



uOttawa

L'Université canadienne
Canada's university

CEG3136

Computer Architecture II

Interrupts

Notes for
Dr. Voicu Groza

Université d'Ottawa | University of Ottawa



Topics of discussion

- Interrupt Basics
 - ☐ What is an interrupt
 - ☐ Interrupt Hardware
- Servicing Interrupts
- HCS12 Interrupts
 - ☐ Interrupt Process
 - ☐ Maskable Interrupts
 - ☐ Unmaskable Interrupts
- Reading: Chapter 12

Interrupt Fundamental Concepts

■ What is an interrupt?

- ☐ *A signal that indicates an event* which interrupts the execution of a program and that needs to be serviced.
- ☐ Examples: I/O, timeouts, illegal code, divide by 0, etc.

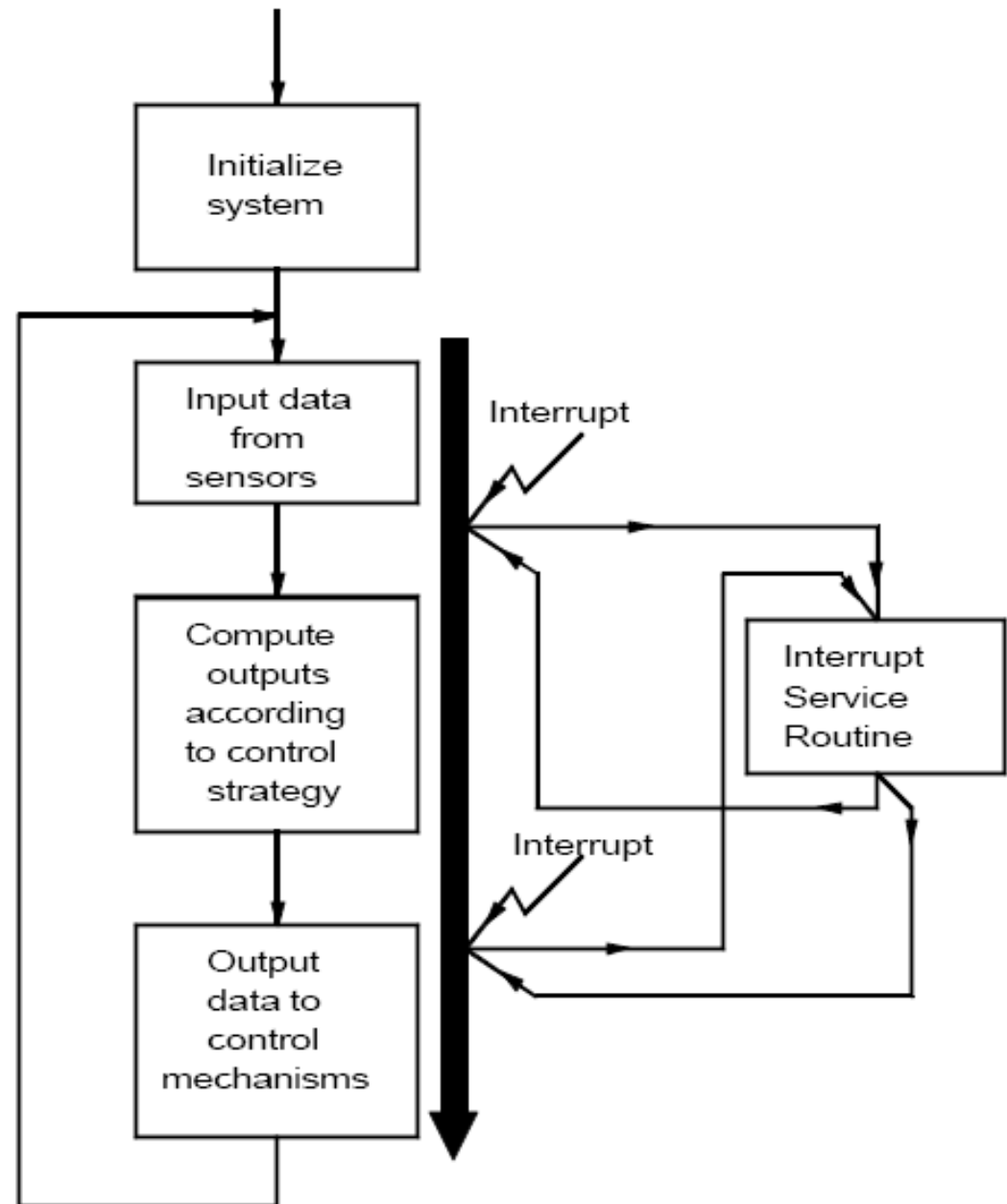
■ Uses for interrupts

- ☐ Coordinating I/O without occupying continually the CPU
- ☐ Dealing with errors
- ☐ Dealing with routine tasks

■ Servicing an interrupt

- ☐ The CPU services an interrupt by executing the *interrupt service routine (ISR)*

Interrupt Fundamental Concepts (continued)

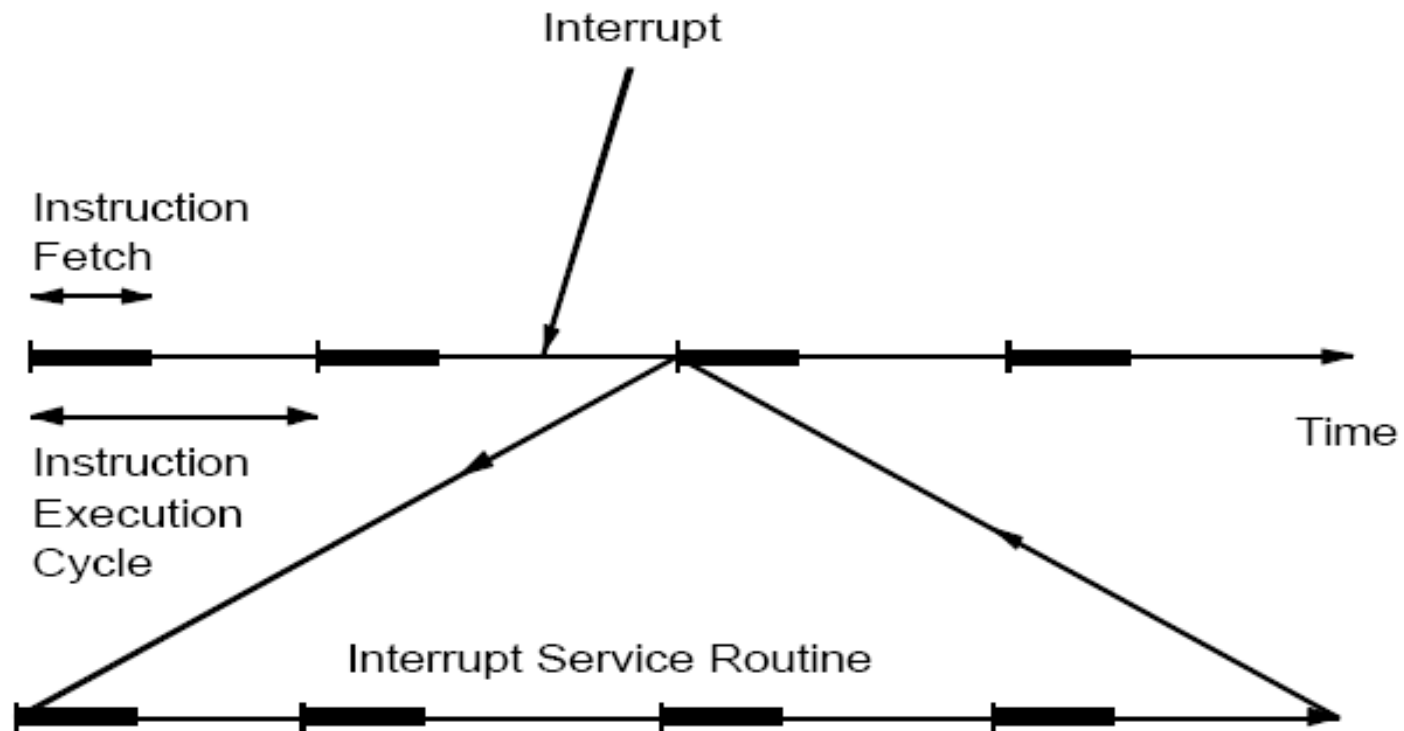


Interrupt System Specification

- Allow for asynchronous events to occur and be recognized.
 - ☐ Variety of interrupt signals – levels and edges.
- Wait for the current instruction to finish before servicing an interrupt.
- Service interrupt with sub-routine and return to interrupted code.
- Enabling and disabling interrupts
 - ☐ All interrupts
 - ☐ Selected interrupts
 - ☐ Disable interrupts while servicing an interrupt
- Multiple sources of interrupts
 - ☐ Simultaneous interrupts.

