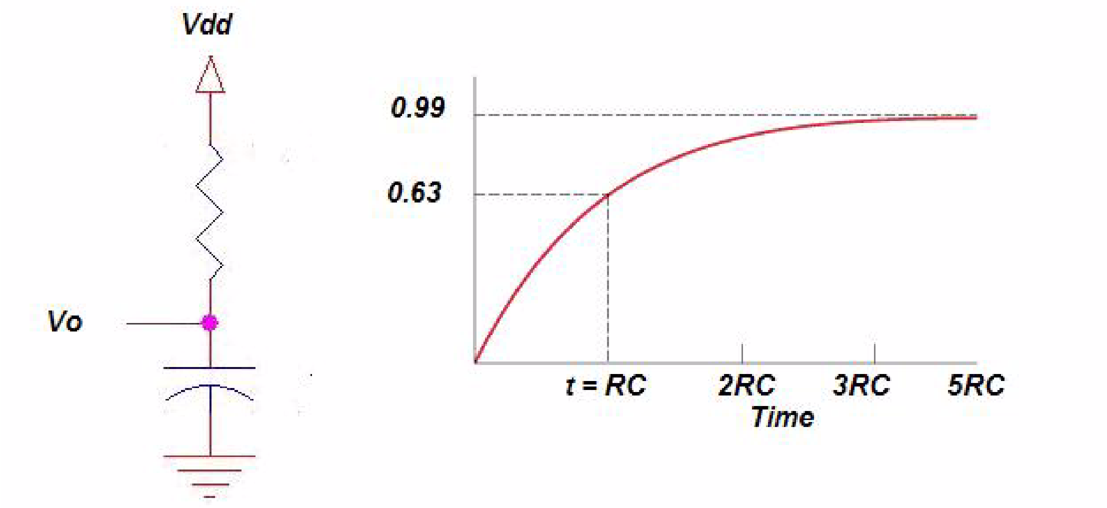
1 Overview

This is a quick reference for implementing an analog-to-digital converter using the modulo timer (MTIM) and analog comparator (ACMP) modules on an MC9RS08KA2 microcontroller (MCU).

Most embedded controller designs require a voltage measurement from sensors. The low-cost MC9RS08KA2 MCU provides an analog comparator with low powered modes, so it is the solution for this issue.

2 Theory

To implement an ADC with the voltage comparator, it is necessary to match an unknown voltage (V_{In}) with a voltage (V_{Out}) controlled by the MC9RS08KA2. This voltage can be easily generated by implementing a low-pass filter (RC) that will return a known voltage depending on the time that the capacitor charges (duty cycle).



The voltage response of the capacitor is not linear because of its behavior at charging:

$$V_o = V_{dd}\left(1 - e^{-\frac{t}{RC}}\right)$$

Because of this issue, it is necessary to create a lookup table with the matching voltage for its specific time.

After V_{Out} has reached the value of V_{In}, the count taken from the timer register is compared with the values table. This table contains the exact voltage that matches the time when V_{Out} reached the V_{In} value. Each count of the timer must be equal to the desired ADC resolution.

3 Calculating Table Values

The values of the table will be calculated based on the RC circuit establishment time. This establishment time is defined by the following formula:

$$\begin{gathered}\text{Establishment time} = 5\ \text{t} \\ \text{t} = \text{RC} \\ \text{Establishment time} = 5\ \text{RC}\end{gathered} \qquad \text{Eqn. 1}$$

The overflow time (t_{OF}) of the timer must be less-than or equal-to the establishment time of the RC circuit, so each count of the timer is defined by:

$$\text{Count time} = \frac{\text{MTIM overflow time}}{256} \qquad \text{Eqn. 2}$$

The converted value in embedded 8-bit ADC format is in the 0 – 255 range. The next formula shows how to calculate each value of the table.

$$Value = \frac{256V_o}{V_{dd}} \qquad \text{Eqn. 3}$$

Where V_{Out} is the calculated voltage on the capacitor in each count of the timer.

The next table contains the formulas that need to get the values of the table data.

