Lecture #13

Interrupts

18-348 Embedded System Engineering

Philip Koopman

Monday, 29-Feb-2016

© Copyright 2006-2016, Philip Koopman, All Rights Reserved

Carnegie Mellon

Example: Electronic Parking Brake

http://www.conti-online.com/generator/www/de/en/continentalteves/continentalteves/themes/products/electronic_brake_systems/parking_brake_1003_en.html

- **"Software" parking brake**
  - Button on dash is a MCU input
  - Lever in center is an MCU input
  - Allows system to "Do The Right Thing"
    - Avoid skidding or spinning
    - Brake car to stop from high speed

- **Possible EPB Functions:**
  - Normal parking brake function
  - "Drive-away" automatic release on hills
  - Emergency braking if primary brakes fail
  - Vehicle immobilizer (car security system)
- **Discussion questions:**

  Assume critical functionality is provided by software
  - What are the worst potential hazards?
  - What is a likely acceptable failure rate?
  - Who is responsible for ensuring safe operation within design flow?

Where Are We Now?

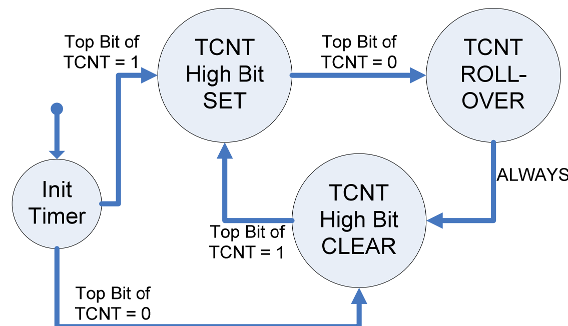
- **Where we've been:**
  - Time and counters – a bit more nitty-gritty
    - Keeping track of timer rollovers was painful, wasn't it?
    - Better approach – use interrupts!
- **Where we're going today:**
  - Interrupts
- **Where we're going next:**
  - Concurrency, scheduling
  - Analog and other I/O
  - Test #2

Preview

- **Really "getting" interrupts is an essential embedded system skill**
  - If I had to pick one job interview question to ask, this would be the topic
- **Hardware interrupts**
  - Asynchronous, hardware-triggered cousin to SWI
- **What happens in a HW interrupt?**
  - Trigger interrupt
  - Save state
  - Execute an Interrupt Service Routine
  - Acknowledge the interrupt (so it doesn't retrigger)
  - Resume execution of main program
- **Timer example**
  - Real time clock from last lecture – but done with interrupts
  - Complete example in both assembler and C
- **SWI as a system call**

Polling – What We've Done Until Now

- **Polling is periodically checking to see if something is ready**
  - Waiting for data ready or ready to transmit on UART
  - Watching timer for rollover
  - Watching for a button to be pressed
- **Polling can be a pain**
  - Need to continually check (difficult to weave checks into complex code)
  - If timing analysis is wrong, might poll too slowly
  - Can waste a lot of CPU time checking for very infrequent events



Interrupts To The Rescue

- **Big idea:**
  Wouldn't it be nice to be notified when something happens (interrupted) instead of having to check continually (polling)?
- **In daily life:**
  - Wristwatch:
    - Polling is checking watch every 5 minutes to see if class is over yet
    - Interrupt is having an alarm ring at end of class
  - Cell phone:
    - Polling is checking your phone to see if text message icon is displayed
    - Interrupt is having an audible alarm (or vibration) if text message is received
  - Making tea:
    - Polling is checking the kettle every minute to see if it is boiling
    - Interrupt is having a the tea kettle whistle

Remember SWI?

SWI

Software Interrupt

Operation:

(SP) – \$0002 ⇒ SP; $RTN_H : RTN_L$ ⇒ ($M_{(SP)} : M_{(SP+1)}$)
(SP) – \$0002 ⇒ SP; $Y_H : Y_L$ ⇒ ($M_{(SP)} : M_{(SP+1)}$)
(SP) – \$0002 ⇒ SP; $X_H : X_L$ ⇒ ($M_{(SP)} : M_{(SP+1)}$)
(SP) – \$0002 ⇒ SP; B : A ⇒ ($M_{(SP)} : M_{(SP+1)}$)
(SP) – \$0001 ⇒ SP; CCR ⇒ ($M_{(SP)}$)
1 ⇒ I
(SWI Vector) ⇒ PC