Internal Asynchronous CPU Timing

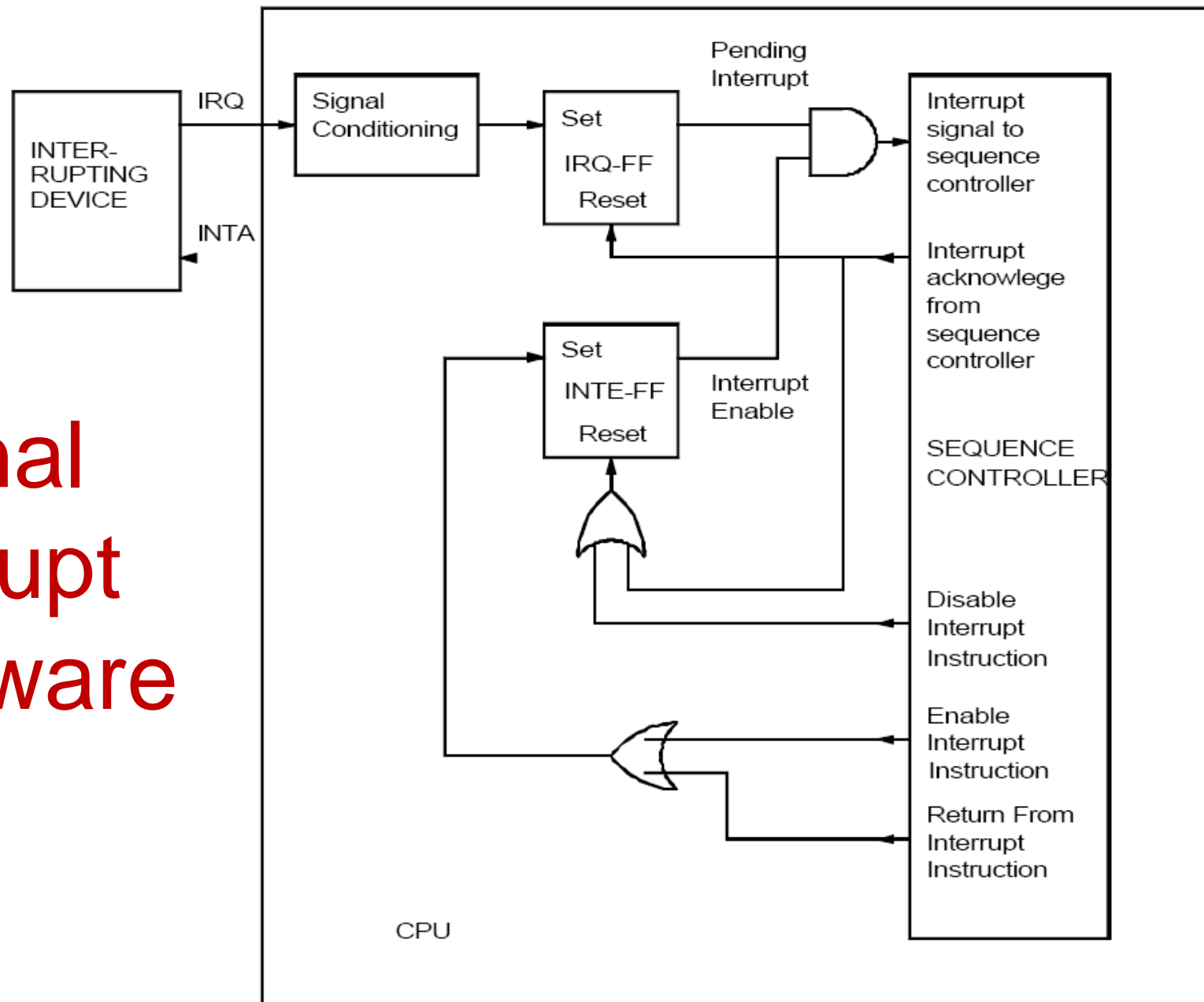
- Interrupts can occur at any time during an instruction cycle



Internal CPU Interrupt Hardware

- Use a flip-flop (IRQ-FF) to catch IRQ
 - ☐ Interrupt signal may be triggered by a signal edge or signal level
 - ☐ Multiple device interrupt lines will be wire-ORed together
 - ☐ Must wait until CPU can process interrupt
- Sequence controller can acknowledge interrupt
- Can enable/disable interrupt using enable-disable flip-flop (INTE-FF)
 - ☐ When interrupt is acknowledged, interrupts are disabled
 - ☐ Also disabled when a reset occurs
- At the end of a service routine RTI is executed
 - ☐ This will enable interrupts

Internal Interrupt Hardware



Interrupts from Multiple Sources

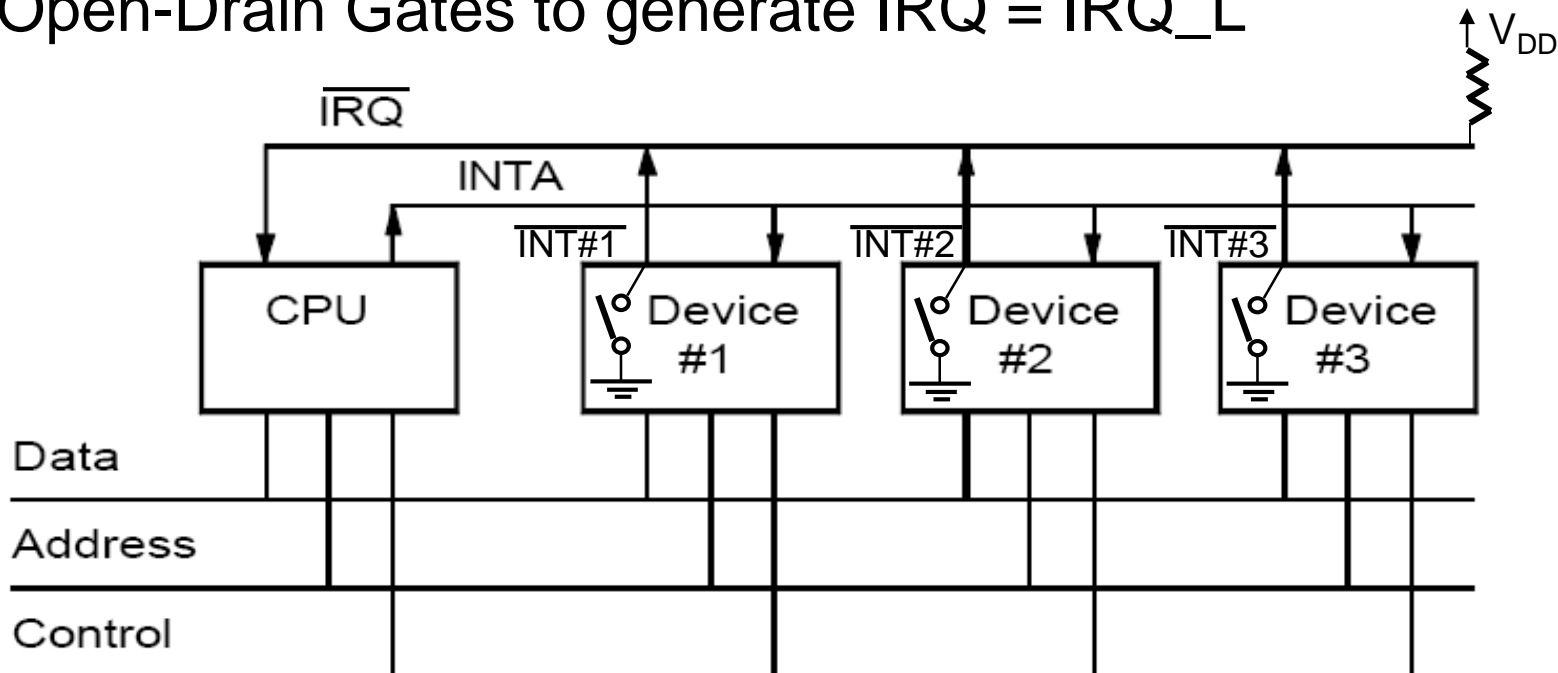
- **Challenge:** how to determine which device has generated the interrupt.