Timer Count	Time	Voltage Value	ADC Value
0–255	$\frac{MTIM\ OF\ time}{256} xTimer\ count$	$V_o = V_{dd}(1 - e^{-\frac{t}{RC}})$	$\frac{256V_o}{V_{dd}}$

Note: OF means overflow

For this application, a specific RC value has been chosen to generate a capacitor charging time of 5 ms. These RC values are:

R = 10 k

C = 100 nF (ceramic capacitor)

3.1 Example Calculating an ADC Value

To start this example, a V_{In} value is proposed with a voltage of 2.28 V.

After the 2.28 V has been detected by the ACMP module,

Counter = 65 (MTIM Counter), the value of the 65th position in the ADC table is:

$$count = 65$$

$$MTIM\ \ OverFlow\ time = 5RC \approx 5ms$$

$$Time = \frac{.005}{256} x65 = 1.26\ ms$$

$$V_o = 3.3(1 - e^{-\frac{1.26\ m}{(10k)(100n)}}) = 2.28\ v$$

So, the ADC value will be:

$$ADC\ Value = \frac{(256)(2.28)}{3.3} = 177$$

4 Code Example and Explanation

The ACMPSC (analog comparator status and control register) must be initialized. Enabling the ACMP module to set the PTA1 as external ACMP+ terminal, enable interrupt and comparator falling edge type are the settings that are necessary to set for the ADC operation.

```
ACMP_Conf:
      MOV #ACMP_ENABLE,ACMPSC     ; ACMP Enabled, ACMP+ pin active, Interrupt enabled, Rising
                                            ; edges detections
      rts
```

The MTIM module is configured with the internal clock reference, prescaler divide by 128 and the mode as free running. See code below for more details:

```
MTIM_ADC_Init:
      mov #MTIM_128_DIV,MTIMCLK   ; Select bus clock as reference, Set prescaler with 128
      mov #FREE_RUN,MTIMMOD       ; Configure Timer as free running
      mov #MTIM_STOP_RESET,MTIMSC
      rts
```

1. Discharge Capacitor — completely discharges the capacitor. Is invoked before an ADC conversion has begun to avoid any unexpected voltage value that the capacitor could have. After this function is invoked, a delay loop is implemented to ensure the capacitor discharge.

```
Discharge_Cap:
      bset  1,PTADD                     ; Configure PTA1 as Output
      bclr  1,PTAD                      ; Start Capacitor discharging
      lda   #$FE                        ; Set delay time
waste_time:
      dbnza waste_time                  ; wait until Delay = 0
      rts
```

2. Read value from ACMP — resets the timer counter, clears the ACMP interrupt flag, disables the ACMP, and calls to a routine to search in the data table.

```
ReadVal:
      mov #MTIM_STOP_RESET,MTIMSC       ; Stop and reset counter
      mov #ACMP_DISABLED, ACMPSC        ; ACMP Disabled, Clear Interrupt flag
      ENTRY_SUB 0
      jsr tabla                         ; Search on table
      EXIT_SUB  0
      rts
```

3. Search in table function — To access the data table (allocated in Flash 0x3E00) it is necessary to use the page register. Four pages are needed because table data needs to store a 256 value and each page can hold 64 bytes. To accomplish this, an algorithm has been generated to calculate the page and the offset page where the data will be.

   To calculate the page, it is necessary to divide the start direction of the table between 64 and add the counter value divided by 64. For this case, the first 64 values of the table are stored in the F8 page, the second one in the F9, third one in the FA, and the last 64 values are in the FB page.

```
tabla:
      lda CounterValue
      clc                               ; clear Carry
      rola                              ; Getting 2 MSB
      rola                              ;
      rola
      add #(Table_Data>>6)              ; Page Calculating
      mov #PAGESEL,Temp_Page            ; Backup actual page
      sta PAGESEL                       ; Page Change
```

   After the page has been validated, the next step is determining in which of the 64 bytes the conversion result is located. The six least significant bits of the timer counter must be applied with a logical AND operation and add a 0xC0 value (start of the paging window). The result of this operation is the physical address that contains the result of the conversion. To access this location, it is necessary to store the address in the X register and go to the accumulator with lda, x instruction (D[X]).

```
lda CounterValue
      and #$3F                              ; Extract 6 LSB
      add #$C0                              ; Index to paging window
      tax
      lda ,x                                ; Load table result
      sta ConvertedValue                    ; Store result
      mov #Temp_Page, PAGESEL               ; Back Page
      rts
```

For example, if the counter value has a count of 65 (0x41), the page register will be loaded with an F9 value (second data page) and the X register will load a 193 (0xC1). That is the second value of the second page.