Subroutine call plus automatic push of Y, X, D, CCR

Jump to a predefined address

Description:

Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to Chapter 7 Exception Processing for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

SWI Is A "Software Interrupt"

- **Sort of like a subroutine call (JSR), but with some differences**
- **Interrupts flow of program control**
  - Jumps to a location specified by a "vector" instead of an address in the instruction
    - That makes it an Inherent operand ("INH") – the address is *not* in the instruction
    - We'll get back to the idea of a vector shortly
  - Always the same address for any SWI invocation, regardless of how many
- **Saves state**
  - Pushes the programmer visible register state on the stack
  - Including the condition code register
  - As long as SWI-processing routine doesn't mess with stack or memory, return from SWI leaves CPU in exactly the same state as before the SWI (see the RTI instruction)
- **RTI is like RTS**
  - BUT with differences to corresponding to SWI placement of stack items

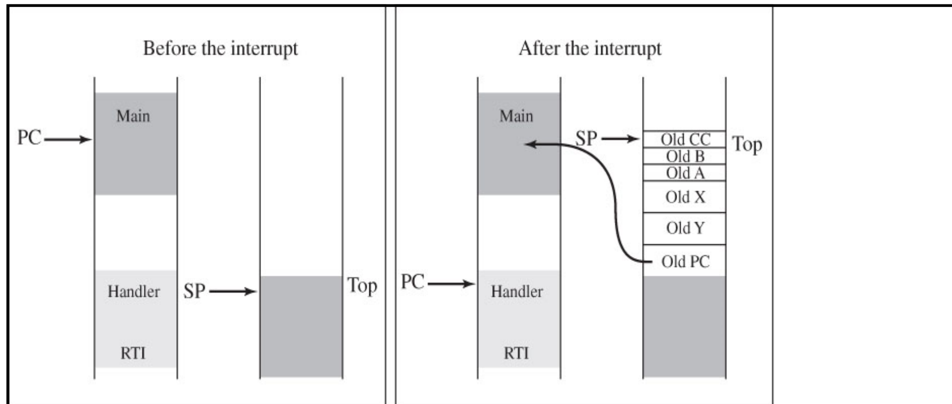


Figure 4.19
6812 stack before and after an interrupt.

[Valvano]

Table 7-2. Stacking Order on Entry to Interrupts

Memory Location	CPU Registers
SP + 7	$RTN_H : RTN_L$
SP + 5	$Y_H : Y_L$
SP + 3	$X_H : X_L$
SP + 1	B : A
SP	CCR

RTI — Return from Interrupt — RTI

Operation:

$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) - \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

Description:

Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

If another interrupt is pending when RTI has finished restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched. This operation is functionally identical to the same operation in the M68HC11, where registers actually are re-stacked, but is faster.

CCR Details:

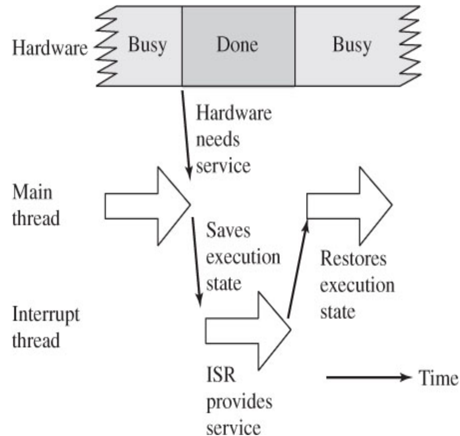
S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

• **Inverse operation of SWI – puts everything back on the stack**

What Do We Need Beyond An SWI?

- **Want to generate an interrupt based on *when*, not *where***
  - SWI has to be placed as an explicit instruction at a specific location … a synchronous interrupt – happens synchronized to program flow
  - What we want are interrupts that occur without an actual instruction… ***asynchronous interrupt*** – happens without regard to program flow

Figure 4.3
An interrupt causes the main thread to be suspended, and the interrupt thread is run.