Interrupt Request i \Rightarrow turn switch ON $\Rightarrow \overline{IRQ} = 0$ ($IRQ = 1$)

NO interrupt request \Rightarrow turn switch OFF $\Rightarrow IRQ = 1$ ($IRQ = 0$)

$IRQ = INT\#1 + INT\#2 + INT\#3 \Leftrightarrow \overline{IRQ} = \overline{INT\#1} * \overline{INT\#2} * \overline{INT\#3}$

Use Open-Drain Gates to generate $\overline{IRQ} = IRQ_L$

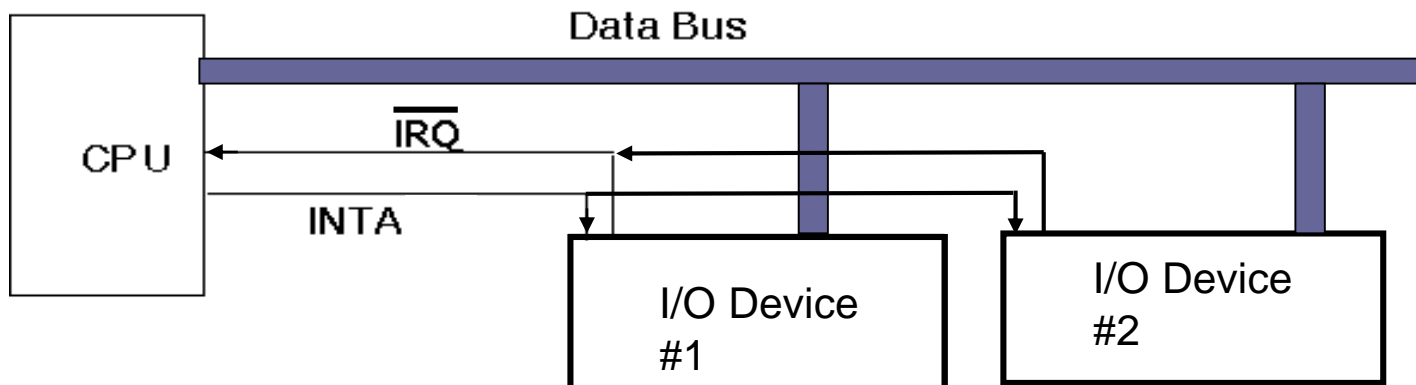


Polled Interrupts

- Vector points to interrupt service routine.
 - A vector is an address in memory.
- Interrupt service routine (ISR)
 - Scans status registers to find which device invoked the interrupt
 - Branches to appropriate code for servicing the device.

Vectored Interrupts – Approach 1

- When the CPU acknowledges the interrupt (INTA), the interrupting device provides some data (can be called a vector).
 - Place on data bus, data used to construct the address of service routine.
 - Logic in sequence controller and interrupting device is more complex
 - Vector data can be hardwired, set by DIP switches, or programmed in the support devices.



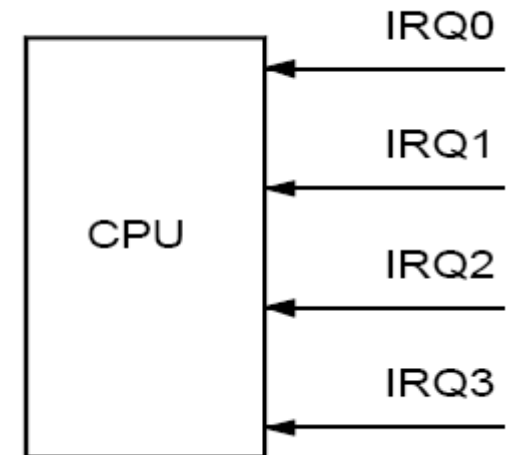
Vectored Interrupts – Approach 2

- Use different signal lines for each interrupt

- A vector in memory exists for each line in reserved locations, in an Interrupt-Vector Table.

- Practical in microcontrollers

- All peripheral devices in a microcontroller invokes separate IRQ signals.

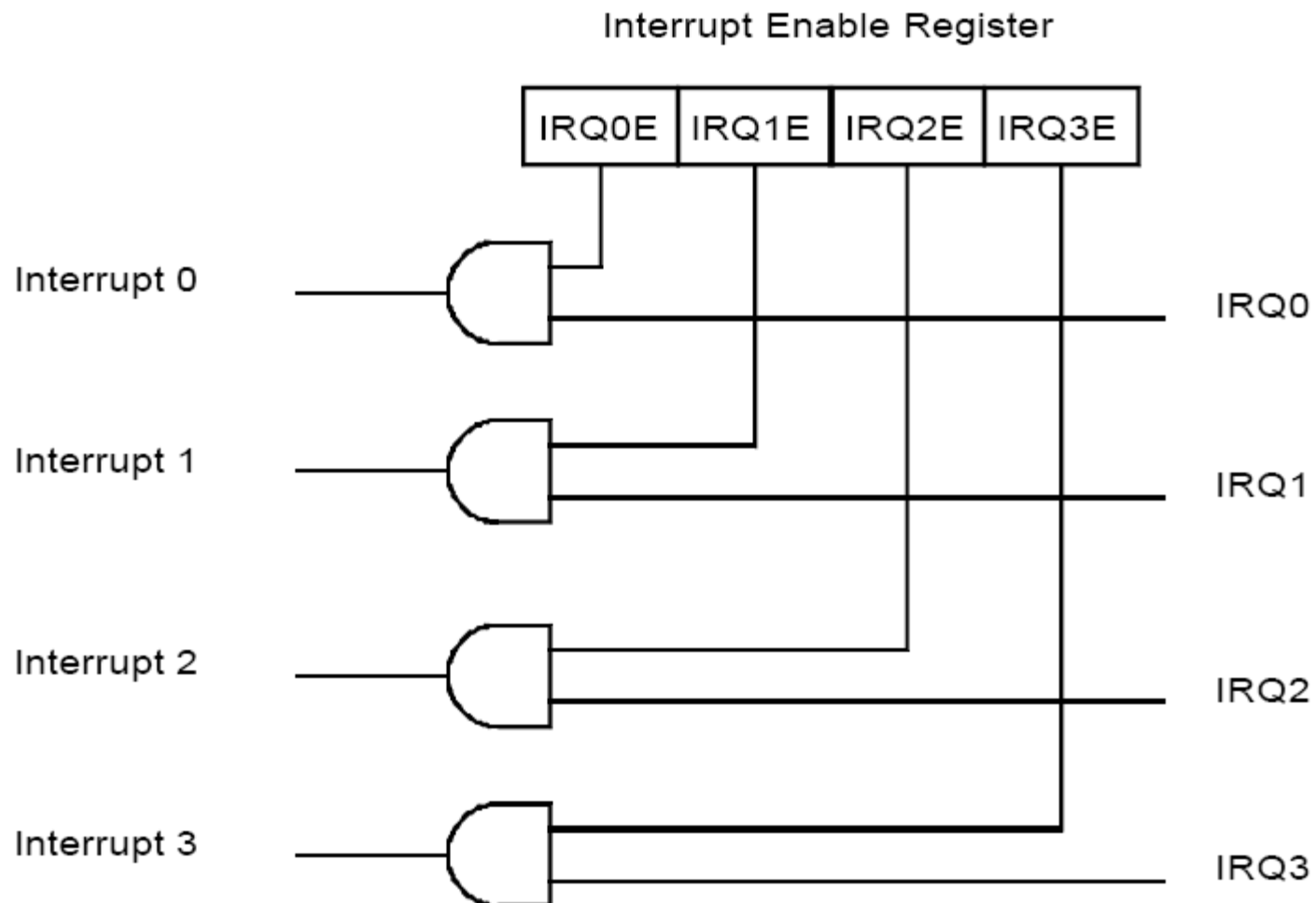


Enabling Interrupts from Multiple Sources

■ Interrupt masking

- ☐ Interrupts that can be ignored by the CPU are called maskable interrupts
- ☐ A maskable interrupt must be enabled
- ☐ Enable flags in the peripherals are used to enable an interrupt
- ☐ Interrupts that cannot be ignored are called non-maskable interrupts.

