Freescale Semiconductor
Application Note

Document Number: AN4032
Rev. 0, 02/2010

Enabling a Single Global Interrupt Vector on the RS08 Platform

by: Li Meng
Microcontroller Solution Group
China

1 Introduction

Based on a need for better interrupt support for the RS08 family, a single interrupt vector is added. Adding a global single interrupt vector is optimized for an entry-level application for lower cost and higher performance. This application note describes how to use it based on the MCRS08KB12.

Contents

2 RS08 Core

The RS08 platform is developed for extremely low cost applications. Its hardware size is optimized and the overall system cost is reduced. The interrupt mechanism in the original RS08 is not used to interrupt the normal flow of instructions, but to wake the RS08 from wait and stop modes. In run mode, interrupt events must be polled by the CPU. The original RS08 architecture does not include an interrupt controller with a vector table lookup mechanism as used on the HC08 and HCS08 devices. The CPU must repeatedly check the status of the interrupt flag until the desired condition is indicated, therefore wasting time and increasing the CPU load. The global single interrupt vector is added, the advantage is that the CPU can enter sleep mode to reduce power consumption and wakes when interrupts occur. Please refer to the data sheet for the interrupts that wake the CPU from various sleep modes. Microcontrollers can also use interrupts to prioritize the tasks and to ensure that certain peripheral modules are quickly serviced.

The MCRS08KB12 is the first silicon with a single global interrupt vector in the RS08 platform. There are many interrupt sources, including analog comparator, keyboard interrupt, modulo timer, real-time clock, ADC, IIC, TPM, and SCI. They all share one interrupt vector and can wake the MCU form wait mode. The KBI, ACMP, and RTI can take the MCU out of stop when the associated interrupt is enabled.

3 Interrupt

An interrupt is usually defined as an event that alerts the sequence of instructions executed by a processor. The RS08 supports a single interrupt vector in the same way as the RS08 reset vector is supported. A single global interrupt vector means that all the modules share the same interrupt vector. When the interrupt occurs, the user application polls the system interrupt pending registers (SIPx) to determine if an interrupt was pending.

3.1 Interrupt Operation

Most (peripheral) modules have status flags, or interrupt flags that can trigger an interrupt. An interrupt is when a module signals to the CPU that the flow of the program code execution should be interrupted and a specific interrupt service routine (ISR) program code must be executed. When IMASK is cleared, a pending interrupt (any flag set in SIP1 or SIP2) forces an interrupt request to the CPU and the interrupt is serviced immediately after completion of the current instruction. When IMASK is set, interrupts must be polled as in the original RS08 architecture. When the interrupt is triggered, the CPU executes code from the internal memory with execution beginning at the address $3FF7. A jump instruction (opcode $BC) at $3FF7 with operand located at $3FF8–$3FF9 must be programmed into the user application for correct interrupt operation. The operand defines the location of the user ISR after a pending interrupt is allowed to interrupt the CPU. The IMASK bit is set when the CPU vectors begin to interrupt servicing. Within the ISR, you can poll SIP1 and SIP2 and service pending interrupts based on application priority. A number of status bits in the SIP1 and SIP2 registers are referred to as interrupt flags. These interrupt flags provide information about change of states in the module, for example, a byte that has been received on the SCI. Interrupt flags are set if the corresponding interrupt is enabled. This makes it possible to poll interrupt flags in the interrupt service routine. Polling can be preferred in some cases, but are often handled by an ISR which is executed when the event occurs. One advantage of ISR handling is that important modules can be given immediate attention by temporarily stopping less important tasks. Interrupts must be cleared manually, typically by writing a logic one to the flag.

After the ISR has been executed, you must follow an ISR exit sequence:

- To re-enable interrupts on return from the ISR, the last two instructions of the ISR must be a clear of the IMASK bit followed by a jump to IEA. There is a three-cycle delay between the writing to clear the IMASK and enable interrupts to allow the double jump (JMP IEA, and JMP [IRA]) to a normal program execution to occur before handling any pending interrupts.
- To continue to mask interrupts on return from the ISR, you must execute the double jump MPIEA, and JMP [IRA] to a normal program execution without writing to the IMASK.

The condition code register (CCR) contents at the time of the interrupt are also saved in the IRAH, the most-significant two bits, and are restored on a return from the ISR. No other registers are saved automatically during interrupt vectoring. If you determine that the ISR will corrupt the accumulator or index registers, then it is up to you to save the contents in RAM and restore them before a return from the ISR. The program counter (PC) of the instruction that would have been executed by the interrupt request is saved in the interrupt return address (IRA) register. In this document, the IRA refers to the concatenation of IRAH and IRAL (IRAH:IRAL). After the ISR code has completed, the address saved in IRA will be used as the return address to the program, executing prior to servicing the interrupt. A jump instruction is hardware encoded in the user memory map at the address immediately preceding the IRA register and is referred to as the interrupt exit address (IEA) register. A jump to IEA is used at the end of the ISR for returning to the program executing prior to servicing the interrupt.

3.2 Interrupt Operation in Wait Mode

Wait mode is entered by executing a WAIT instruction. Upon execution of the WAIT instruction, the CPU enters a low-power state that is not clocked. The program counter (PC) is halted at the position where the WAIT instruction is executed. The RS08 supports a wakeup from WAIT differently. This depends on the state of IMASK in SIP2. If IMASK = 1, interrupt vectoring is disabled. When an interrupt pending flag becomes set, the MCU exits wait mode and resumes processing at the next instruction. If IMASK = 0, interrupt vectoring is enabled. When an interrupt pending flag becomes set, the MCU exits wait mode. The address of the instruction following the WAIT is stored in IRA. The PC is loaded with 0x3FF7, and code execution begins.