[Valvano]

Simplified Execution Of A Hardware Interrupt

(More concrete example with details coming soon…)

- **Some piece of hardware generates an interrupt**
  - Not an SWI instruction – happens asynchronously with and independent from the program execution
  - So this means it can happen ***any time*** *(with exceptions we'll get to soon)*
  - You can think of this as hardware shoving an SWI into the instruction fetch queue … even though the SWI wasn't actually in memory
- **CPU executes an interrupt handling process**
  - That interrupt causes CPU to execute a virtual interrupt opcode (same effects of SWI, but without that instruction coming from memory)
  - CPU jumps to a particular handling routine via a vector
- **Interrupt handling software executes**
  - An Interrupt Sub-Routine **(ISR)** executes (subroutine to handle the interrupt)
  - When completed, the ISR executes an RTI instruction
    - This is a real RTI instruction, just like we saw with SWI
- **Normal program resumes operation**
  - CPU registers unchanged – program has no idea it was interrupted

What Can Generate An Interrupt?

- **General categories on any CPU – each has a different vector location**
  - Interrupt jumps to address at vector (e.g., SWI jumps to [$FFF6])
  - Various types of resets
  - Various types of illegal situations (e.g., undefined opcode executed)
  - Hardware signals from devices
  - SWI

Table 7-1. CPU12 Exception Vector Map

Vector Address	Source
$FFFE–$FFFF	System Reset
$FFFC–$FFFD	Clock Monitor Reset
$FFFA–$FFFB	COP Reset
$FFF8–$FFF9	Unimplemented Opcode Trap
$FFF6–$FFF7	Software Interrupt Instruction (SWI)
$FFF4–$FFF5	$\overline{XIRQ}$ Signal
$FFF2–$FFF3	$\overline{IRQ}$ Signal
$FF00–$FFF1	Device-Specific Interrupt Sources (HCS12)
$FFC0–$FFF1	Device-Specific Interrupt Sources (M68HC12)

0xFFDE, 0xFFDF	Standard timer overflow	I bit	TMSK2 (TOI)

More Specific Example – Time Of Day

- **Remember the time of day example? (statechart in an earlier slide)**
  - Needed to tightly loop monitoring the TCNT value, watching for zero crossing
  - Better way – use an interrupt

- **Recap of program at a high level:**
  - TCNT is the current timer value; assume bus clock divided by 8
  - Current_time is a uint32
    - Add <*fractional_value*> to 32-bit value every TCNT rollover
    - High 16 bits are current time in seconds
  - Algorithm for the old approach (polled version):

```
for(;;)
{ <wait for TCNT roll-over (TCNT changes from $FFFF to $0000)>
  timer_count += <fraction_value>;
  <display (timer_count>>16) as seconds on LCD>
}
```

New Time Of Day Approach Using Interrupts

- **Main program – can do anything we want**

```
for(;;)
{ <do anything else you want; doesn't matter how long>
  <display (timer_count>>16) as seconds on LCD>
}
```

  - But wait, what is changing the timer_count value? ➔ **The ISR Does It**

- **ISR – (Interrupt Service Routine) – keeps track of TCNT rollovers**
  - This is what is changing current_time – main loop only displays it!
  - Keeps a time of day clock updated
  - Executes *only* when TCNT rolls over
  - ISR:

```
timer_count += <fractional_value>;
<return from interrupt>
```

  - How do we know when to executed ISR?
    Ask Timer HW to generate an Interrupt!

- **Following slides are how to do this step by step…**

Timer Register Setup Info

- **TEN and PR[2:0] discussed in last lecture; TOI and TFLG2 are new**

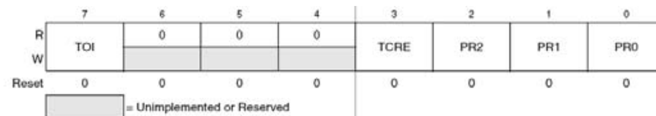


Figure 15-19. Timer System Control Register 2 (TSCR2)

Field	Description
7 TOI	**Timer Overflow Interrupt Enable** 0 Interrupt inhibited. 1 Hardware interrupt requested when TOF flag set.