Multiple Interrupt Masking



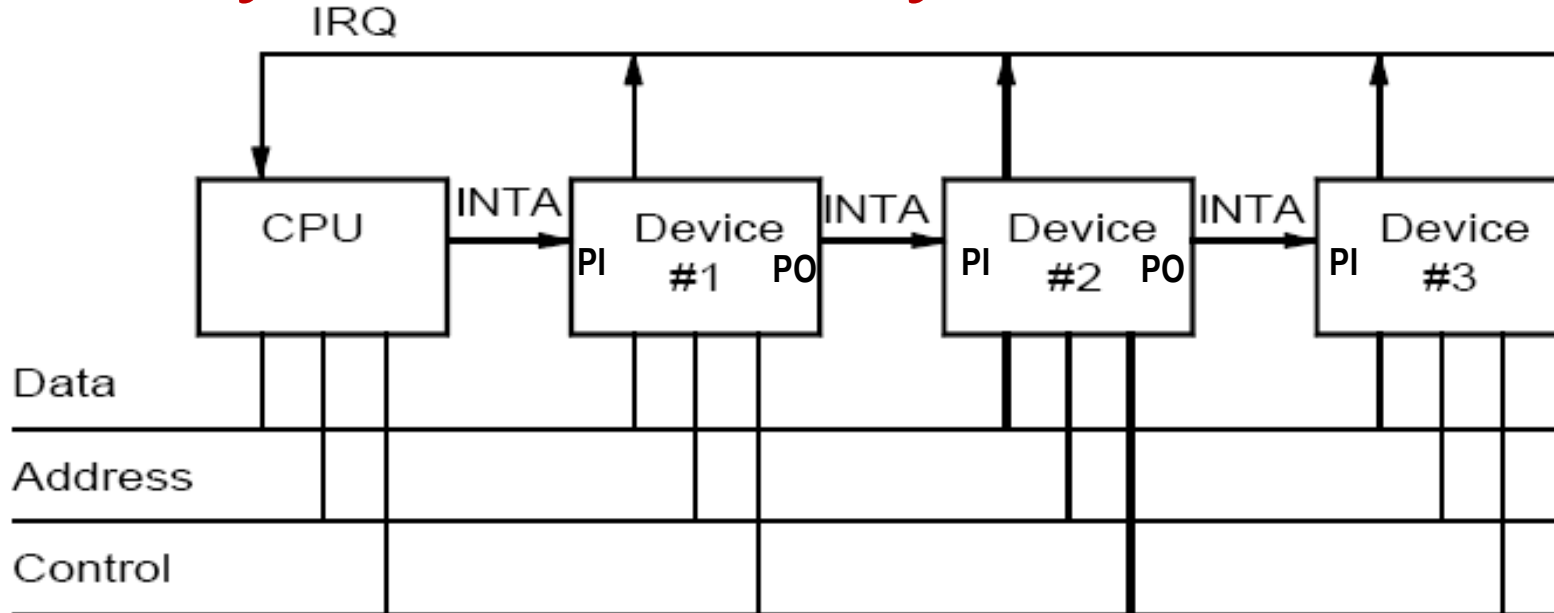
Sequential and Nested Interrupts

- Consequence of disabling interrupts when an interrupt occurs
 - ☐ Interrupts are serviced sequentially
- Can also nest interrupt service routines
 - ☐ Nesting allows another interrupt to interrupt a service routine.
 - ☐ When entering a service routine, the interrupts must be enabled.

Simultaneous Interrupt - Priorities

- Challenge: Two interrupts occur simultaneously – which to service first?
 - ☐ Define interrupt priorities.
- Software method
 - ☐ Polling software checks device status registers in order of priority
- Hardware methods
 - ☐ Daisy chain: Interrupt acknowledge signal passed from device to device (an interrupting device does not pass interrupt acknowledge signal).
 - ☐ Separate IRQ Lines: The lines have a priority.
 - ☐ Hierarchical: Mask enables a set of higher priority signals and masks lower ones
 - ☐ Nonmaskable interrupts: cannot be disabled.

Daisy Chain Priority Scheme



When an interrupt occurs, the CPU acknowledges with an *interrupt acknowledge* signal (INTA)

This signal is received by the PI of the device with the highest priority

- If this device caused the interrupt its PO output is left inactive and its interrupt vector is placed on the data bus.
- If the device did not cause the interrupt, it activates its PO output which is passed on the next device in the chain.

NOTE: Only one interrupt can be serviced at a time with this scheme

Interrupt Latency

- Time between interrupt assertion and start of service routine
 - ☐ Completion of current instruction.
 - ☐ Time to save registers on the stack
 - ☐ Resolve priorities
 - ☐ Completion of other service routines

Interrupt Service Cycle

1. Save the PC on the stack
2. Save the state of the CPU on the stack (including the CCR)
3. Identify the source of the interrupt
4. Find the ISR address corresponding to the interrupt
5. Execute the Interrupt Service Routine (ISR)
6. Restore the CPU to its state before the interrupt
7. Restart the interrupted program

Steps in Interrupt Programming

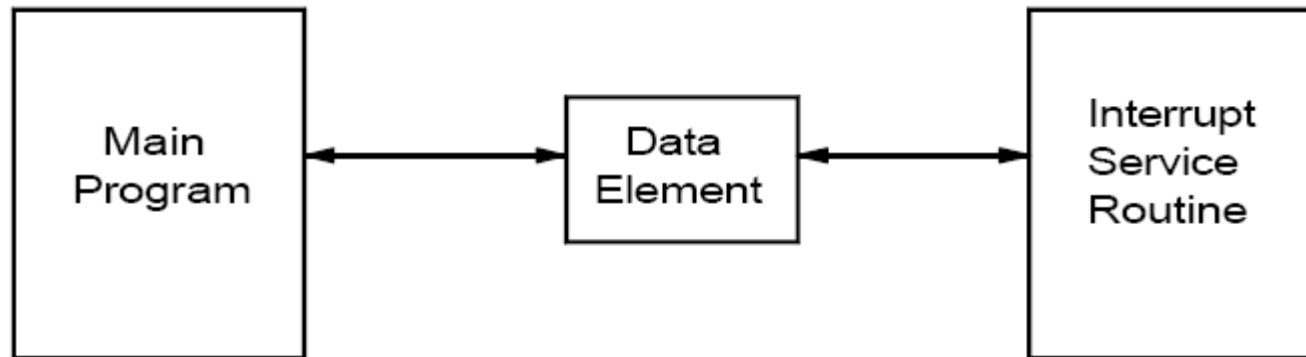
- Step 1: Initialise the interrupt vector table
- Step 2: Write the interrupt service routine (ISR)
- Step 3: In the main program, activate the interrupt
 - Enabling global interrupts in global initialisation routine
 - Enable specific interrupts in associated modules
- Interrupt overhead
 - Saving and restoring the CPU state (HCS12 saves all CPU registers)
 - Execution of the Interrupt Service Routine (ISR)

Developing Interrupt Service Routine (ISR)