4. Main Function — The first step of the main function is to initialize the microcontroller to run at 8 MHz and trim the internal oscillator. Then configure the timer by calling MTIM_ADC_Init function. Next, discharge the capacitor thought Discharge_Cap function. After that, the conversion starts. ACMP_Conf is called and the capacitor charging starts, immediately initializing the timer. The charge continues until the voltage of capacitor is equal to V_{In}. When the non-inverting voltage reaches the inverting voltage ($V_{Out} = V_{In}$), the ACMP interrupt will be triggered leaving the wait mode.

   After the ACMP interrupt has been triggered, the counter value of the timer is stored immediately in a variable. Finally, the code must branch into the ReadVal subroutine (if the interrupt was caused by ACMP).

```
_Startup:
      bsr Init_mc
      bsr MTIM_ADC_Init                     ; Configure MITM
      bsr Discharge_Ca                      ; Discharge Capacitor
      bsr ACMP_Conf                         ; Configure ACMP+ and ACMP-
      mov #MTIM_ENABLE,MTIMSC               ; Timer Counter Enabled
mainLoop:
      wait                                  ; Wait for ACMP interrupt
      bset 1,MTIMSC
      lda MTIMCNT
      sta CounterValue                      ; store counter value
      mov #HIGH_6_13(SIP1), PAGESEL
      brset 3, MAP_ADDR_6(SIP1),ReadVal     ; branch if ACMP interrupt arrives
      bra mainLoop

Data table
ORG Table_Data
dc.b 0,5,10,14,19,23,28,32,36,40,44,48,52,56,60,63
dc.b 67,71,74,78,81,84,87,91,94,97,100,103,106,108,111,114
dc.b 117,119,122,124,127,129,132,134,136,139,141,143,145,147,149,151
dc.b 153,155,157,159,161,162,164,166,168,169,171,173,174,176,177,179
dc.b 180,182,183,184,186,187,188,190,191,192,193,194,196,197,198,199
dc.b 200,201,202,203,204,205,206,207,208,209,210,211,211,212,213,214
dc.b 215,215,216,217,218,218,219,220,221,221,222,222,223,224,224,225
dc.b 226,226,227,227,228,228,229,229,230,230,231,231,232,232,233,233
dc.b 234,234,234,235,235,236,236,236,237,237,237,238,238,238,239,239
dc.b 239,240,240,240,241,241,241,241,242,242,242,243,243,243,243,244
dc.b 244,244,244,244,245,245,245,245,245,246,246,246,246,246,247,247
dc.b 247,247,247,247,248,248,248,248,248,248,249,249,249,249,249,249
```

```
        dc.b 249,249,250,250,250,250,250,250,250,250,250,251,251,251,251,251
        dc.b 251,251,251,251,251,252,252,252,252,252,252,252,252,252,252,252
        dc.b 252,252,253,253,253,253,253,253,253,253,253,253,253,253,253,253
        dc.b 253,253,253,253,254,254,254,254,254,254,254,254,254,254,254,254
```