PR2	PR1	PR0	Timer Clock
0	0	0	Bus Clock / 1
0	0	1	Bus Clock / 2
0	1	0	Bus Clock / 4
0	1	1	Bus Clock / 8
1	0	0	Bus Clock / 16
1	0	1	Bus Clock / 32
1	1	0	Bus Clock / 64
1	1	1	Bus Clock / 128

TOF

- **TOF set whenever TCNT rolls over**
  - If TOI is set, causes an interrupt
  - KEEPS causing an interrupt until it is cleared!!
  - Clear by writing a “1” bit to every flag to be cleared (i.e., write $80)
    - This is because TFLG1 has multiple bits and only want to clear some/one of them

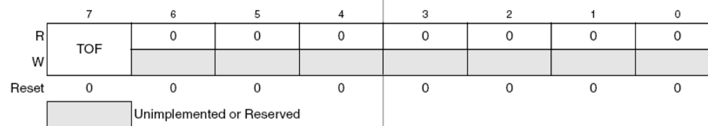


Figure 15-21. Main Timer Interrupt Flag 2 (TFLG2)

TFLG2 indicates when interrupt conditions have occurred. To clear a bit in the flag register, write the bit to one.

Read: Anytime

Write: Used in clearing mechanism (set bits cause corresponding bits to be cleared).

Any access to TCNT will clear TFLG2 register if the TFFCA bit in TSCR register is set.

Table 15-16. TRLG2 Field Descriptions

Field	Description
7 TOF	**Timer Overflow Flag** — Set when 16-bit free-running timer overflows from 0xFFFF to 0x0000. This bit is cleared automatically by a write to the TFLG2 register with bit 7 set. (See also TCRE control bit explanation.)

Timer Register Setup

- **TEN – Timer enable**
  - Bit 7 of TSCR1 -- set to enable timer

```
TSCR1 |= 0x80;
```

- **PR[2:0] – Timer prescale**
  - Bits 2..0 of TSCR2 set prescale – set to bus clock / 8
  - From last lecture, TCNT rollover every 0.167772 seconds

- **TOI**
  - Bit 7 of TSCR2 set to one – generates interrupt every time TCNT rolls
  - TCNT rollover is caught by the TOF – Timer Overflow Flag

```
// TSCR2[2:0] binary 011=bus clock/8
// TSCR2[7] TOI set to interrupt on TOF (TCNT rollover)
TSCR2 = (TSCR2&0x78)|0x03|0x80;
```

Timer Main Program

```
volatile  uint32  timer_val=0;

void main(void)
{
 // set TN = 1   Timer Enable   TSCR1 bit 7
 TSCR1 |= 0x80;

 // TSCR2[2:0] binary 011=bus clock/8
 // TSCR2[7] TOI set to interrupt on TOF (TCNT rollover)
 TSCR2 = (TSCR2&0x78)|0x03|0x80;

 EnableInterrupts;
 for(;;)
 { // code goes here to copy (timer_val>>16) to display
 } /* loop forever */
}
```


What About The Interrupt?

- **Need to initialize interrupt vector to point to ISR**
  - Usually done at load time, not run time
  - For us it is in flash memory, so must be done at load time
- **Need to import timer_val symbol from C code so it can be modified**
  - XREF timer_val   (means "this is a symbol defined in another module")
- **Need to clear TOF**
  - Or else hardware just re-triggers ISR forever
- **Need to add a fractional part to 32-bit integer time counter**
  - 8 MHz bus clock with divider of 8 and 64K rollover:
    (8*65536) / 8,000,000 = 0.065536 / (1/65536) scale factor = 4295 = $10C6
  - Add 4295 to 32-bit integer each rollover to get high 16 bits as integer seconds
- **Need to RTI to restore operation after ISR executes**
  - **Don't use "RTS"** because it doesn't restore registers and flags!

```
              XREF timer_val          ; import symbol from C code
MyCode:       SECTION
count_isr:      ; this is the ISR routine
              LDAA  #$80    ;  Clear TOF; top bit in TFLG2
              STAA  TFLG2   ;  This acknowledges the rollover intrpt
              ; 32-bit add $10C6 to increment fractional second
              LDAA  timer_val+3       ; byte-wise 32-bit add
              ADDA  #$C6
              STAA  timer_val+3
              LDAA  timer_val+2
              ADCA  #$10
              STAA  timer_val+2
              LDAA  timer_val+1
              ADCA  #$00
              STAA  timer_val+1
              LDAA  timer_val         ; why isn't this a loop here?
              ADCA  #$00              ; (what if infinite loop here?)
              STAA  timer_val
              RTI                ; return to interrupted program

                ORG $FFDE             ; set interrupt vector for timer

                DC.W count_isr
```