3.3 Interrupt Operation in Stop Mode

Stop mode is entered upon execution of a STOP instruction when the STOPE bit in the SOPT1 is set. In stop mode, all internal clocks to the CPU and the modules are halted. If the STOPE bit is not set when the CPU executes a STOP instruction, the MCU will not enter stop mode and an illegal opcode reset is forced. Stop is exited by asserting RESET or any asynchronous interrupt that is enabled. If stop is exited by asserting the RESET pin, the MCU will be reset. The program execution starts at location $3FFD. If exited by means of an asynchronous interrupt, the sequence varies depending on the state of the IMASK in SIP2. If IMASK = 1, the interrupt vectoring is disabled. When an asynchronous interrupt request occurs, the MCU exits stop mode and resumes processing the next instruction. If IMASK = 0, interrupt vectoring is enabled. When an interrupt pending flag becomes set, the MCU exits stop mode, the address of the instruction following the STOP is stored in IRA. The PC is loaded with 0x3FF7, and code execution begins.

4 Example Code

This code must be added in the application program to enable the interrupt vectoring mechanism. You can also use the macro of EnableInterrupts that has been defined in the file derivative.h.

```
SIP2_IMASK=0;                // enable interrupt
```

Enable the corresponding bit in the module, such as:

```
KBISC_KBMOD =0;          // Keyboard detects edges only
KBIPE = 0xff;            // enabled KBI pin
KBIES = 0x0;             // falling edge
KBISC_KBIE = 0x1;        // Enable KBI interrupt
```

The macro below is used to store and restore the data in the accumulator and index register. This has been added to the head file “derivative.h”., and can be generated automatically by creating the project.

```
#define ISR_WRAPPER(name) void ISR(void)
{
    asm(STA $50);       //store the content of A in the adress of $50
    asm(STX $51);       //store the content of Index register in the adress of $51
    asm(MOV $1F, $52)   //store the content of Pagesel register in the adress of $52
    asm(SHA);           // swap shadow PC High with A
    asm(STA $53);       // Sore the content of  shadow PC High in the adress of $53
    asm(SLA);           // swap shadow PC Low with A
    asm(STA $54);       // Sore the content of  shadow PC Low in the adress of $54
    asm(JSR name );     // jump into the user’s ISR
    asm(LDA $54);
    asm(SLA);           // Restore the content of the shadow PC Low
    asm(LDA $53);
    asm(SHA);           // Restore the content of the shadow PC High
    asm(MOV $52,$1F);   // Restore the content of the Pagesel register
    asm( LDX $51);      // Restore the content of the index register
    asm( LDA $50);      // Restore the content of the A
    asm(BCLR 7,  SIP2); // enable the interrupt
    asm( JMP  IEA);     // return form routine
}
```

The wrapper for the interrupt routine which is the user's interrupt handler name is the parameter:

ISR_WRAPPER(myISR) ;

The user’s interrupt handler name must be the same name as defined in the wrapper. You can lookup the status of the pending interrupt in SIP1 or SIP2. The sequence depends on the priority, so you can implement a prioritized interrupt mechanism in the software. Code can be deleted or added according to your situation.

```
void myISR(void)
{
/* Place Interrupt handler code here */
if (SIP1_KBI)
   {
    if((PTAD_PTAD2==0)&&(Key1_Flag==0))
     {
      Key1_Flag =1;
      PTBD_PTBD4 ^=1;
      Enable_Timer1();
     }
     if((PTAD_PTAD3==0)&&(Key2_Flag==0))
     {
      Key2_Flag =1;
      PTBD_PTBD5 ^=1;
      Enable_Timer2();
     }
     KBISC_KBACK = 0x1;
    }
    if (SIP2_TPMCH1)
    {
      TPMC1SC_CH1F=0;
      TPMC1SC = 0x00;
      TPMSC = 0x00;
    }
    if(SIP1_MTIM1)
    {
      MTIM1SC_TOF = 0;
      Time1Counter++;
      if(Time1Counter>=100)
      {
        Disable_Timer1();
        Key1_Flag =0;
        PTBD_PTBD4 ^=1;
        Time1Counter=0;
      }
     }
     if(SIP1_MTIM2)
     {
       MTIM2SC_TOF = 0;
       Time2Counter++ ;
       if(Time2Counter>=100)
        {
          Disable_Timer2();
          Key2_Flag =0;
          PTBD_PTBD5 ^=1;
          Time2Counter=0;
        }
     }
     if(SIP1_ADC)
     {
      ADCSC1_COCO =1;
     }
     if(SIP1_RTI)
     {
       SRTISC_RTIACK =1;
     }
     if(SIP1_IIC)
     {
       IICS_IICIF=1;
     }
```

```
      if(SIP1_LVD)
      {
        SPMSC1_LVDACK =1;
      }
      if(SIP1_ACMP)
      {
        ACMPSC_ACF = 1;
      }
      if(SIP2_TPMCH0)
      {
        TPMC0SC_CH0F=1;
      }
      if(SIP2_TPMCH1)
      {
        TPMC1SC_CH1F=1;
      }
      if(SIP2_TPM)
      {
        TPMSC_TOF =1;
      }
      if(SIP2_SCIE)
      {
        PTCD_PTCD1 ^=1;
    }
  }
```

5 Conclusion

A single global vector can easily be used. It can be used to handle anticipated and unanticipated events, to implement a prioritized interrupt mechanism in the software, helps reduce power consumption, and the CPU load.

How to Reach Us:

Home Page:
www.freescale.com

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

USA/Europe or Locations Not Listed:
Freescale Semiconductor
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 +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4032
Rev. 0, 02/2010