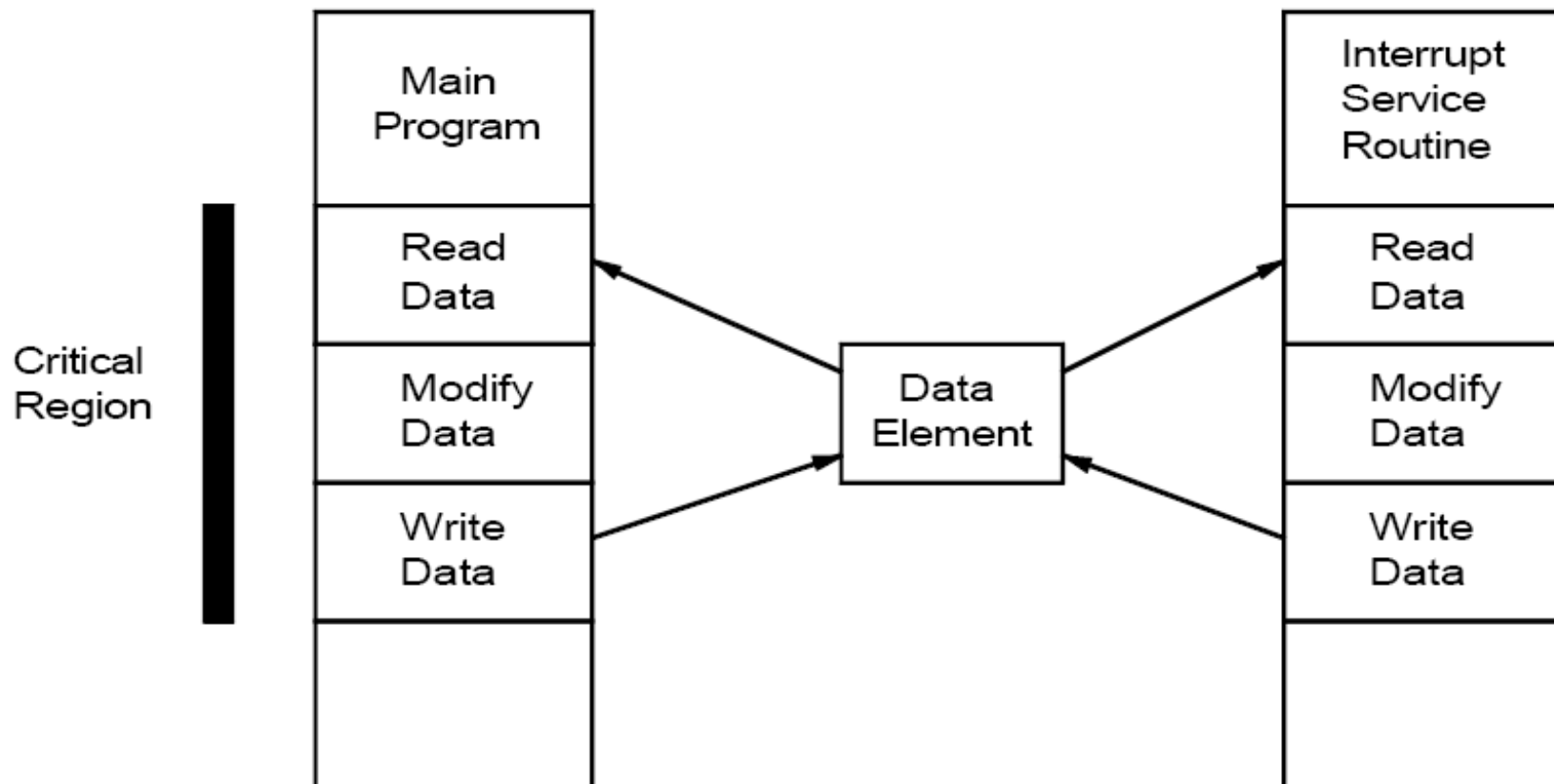
- Do not modify registers and flags
 - ☐ Necessary to allow correct execution of code
- Unmasking interrupts
 - ☐ Only if needed, unmask interrupts to service higher priority interrupts
- Keep the ISR as short as possible
- If possible avoid nested interrupts

Data Exchange with ISRs

- Use global data
- May need to disable interrupts for critical regions of code using this data



Critical Regions



Return from Interrupts

- Important to preserve the state of the CPU
- Must use a return from interrupt instruction, not return from subroutine
 - Sequence controller processes return from interrupt differently than return from subroutine.
- Motorola CPUs saves CPU state on the stack
- Intel CPUs typically do not save CPU state (resembles more the call to subroutine)

Other Interrupt Service Requests

■ Non-maskable Interrupts (NMI)

- ☐ To process important events such as power failure

■ Software Interrupts

- ☐ Used for testing and debugging (SWI in D-Bug12)

■ Exceptions

- ☐ Divide by zero, illegal instructions, bus errors, etc.

■ Resets

- ☐ To start a program after reset.

HCS12 Interrupts

■ Maskable:

- ☐ IRQ pin (port E)
- ☐ peripheral modules
- ☐ Port P: 8 pins
- ☐ Port J: 2 pins

■ Non-maskable:

- ☐ XIRQ pin, SWI, and illegal opcode trap
- ☐ Reset: power on, manual reset, COP (computer operate properly), and clock reset

Interrupt Process

■ Interrupt Enable

- **CLI (I← 0)** (ANDCC #%11101111) – Clears the I bit in the CCR to enable the maskable interrupts

- When used at the beginning of an ISR, allows nested interrupts

■ Interrupt Disable

- **SEI (I← 1)** (ORCC #%00010000) – Sets the I bit in the CCR to disable maskable interrupts

■ Interrupt Request

- Internal interrupts

- External interrupts (IRQ, XIRQ, Port P & Port J)

- Rising Edge triggered (IRQ, XIRQ, Port P, Port J) or
Falling Edge triggered

- Level triggered (IRQ, XIRQ)

- With multiple devices, use level triggered with wire-OR'ed devices using open collector gates.

- CPU must poll external devices to determine interrupting device

- Uses a vector table

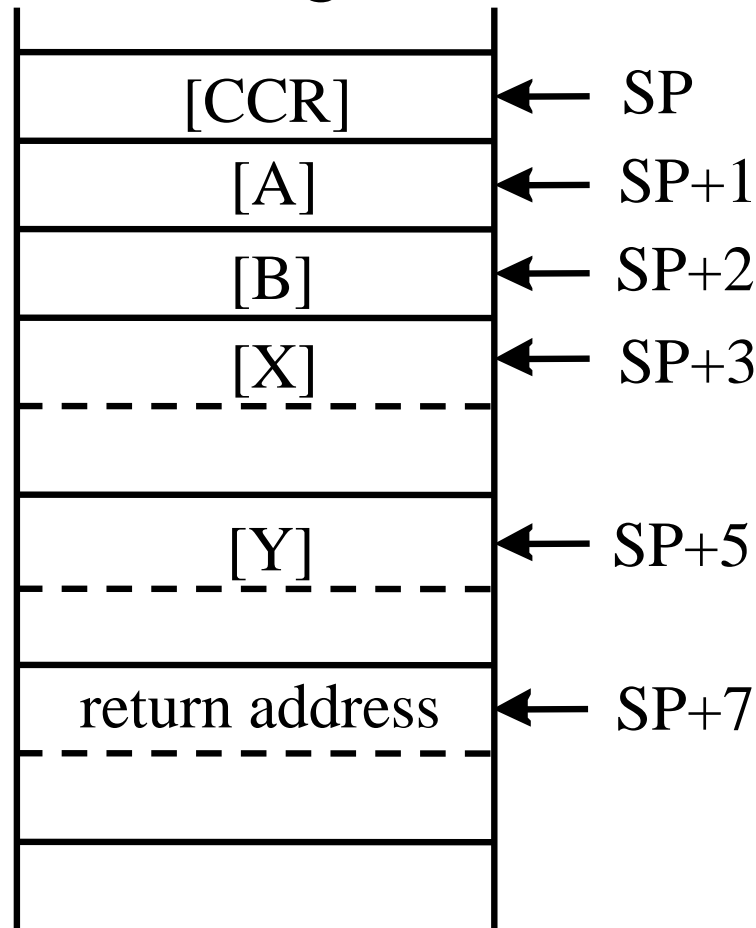
■ Interrupt Sequence – process the interrupt