Hardware Interrupt Recap

- **Some piece of hardware generates an interrupt**
  - Happens asynchronously with and independent from the program execution
  - So this means it can happen ***any time*** *(with exceptions we'll get to soon)*
- **CPU executes an interrupt handling process**
  - That interrupt causes CPU to execute a virtual interrupt instruction
    - Happens between instructions, but anywhere in program
  - CPU jumps to a particular handling routine via a vector
    - Something has to set that vector to point to the ISR!
- **Interrupt handling software executes**
  - An Interrupt Sub-Routine **(ISR)** executes (subroutine to handle the interrupt)
    - Hardware saves registers
  - When completed, the ISR executes an RTI instruction
    - RTI restores the registers
- **Normal program resumes operation**
  - CPU registers unchanged – program has no idea it was interrupted

Leap Year Bug

Bad Code in the Microsoft Zune

```
1  year = 1980;
2  while (days > 365) {
3      if (IsLeapYear(year)) {
4          if (days > 366) {
5              days -= 366;
6              year++;
7          }
8      }
9      else {
10         days -= 365;
11         year++;
12     }
13 }
```

No else. Infinite loop when days == 366!

This ran in an ISR, which "bricked" the Zune until discharged.

Copyright © 2009 Netrino LLC. All rights reserved.

CW C Interrupt Syntax

- **You can handle interrupts in C/C++ as well!**
- **Syntax:**

```
void interrupt <n> <fn>(void)
{  }
```

```
void interrupt 0 ResetFunction(void) {
  /* reset handler */
}
```

- **<n> is the entry number in the interrupt vector list**
  - "2" is second entry, etc. -- it's the entry number, not byte address
  - Be careful, these numbers count *opposite* to address direction!

Vector Number	Vector Address	Vector Address Size
0	0xFFFE, 0xFFFF	2
1	0xFFFC, 0xFFFD	2
2 This is #2	0xFFFA, 0xFFFB	2
...	...	...
n	0xFFFF - (n*2)	2

Same Timer Example, In C

```
extern volatile unsigned long timer_val = 0;

void main(void)
{ // set TEN = 1   Timer Enable   TSCR1 bit 7
  TSCR1 = TSCR1 | 0x80;
 // TSCR2[2:0] binary 0=bus clock/8
 // TSCR2[7] TOI set to interrupt on TOF (TCNT rollover)
 TSCR2 = (TSCR2&0x78)|0x03|0x80;
 EnableInterrupts;
 for(;;) {
   // code goes here to copy (timer_val>>16) to display
 } /* loop forever */
}

void interrupt 16 timer_handler(void) //-(16*2)-2 = $FFDE for TOI
{ TFLG2 = 0x80;
  timer_val += 0x10C6;
}
```


Timer isn't the only thing that uses interrupts

- 128 interrupt vectors supported by course MCU
  - Most or all of them can be used in the same program!
  - Each vector gets its own ISR
  - Higher vectors get higher priority (pick one with highest address to service next)

Table 1-9. Interrupt Vector Locations

Vector Address	Interrupt Source	CCR Mask	Local Enable
0xFFFE, 0xFFFF	External reset, power on reset, or low voltage reset (see CRG flags register to determine reset source)	None	None
0xFFFC, 0xFFFD	Clock monitor fail reset	None	COPCTL (CME, FCME)
0xFFFA, 0xFFFB	COP failure reset	None	COP rate select
0xFFF8, 0xFFF9	Unimplemented instruction trap	None	None
0xFFF6, 0xFFF7	SWI	None	None
0xFFF4, 0xFFF5	XIRQ	X-Bit	None
0xFFF2, 0xFFF3	IRQ	I bit	INTCR (IRQEN)
0xFFF0, 0xFFF1	Real time Interrupt	I bit	CRGINT (RTIE)
0xFFEE, 0xFFEF	Standard timer channel 0	I bit	TIE (C0I)
0xFFEC, 0xFFED	Standard timer channel 1	I bit	TIE (C1I)
0xFFEA, 0xFFEB	Standard timer channel 2	I bit	TIE (C2I)
0xFFE8, 0xFFE9	Standard timer channel 3	I bit	TIE (C3I)
0xFFE6, 0xFFE7	Standard timer channel 4	I bit	TIE (C4I)
0xFFE4, 0xFFE5	Standard timer channel 5	I bit	TIE (C5I)
0xFFE2, 0xFFE3	Standard timer channel 6	I bit	TIE (C6I)
0xFFE0, 0xFFE1	Standard timer channel 7	I bit	TIE (C7I)
0xFFDE, 0xFFDF	Standard timer overflow	I bit	TMSK2 (TOI)
0xFFDC, 0xFFDD	Pulse accumulator A overflow	I bit	PACTL (PAOVI)