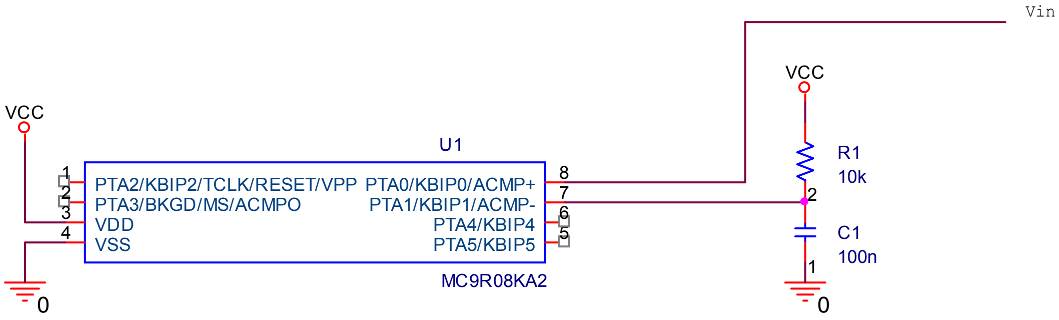
5 Hardware Implementation

This schematic shows the hardware used to evaluate the code. For this example, the hardware implementation is fairly simple because the application itself uses only four of the pins available on the MCU. Only two external components are needed to implement this 8-bit ADC, giving a really low-cost ADC implementation.

Follwoing four pins of the MCU are needed:

- Supply voltage pin
- Ground reference pin
- ACMP+ (PTA0)
- ACMP– (PTA1)

Figure 1. Hardware Implementation ADC



Freescale Semiconductor
Users Guide

Serial Communication Interface Implementation using MTIM module for the MC9RS08KA2 Microcontroller

by: José Ruiz Juárez
Oscar Luna González
Alan Led Collins Rivera
Miguel Agnesi Meléndez
México 2011

Table of Contents

1 Overview

This is a quick reference to implement a basic serial communication interface (SCI) using the timer (MTIM) module of the MC9RS08KA2 microcontroller.

Some embedded designs require communication because of a peripheral or with another application.

This document describes how to implement a low-cost serial communication interface transmitter using one pin and the MTIM module of MC9RS08KA2.

2 Theory — Asynchronous Serial Communication Interface

This communication protocol uses two pins of data, one for transmitting (TxD) and one for receiving (RxD) data. There is no need for a clock pin, because of the validation at the start and stop bits of the communication.

In first instance, the signal is on idle state (high). This means that no data transfer is taking place. Then, if a falling edge is detected, a time equal to the baud rate chosen (9600 bps) is needed for validation of data. So, if there is a change in the signal before this time has come, a false start bit error occurred (Figure 1). This means that the data is just noise and there is no need to save it.

Otherwise, the start bit validates the communication and the next bit that transfers will be the LSB of the data byte. Every time the baud rate time elapses, a data bit will be taken from the data pin until the complete byte is received. Then, the data pin must be in high state for one baud rate time. If the data changes before the time of validation, a false stop bit has occurred and this means that the data byte taken is not correct. If the data does not change, the stop bit has validated the byte received and the data pin enters in idle state for a new communication.

Figure 2 summarizes the protocol.