Interrupt Sequence

- CPU completes current instruction (some of the longer instructions are not completed)
- Return address is pushed onto the stack
- Pushes all CPU registers, X, Y, A, B, and CCR onto the stack
- I bit is set to prevent further maskable interrupts
- Find the vector corresponding to the interrupt
 - Branches to the ISR
- At the return
 - Developer must ensure that interrupt has been cleared in interrupting device
 - RTI will restore CPU state (registers)
 - If no other interrupt is pending, restore PC
 - If another interrupt is pending, start sequence again

Stack on Entry to Interrupt Service Routine (ISR)

- The HCS12 saves all registers on the stack



Interrupt Vectors – MC9S12DG256 (1)

Table 5-1 Interrupt Vector Locations

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFFE, \$FFFF	Reset	None	None	–
\$FFFC, \$FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
\$FFFA, \$FFFB	COP failure reset	None	COP rate select	–
\$FFF8, \$FFF9	Unimplemented instruction trap	None	None	–
\$FFF6, \$FFF7	SWI	None	None	–
\$FFF4, \$FFF5	XIRQ	X-Bit	None	–
\$FFF2, \$FFF3	IRQ	I-Bit	IRQCR (IRQEN)	\$F2
\$FFF0, \$FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	\$F0

Source: MC9S12DT256 Device User Guide, V03.07

Interrupt Vectors – MC9S12DG256 (2)

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFEE, \$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	\$EE
\$FFEC, \$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	\$EC
\$FFEA, \$FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	\$EA
\$FFE8, \$FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	\$E8
\$FFE6, \$FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	\$E6
\$FFE4, \$FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	\$E4
\$FFE2, \$FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	\$E2
\$FFE0, \$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	\$E0
\$FFDE, \$FFDF	Enhanced Capture Timer over o w	I-Bit	TSRC2 (TOF)	\$DE

Source: MC9S12DT256 Device User Guide, V03.07

Interrupt Vectors – MC9S12DG256 (3)

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFD8, \$FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	\$D8
\$FFD6, \$FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	\$D6
\$FFD4, \$FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	\$D4
\$FFD2, \$FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	\$D2
\$FFD0, \$FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	\$D0
\$FFCE, \$FFCF	Port J	I-Bit	PTJIF (PTJIE)	\$CE
\$FFCC, \$FFCD	Port H	I-Bit	PTHIF(PTHIE)	\$CC
\$FF8E, \$FF8F	Port P Interrupt	I-Bit	PTPIF (PTPIE)	\$8E

Source: MC9S12DT256 Device User Guide, V03.07

HOW TO Initialise the Vector Table of HCS12

■ With assembler directives

```
org $FF80
```

```
...
```

```
dc.w    pbov_isr ; pulse accumulator B overflow interrupt vector
```

```
...
```

```
dc.w    atd_isr      ; A/D conversion complete interrupt service routine
```

```
...
```

```
dc.w    pai_isr      ; PAI interrupt service routine
```

```
...
```

```
dc.w    c7_isr      ; output compare channel 7 interrupt service routine
```

```
dc.w    c6_isr      ; input capture channel 6 interrupt service routine
```

```
...
```

Vector Initialisation with D-Bug-12

- Interrupt vector addresses are part of the D-Bug12 code
 - D-Bug12 contains default ISR's
 - Also, it maintains in **RAM** a parallel table to the HCS12 vector interrupt table
 - When an address in the RAM memory table is non-zero, it is used as the ISR vector address
- The D-Bug12 monitor offers the ***SetUserVector*** function to initialise a **user** interrupt vector

int SetUserVector(**int** VectNum, **Address** UserAddress);

; VectNum – vector number (see next table)

; UserAddress – **YOUR** ISR address

D-Bug12 RAM Vector Table

- In RAM @ RAM Vector Address \$3E00+...
- IF(RAM Vector Address) is
= \$0000 → control is returned to D-Bug12 which displays the registers' content and the source of Interrupt
≠ \$0000 → control is given to the respective ISR starting at the address (≠ \$0000) provided here

Interrupt Source	RAM Vector Address	Interrupt Source	RAM Vector Address
Reserved \$FF80	\$3E00	IIC Bus	\$3E40
Reserved \$FF82	\$3E02	DLC	\$3E42
Reserved \$FF84	\$3E04	SCME	\$3E44
Reserved \$FF86	\$3E06	CRG Lock	\$3E46
Reserved \$FF88	\$3E08	Pulse Accumulator B Overflow	\$3E48
Reserved \$FF8A	\$3E0A	Modulus Down Counter Underflow	\$3E4A
PWM Emergency Shutdown	\$3E0C	Port H Interrupt	\$3E4C
Port P Interrupt	\$3E0E	Port J Interrupt	\$3E4E
MSCAN 4 Transmit	\$3E10	ATD1	\$3E50
MSCAN 4 Receive	\$3E12	ATD0	\$3E52
MSCAN 4 Errors	\$3E14	SCI1	\$3E54
MSCAN 4 Wake-up	\$3E16	SCI0	\$3E56
MSCAN 3 Transmit	\$3E18	SPI0	\$3E58
MSCAN 3 Receive	\$3E1A	Pulse Accumulator A Input Edge	\$3E5A
MSCAN 3 Errors	\$3E1C	Pulse Accumulator A Overflow	\$3E5C
MSCAN 3 Wake-up	\$3E1E	Timer Overflow	\$3E5E
MSCAN 2 Transmit	\$3E20	Timer Channel 7	\$3E60
MSCAN 2 Receive	\$3E22	Timer Channel 6	\$3E62
MSCAN 2 Errors	\$3E24	Timer Channel 5	\$3E64
MSCAN 2 Wake-up	\$3E26	Timer Channel 4	\$3E66
MSCAN 1 Transmit	\$3E28	Timer Channel 3	\$3E68
MSCAN 1 Receive	\$3E2A	Timer Channel 2	\$3E6A
MSCAN 1 Errors	\$3E2C	Timer Channel 1	\$3E6C
MSCAN 1 Wake-up	\$3E2E	Timer Channel 0	\$3E6E
MSCAN 0 Transmit	\$3E30	Real Time Interrupt	\$3E70
MSCAN 0 Receive	\$3E32	IRQ	\$3E72
MSCAN 0 Errors	\$3E34	XIRQ	\$3E74
MSCAN 0 Wake-up	\$3E36	SWI	\$3E76
Flash	\$3E38	Unimplemented Instruction Trap	\$3E78
EEPROM	\$3E3A	N/A	\$3E7A
SPI2	\$3E3C	N/A	\$3E7C
SPI1	\$3E3E	N/A	\$3E7E

Vector Numbers

```
Address char *;  
enum Vect {
```

```
UserRsrv0x80 = 0,  
UserRsrv0x82 = 1,  
UserRsrv0x84 = 2,  
UserRsrv0x86 = 3,  
UserRsrv0x88 = 4,  
UserRsrv0x8a = 5,  
UserPWMSHdn = 6,  
UserPortP = 7,  
UserMSCAN4Tx = 8,  
UserMSCAN4Rx = 9,  
UserMSCAN4Errs = 10,  
UserMSCAN4Wake = 11,  
UserMSCAN3Tx = 12,  
UserMSCAN3Rx = 13,  
UserMSCAN3Errs = 14,  
UserMSCAN3Wake = 15,  
UserMSCAN2Tx = 16,  
UserMSCAN2Rx = 17,  
UserMSCAN2Errs = 18,  
UserMSCAN2Wake = 19,  
UserMSCAN1Tx = 20,
```