Vector Address	Interrupt Source	CCR Mask	Local Enable
0xFFDA, 0xFFDB	Pulse accumulator input edge	I bit	PACTL (PAI)
0xFFD8, 0xFFD9	SPI	I bit	SPICR1 (SPIE, SPTIE)
0xFFD6, 0xFFD7	SCI	I bit	SCICR2 (TIE, TCIE, RIE, ILIE)
0xFFD4, 0xFFD5	Reserved		
0xFFD2, 0xFFD3	ATD	I bit	ATDCTL2 (ASCIE)
0xFFD0, 0xFFD1	Reserved		
0xFFCE, 0xFFCF	Port J	I bit	PIEP (PIEP7-6)
0xFFCC, 0xFFCD	Reserved		
0xFFCA, 0xFFCB	Reserved		
0xFFC8, 0xFFC9	Reserved		
0xFFC6, 0xFFC7	CRG PLL lock	I bit	PLLCR (LOCKIE)
0xFFC4, 0xFFC5	CRG self clock mode	I bit	PLLCR (SCMIE)
0xFFBA to 0xFFC3	Reserved		
0xFFB8, 0xFFB9	FLASH	I bit	FCNFG (CCIE, CBEIE)
0xFFB6, 0xFFB7	CAN wake-up(1)	I bit	CANRIER (WUPIE)
0xFFB4, 0xFFB5	CAN errors[1]	I bit	CANRIER (CSCIE, OVRIE)
0xFFB2, 0xFFB3	CAN receive[1]	I bit	CANRIER (RXFIE)
0xFFB0, 0xFFB1	CAN transmit[1]	I bit	CANTIER (TXEIE[2:0])
0xFF90 to 0xFFAF	Reserved		
0xFF8E, 0xFF8F	Port P	I bit	PIEP (PIEP7-0)
0xFF8C, 0xFF8D	PWM Emergency Shutdown	I bit	PWMSDN(PWMIE)
0xFF8A, 0xFF8B	VREG LVI	I bit	CTRL0 (LVIE)
0xFF80 to 0xFF89	Reserved		

For Example, The SCI/UART Does Interrupts

- **Generates interrupts when you need to service the SCI**
  - Interrupts acknowledge (stop being asserted) when status flags reset

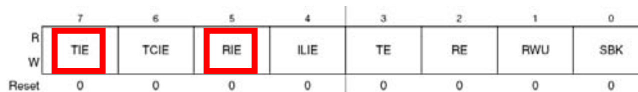


Figure 13-5. SCI Control Register 2 (SCICR2)

Table 13-4. SCICR2 Field Descriptions

Field	Description
7 TIE	**Transmitter Interrupt Enable Bit** — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	**Transmission Complete Interrupt Enable Bit** — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	**Receiver Full Interrupt Enable Bit** — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled

Some Notes On Saving State

- **Many processors don't automatically save state!**
  - For example, a RISC with 32 registers usually doesn't save them
  - It is the ISR's responsibility to save things it changes, then restore them
- **In most systems, the flags are automatically saved**
  - Interrupt can happen after any instruction – so need to save the flags
  - What if you get an interrupt partway through a multi-precision add?
  - What if you get an interrupt between a TST and BEQ?
- **Tricky part – what's up with the "I" bit?**
  - Part of the flag bits
  - … see next slide …

Interrupt Masking

- **When we're in the ISR, what prevents TOF from interrupting us again?**
  - Interrupt processing saves the flag registers, including old I bit
  - The I bit gets set by hardware while the ISR vector is fetched, masking interrupts (causes interrupts to be **Ignored**)
    - No futher interrupts will be recognized in the ISR
  - I bit gets restored as part of the RTI – re-enabling interrupts

2.2.5.4 I Mask Bit

The I bit enables and disables maskable interrupt sources. By default, the I bit is set to 1 during reset. An instruction must clear the I bit to enable maskable interrupts. While the I bit is set, maskable interrupts can become pending and are remembered, but operation continues uninterrupted until the I bit is cleared.