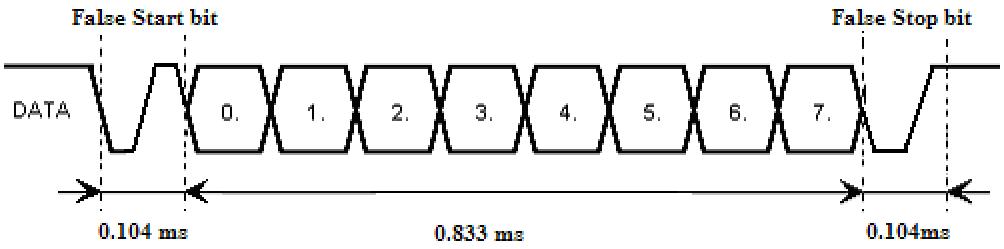


Figure 1. Example of False Start and Stop Bits

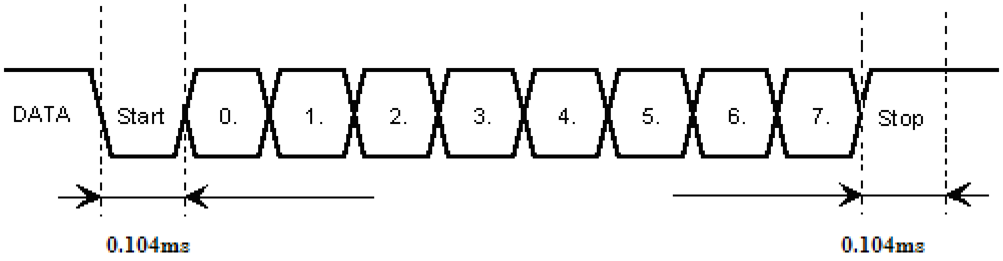
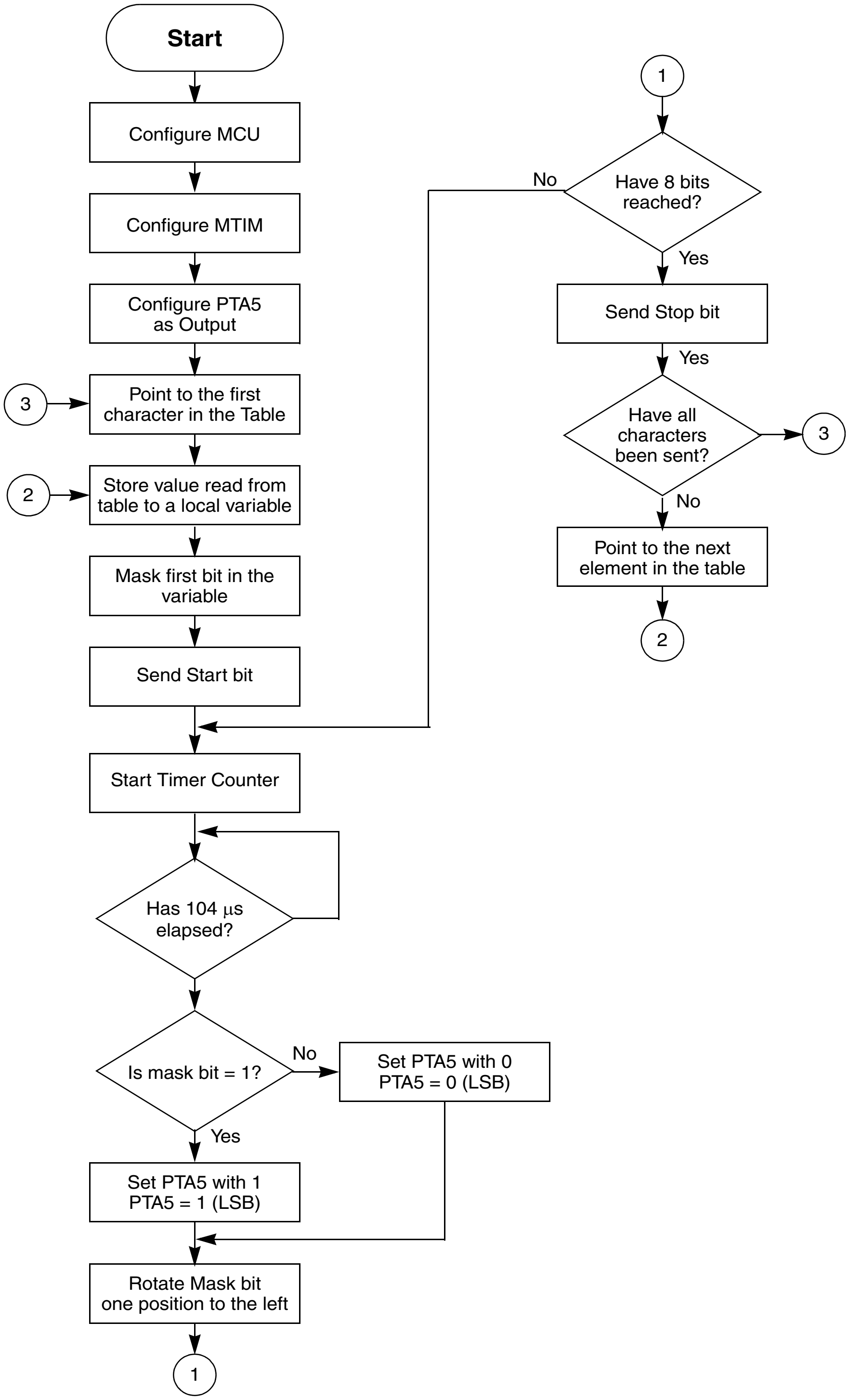