When accessing the RAM vector table by using the base address (\$3E00), the Vect enumerated constants must be multiplied by two before being used as an offset into the RAM vector table, e.g.,
 $\text{UserIRQ} = \$3\text{E}00 + 57 * 2 = \$3\text{E}00 + \$72 = \$3\text{E}72$

```
UserMSCAN1Rx = 21,  
UserMSCAN1Errs = 22,  
UserMSCAN1Wake = 23,  
UserMSCAN0Tx = 24,  
UserMSCAN0Rx = 25,  
UserMSCAN0Errs = 26,  
UserMSCAN0Wake = 27,  
UserFlash = 28,  
UserEEPROM = 29,  
UserSPI2 = 30,  
UserSPI1 = 31,  
UserIIC = 32,  
UserDLC = 33,  
UserSCME = 34,  
UserCRG = 35,  
UserPAccBOv = 36,  
UserModDwnCtr = 37,  
UserPortH = 38,  
UserPortJ = 39,  
UserAtoD1 = 40,
```

```
UserAtoD0 = 41,  
UserSCI1 = 42,  
UserSCI0 = 43,  
UserSPI0 = 44,  
UserPAccEdge = 45,  
UserPAccOvf = 46,  
UserTimerOvf = 47,  
UserTimerCh7 = 48,  
UserTimerCh6 = 49,  
UserTimerCh5 = 50,  
UserTimerCh4 = 51,  
UserTimerCh3 = 52,  
UserTimerCh2 = 53,  
UserTimerCh1 = 54,  
UserTimerCh0 = 55,  
UserRTI = 56,  
UserIRQ = 57 = $39  
UserXIRQ = 58,  
UserSWI = 59,  
UserTrap = 60,  
RAMVectAddr = -1 };
```

Example 1 – Initialise the RAM Interrupt Vector of Port P

```
portP_isr ...  
    rti
```

Define ISR for port P

```
UserPortP      equ      7 ; interrupt vector number (#) of Port P  
setuservector equ $EEA4 ; setuservector D-Bug12 subroutine  
                  ; starts at $EEA4  
  
    ldd #portP_isr  
    pshd                ; pass the Port P interrupt vector  
                        ; via stack  
  
    ldd # UserPortP ; pass interrupt vector number of Port  
                  ; P through D to setuservector D-Bug12  
                  ; subroutine  
  
    jsr [setuservector,PCR] ; call D-Bug12 function  
                          ; to setup the RAM interrupt  
                          ; vector  
  
    leas 2, sp
```

; deallocate stack space

Servicing an Interrupt in C

- The CPU maintains a vector of addresses to these routines to associate them to the specific interrupts
- CodeWarrior provides the « interrupt » qualifier to declare functions that service interrupts
 - Include the « interrupt » keyword in the function definition.
 - Can also add the interrupt vector number to associate the function to a specific interrupt

Example of an ISR C Function in CodeWarrior

```
/* LED connected to bit 0 of Port A */
/* Interrupts received on IRQ pin */
#define BIT0 0b00000001
#include < mc9s12dg256.h > // Definitions for MC9S12DG256
void main(void) {
    DDRA |= BIT0; / DDRA = DDRA | BIT0 *Output*/
    PORTA &= ~BIT0; / PORTA = PORTA & (BIT0)' *Turn off LED* /
    INTCR |= 0b10000000; // IRQE = 1, IRQ edge-sensitive
    INTCR |= 0b01000000; // IRQEN = 1, Activate IRQ
    asm cli; /* allow interrupts*/
    for(;;) ; /* infinite loop */
}
/*****ISR*****/
/* VectorNumber_Virq = 6 */
void interrupt VectorNumber_Virq irq_handler(void) {
    if(PORTA&BIT0) /* is lit? */
        PORTA &= ~BIT0; /* turn off */
    else
        PORTA |= BIT0; /* turn on */
}
```

Assembler Version with D-Bug12

```
#include mc9s12dg256.inc
UserIRQ    equ 57
setuservector equ    $EEA4

main:
    bset DDRA,0x01 ; Output
    bclr PORTA,0x01 ; turn off
    bset INTCR,%10000000 ; IRQE
    bset INTCR,%01000000 ; IRQEN
    ldd    #irqisr
    pshd   ; pass the IRQ int,
           ; vector via stack
    ldd    #UserIRQ ; IRQ vector #
    ; call D-Bug12 function
    jsr    [setuservector,PCR]
    leas   2,sp ; clean stack
    cli
loop: bra loop ; infinite loop
```

```
;-----
; ISR - interrupt service
irqisr:
    brset PORTA,0x01,turnoff
    bclr PORTA,0x01
    bra leave
turnoff: bset PORTA, 0x01
leave: rti
```


Example 2 – Initialise RAM Interrupt Vector

```
#include "c:\miniide\hcs12.inc"
```

```
RAM_addr_IRQ equ    $3E72                ; = $3E00 + UserIRQ*2 = $3E00 + $39*2
```

```
count    org    $1000
ds.b     1      ; reserve one byte for count
org      $1500
lds      #$1500 ; set up the stack pointer
clr      count
movb     #$FF,ddrb ; configure port B for output
movb     count,portb ; display the count value on LEDs
```

```
movb     #$C0,irqcr ; enable IRQ pin interrupt and select edge-triggering
cli      ;
```

```
forever  nop
bra      forever ; wait for IRQ pin interrupt
```

```
irq_isr  inc      count      ; increment count
movb     count,portb        ; and display it
rti
```

```
org      RAM_addr_IRQ ; establish IRQ interrupt vector entry
dc.w     irq_isr
end
```

Example 3 – Initialise RAM Interrupt Vector

```
#include "c:\miniide\hcs12.inc"
```

```
RAM_addr_IRQ      equ $3E72      ; = $3E00 + UserIRQ*2 = $3E00 + $39*2
```

```
count  org    $1000
        ds.b   1      ; reserve one byte for count
        org    $1500
        lds    #$1500 ; set up the stack pointer
        clr    count
        movb   #$FF,ddrb ; configure port B for output
        movb   count,portb ; display the count value on LEDs
```

```
movw   #irq_isr, RAM_addr_IRQ ; establish IRQ interrupt vector entry in RAM
movb     #$C0,irqcr             ; enable IRQ pin interrupt & select edge-triggering
cli                                             ;
```

```
forever nop
        bra    forever          ; wait for IRQ pin interrupt
```

```
irq_isr inc    count          ; increment count
        movb   count,portb    ; and display it
        rti
end
```

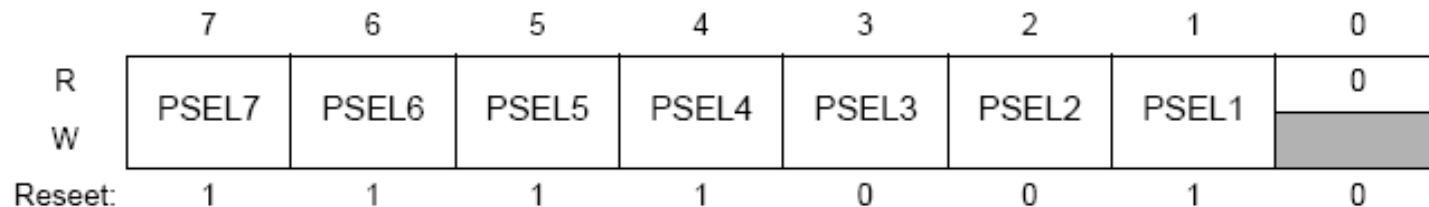
Waiting for an Interrupt

- Spin Loop
 - ☐ Processor branches on itself
- WAI instruction – Wait for interrupt
 - ☐ Pushes all registers on stack
 - ☐ Places CPU in WAIT mode – reduces power consumption
- STOP instruction – Stop clocks
 - ☐ Stops all CPU clocks – dramatically reduces power consumption

Maskable Interrupts - Priority

- Interrupt priorities are established by the vector table
- One maskable interrupt can have its priority raised ahead of the others
- Priority can be raised using the HPRIO register
- To increase the interrupt priority, write the least significant byte of the vector address to the HPRIO register
- For example, vector address for the ATD is \$FFD2
- **To increase its priority, write \$D2 to the HPRIO register**