When an interrupt occurs after interrupts are enabled, the I bit is automatically set to prevent other maskable interrupts during the interrupt service routine. The I bit is set after the registers are stacked, but before the first instruction in the interrupt service routine is executed.

Normally, an RTI instruction at the end of the interrupt service routine restores register values that were present before the interrupt occurred. Since the CCR is stacked before the I bit is set, the RTI normally clears the I bit, and thus re-enables interrupts. Interrupts can be re-enabled by clearing the I bit within the service routine, but implementing a nested interrupt management scheme requires great care and seldom improves system performance.

Software Can Set / Clear The I Bit Too

- **Assembly language**
  - SEI – set I bit to 1
    - Causes interrupts to be ignored (masked)
  - CLI – clear I bit to 0
    - Causes interrupts to be permitted
  - These are not needed within interrupts themselves
    - ISRs disable/enable automatically
    - But sometimes you want to disable/enable outside an ISR for some reason
- **In C**
  - EnableInterrupts();
    - Put into your main; I bit set on system reset and you need to clear it this way
  - Other obtuse syntax approaches as well … see CW C references
  - Chip has a few other specialized interrupt masks as well…
    - But don't worry about them for this lecture
- **We're going to see more about the I bit when we discuss concurrency**

SWI As A System Call

- **Totally different use of "interrupts" from everything else in this lecture**
  - So, first, any questions up to this point?
  - The below technique is more common on larger systems, but still important
- **Background – what does a BIOS do?**
  - BIOS = "Basic Input/Output Subsystem"
  - Originated as a UV-EPROM on early microcomputers
    - Knows how to get a keystroke input
    - Knows how to write a character to the screen
    - Knows how to write a sector to disk
    - Keep real time clock
  - In embedded, might also:
    - Read/write serial port
    - Read A/D; write D/A

Simple But Fragile System Calls

- **Old way – early personal computers (for example, Apple][ )**
  - Write BIOS in assembly language
  - Record start addresses of every service routine
  - JSR to the service you want (e.g., GetKey EQU $F75B
    JSR GetKey)
- **What if you need to change the BIOS?**
  - Need to preserve the entry points, but new software might be in different places
  - Once you publish an entry point, you have to support it forever
  - Can't just re-compile the applications; many are distributed as binary only
  - Having a jump table at start of BIOS might help a bit
    - Nth jump table entry is a vector to Nth BIOS service; updated with new version
- **What if you want to establish some sort of protected mode for the OS**
  - What if someone just JSRs to an address other than the designated service address?
  - Without protection, tasks can access any OS fragment they want

Using SWI As A System Call

- **Solution: use SWI as a system call (or "service" call)**
  - Put which OS function you want in A register
  - Put parameters in B, X, Y registers
  - Optionally can put other parameters on stack
  - IBM PC BIOS used this approach
- **When you want a service, load A register and execute SWI**
  - e.g.,

```
LDAB OutByte  ; data to send to serial port
LDAA #7       ; function 7 outputs byte to serial port
SWI           ; BIOS call = send data byte in B to serial port
```

- **Advantages to SWI**
  - SWI handler knows where services start
    - Can change entry points with ease when recompiling the BIOS
  - One place to handle all service calls
    - CPU can change protection modes when it executes SWI
  - Easier to protect BIOS code from malicious execution
    - Use memory management unit to block JSRs into BIOS
  - (Some CPUs don't push all registers on interrupt, so can be very fast as well)

Review

- **Really "getting" interrupts is an essential embedded system skill**
- **Hardware interrupts**
  - Asynchronous, hardware-triggered cousin to SWI
- **What happens in a HW interrupt? (at a detailed level)**
  - Trigger interrupt and set I bit
  - Save state
  - Execute an Interrupt Service Routine specified by a vector
  - Acknowledge or otherwise clear the interrupt (so it doesn't retrigger)
  - Take care of any action needed (execute body of ISR)
  - Clear I bit and resume execution of main program
- **Timer example**
  - Real time clock from last lecture – but done with interrupts
  - Complete example in both assembler and C – what do they do; how do they work?
- **Basic idea of SWI as a service call**