Figure 2. Correct Serial Communication

3 Flow Chart

Figure 3. Implementation a serial communication interface transmitter using one pin and MTIM module


Start
Configure MCU
Configure MTIM
Configure PTA5 as Output
3
Point to the first character in the Table
2
Store value read from table to a local variable
Mask first bit in the variable
Send Start bit
Start Timer Counter
Has 104 µs elapsed?
Is mask bit = 1?
No
Set PTA5 with 0 PTA5 = 0 (LSB)
Yes
Set PTA5 with 1 PTA5 = 1 (LSB)
Rotate Mask bit one position to the left
1
1
No
Have 8 bits reached?
Yes
Send Stop bit
Yes
Have all characters been sent?
3
No
Point to the next element in the table
2

4 Code Example and Explanation

The advantage of using the MCRS08KA2 microcontroller for implementing a serial communication interface (SCI) is that, it is extremely low cost. The MCU may communicate with other microcontrollers or the PC, depending on the requirements of the application.

This example implements only the transferring data communication running at 9600 bits per second, which means that every validated bit length is 104 μs. Serial communication interface protocol allows a maximum bit error of 4%, so the MCU must generate a precise signal every 104 μs. Therefore the MCU must be trimmed to generate the fewest errors in the SCI signal generation using the MTIM module. After these considerations, a free pin in the MCU is needed to transmit data.

This project has benn developed using CodeWarrior v5.1 and the DEMO9RS08KA2 board from Softec.

This example sends the legend FREESCALE through PTA5 using the RS232 protocol. The main function calls three initialization functions. The first function loads from the table data the value that will be sent with the page register and calls for the Send_SCI function to send 1 byte each cycle. The operation repeats Letter_Number times. This variable is the number of characters that contain the data table.

```
_Startup:
      jsr Init_Conf
      jsr Init_MTIM
      jsr Init_PTA
mainloop:
      mov #Letter_Number,Letter_Counter
      lda #$C0
      tax                              ;Load in X the first position of table
cicle:
      mov #$F8,PAGESEL                 ;Change Page to Table data
      lda ,x                           ;Load value of allocation
      sta Byte_to_Send                 ;Store value in Variable to Send
      jsr Send_SCI
      inc x                            ;Increment to the next position of table
      dbnz Letter_Counter,cicle
      BRA    mainloop
```

To generate a 9600 baud rate, the time of each bit on the package will be generated every 104 μs. Running at frequency of 8 MHz, it is necessary to establish the MTIM prescaler value with 32, which gives a result of 4 μs for each count. So, to get the 104 μs value, it is necessary to generate 26 counts.

```
Init_MTIM:
      mov #$70, MTIMSC                 ; Enables interrupt, stops and resets timer
                                       ; counter
      mov #$1A, MTIMMOD                ; MTIM modulo with 26 counts before interrupt.
      mov #$05, MTIMCLK                ; Selects internal clock as reference bus and 32
                                       ; preescaler
      rts
```

PTA5 pin is configured as an output because this pin will serve as the transmitter pin.

```
Init_PTA:
      bset 5,PTADD                      ;PTA5 as output
      bset 5,PTAD                       ;Set PTA5
      rts
```

Send_SCI function initializes the counter variable with eight because the package contains 8-bit data. The roll_bit serves as a mask bit. This bit will shift left eight times (one each cycle) to make a logical and mask with the byte to send. If the result is zero, the bit to send will be zero and if the result is non-zero, the bit to send will be 1. Each bit waits for a timer overflow (104 µs) to be sent. When the eight bits have been sent, PTA is set to 1 for two more timer overflows: the first is the STOP bit and the second one is the gap time to do the next START bit.

```
Send_SCI:
      mov #08,counter                   ; variable for control
      mov #01,roll_bit                  ; Mask Variable
      bclr 4,MTIMSC                     ; run Timer
      bclr 5,PTAD                       ; Start Bit
      mov #HIGH_6_13(SIP1), PAGESEL
wait2:
      brset 2, MAP_ADDR_6(SIP1),data
      bra wait2
data: ; Start to Send a data Bit
      lda MTIMSC                        ; Clear overflow interrupt flag
      mov #$60,MTIMSC                   ; Reset MTIM Counter, Clear  overflow flag
      lda Byte_to_Send
      and roll_bit                      ; Mask with data
      beq value_0                       ; If bit=0 call Value_0
      bra value_1                       ; If bit=1 call Value_1
temp:
      lda roll_bit
      asla                              ; Shift left Mask Bit
      sta roll_bit
      dbnz counter,wait2                ; loop until 8 data bits
wait3:
      brset 2, MAP_ADDR_6(SIP1),data2
      bra wait3
data2:
      lda MTIMSC
      mov #$60,MTIMSC                   ; Reset MTIM Counter, Clear  overflow flag
      bset 5,PTAD                       ; Stop Bit
wait4:
      brset 2, MAP_ADDR_6(SIP1),data3
      bra wait4
data3:
      lda MTIMSC
      mov #$60,MTIMSC                   ; Reset MTIM Counter, Clear  overflow flag
wait5:
      brset 2, MAP_ADDR_6(SIP1),data4
      bra wait5
data4:
      lda MTIMSC
      mov #$60,MTIMSC                   ; Reset MTIM Counter, Clear  overflow flag
      rts
```

```
value_0:
       bclr 5,PTAD                          ; send 0 to data port
       bra  temp
value_1:
       bset 5,PTAD                          ; send 1 to data port
       bra temp
```

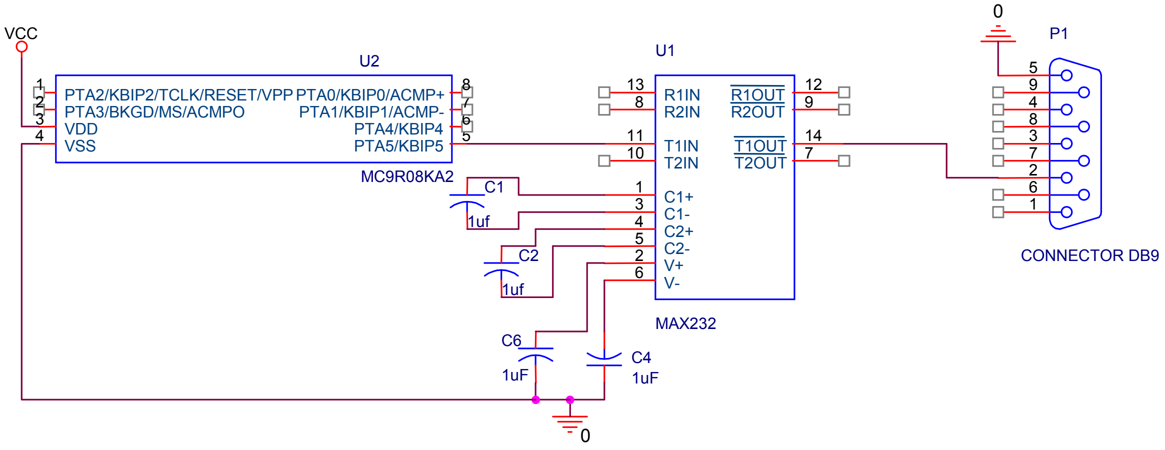
5 Hardware Implementation

This schematic shows the hardware used to evaluate the code provided. For this example, the hardware implementation is fairly simple because the application itself uses only three of the pins available on the MCU.

Only three pins of the MCU will be needed:

- Supply voltage pin
- Ground reference pin
- One GPIO pin

Figure 4. SCI transmitter implementation



How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: RS08QRUG
Rev. 2
11/2011

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.