Register address: Base + \$001F



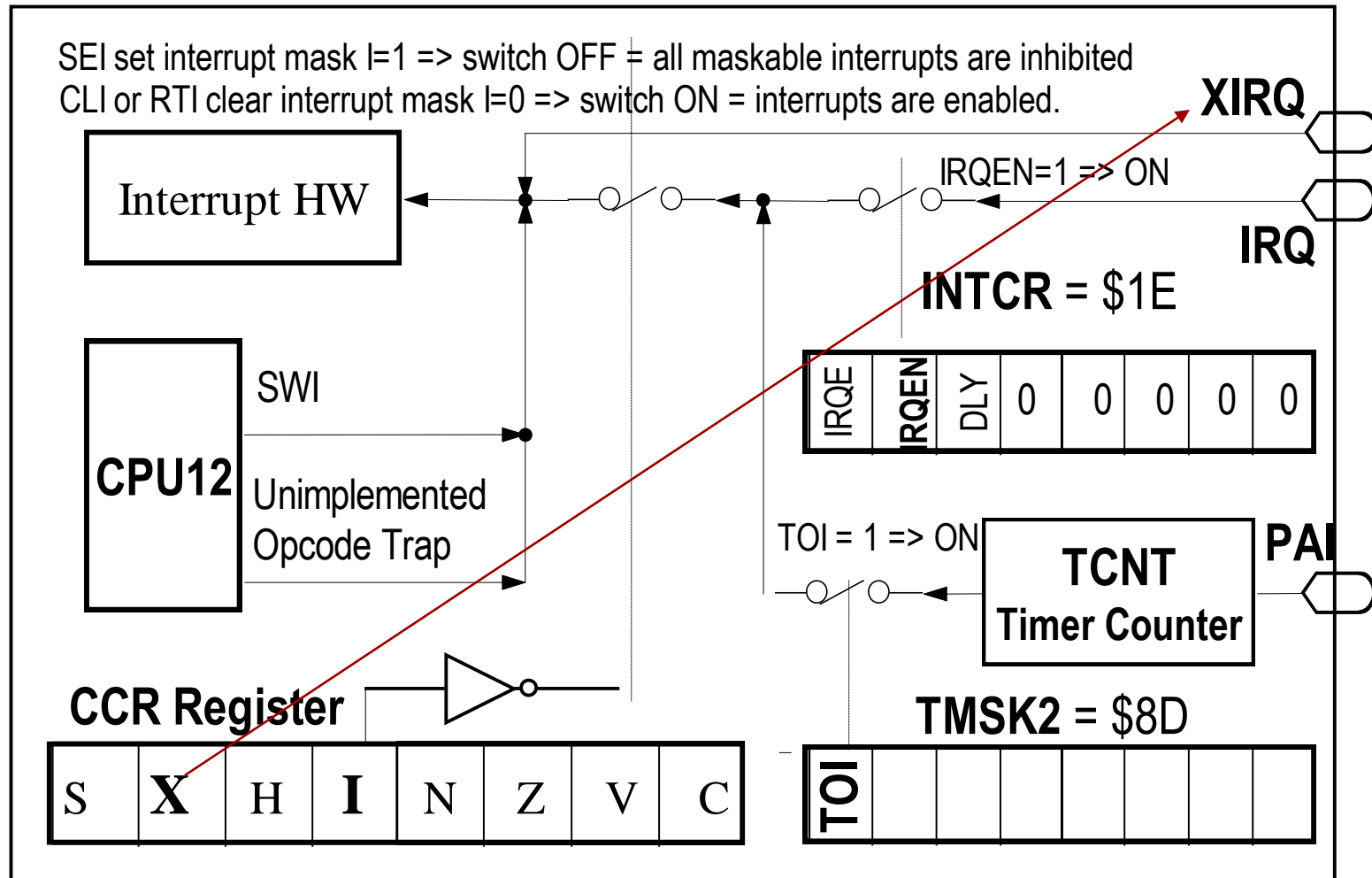
= Unimplemented or Reserved

Figure 3-4 Highest Priority I Interrupt Register (HPRIO)

IRQ Pin Interrupt

- External maskable interrupt for the HCS12.
- Port E, pin 1 (PE1)
- Configurable using the IRQCR register (**IRQ control register**)
 - ☐ Enable bit in the INTCR register
 - ☐ Level triggered
 - ☐ Falling Edge triggered

68HC12 (partial) Interrupt System



INTCR Register

Address	Base + \$__1E							
	Bit 7	6	5	4	3	2	1	Bit 0
Read:	IRQE	IRQEN	0	0	0	0	0	0
Write:								
Reset:	0	1	0	0	0	0	0	0


 = Unimplemented

Figure 3-15 IRQ Control Register (IRQCR)

Read: see individual bit descriptions below

Write: see individual bit descriptions below

IRQE — IRQ Select Edge Sensitive Only

Special modes: read or write anytime

Normal and Emulation modes: read anytime, write once

- 1 = IRQ configured to respond only to falling edges. Falling edges on the IRQ pin will be detected anytime IRQE = 1 and will be cleared only upon a reset or the servicing of the IRQ interrupt.
- 0 = IRQ configured for low level recognition.

IRQEN — External IRQ Enable

Normal, Emulation, and Special modes: read or write anytime

- 1 = External IRQ pin is connected to interrupt logic.
- 0 = External IRQ pin is disconnected from interrupt logic.

NOTE: When *IRQEN* = 0, the edge detect latch is disabled.

Triggering IRQ

■ Level triggered

- ☐ Advantage – many sources can use the same pin
- ☐ Disadvantage – Need to deactivate the signal before the end of the ISR

■ Edge Triggered

- ☐ Advantage: the pulse width is not important
- ☐ Disadvantage – Not practical for noisy environments where noise can be recognised as an interrupt

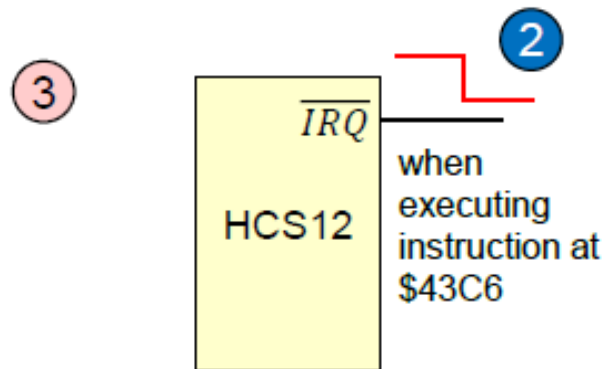
■ CPU checks for interrupts at the end of the current instruction

Picturing an Interrupt

IRQ	\$FFF3	\$04	Address of Interrupt Service Routine in Vector Table
	\$FFF2	\$28	
	\$D000	??	PC (\$43C7) and all registers will be pushed on the stack
	
SP→	\$CFF2	??	
Main:	\$4000	??	Main Program
	
	\$43C6	incx	
	\$43C7	aba	
	
IRQ_ISR:	\$2804	??	Interrupt Service Routine
	
	\$282A	rti	
IRQCR	\$001E	\$C0	IRQ Control – Int. enabled

interrupted here →

- 1 Interrupts have been enabled by clearing I bit in CCR (global) and setting IRQCR to \$C0 (local)



- 4 CPU transfers control to address of interrupt service routine stored in interrupt table
- 5 rti instruction restores registers by pulling their values off the stack
- 6 rti instruction loads PC with return address from stack

Example 2 – Initialise IRQ, ISR, Interrupt Vector

```
#include "c:\miniide\hcs12.inc"
```

```
RAM_addr_IRQ equ    $3E72                ; = $3E00 + UserIRQ*2 = $3E00 + $39*2
```

```
count    org    $1000
          ds.b    1          ; reserve one byte for count
          org    $1500
          lds     #$1500 ; set up the stack pointer
          clr     count
          movb    #$FF,ddrb ; configure port B for output
          movb    count,portb ; display the count value on LEDs
```

A counter is incremented by pressing a push button that is connected to IRQ. Counter's content (binary) is displayed on LED's

```
movb    #$C0,irqcr ; enable IRQ pin interrupt (IRQEN=1)
                ; and select edge-triggering (IRQE=1)
cli      ; enable all maskable interrupts (CCR: I=0)
```

```
forever  nop
          bra     forever ; wait for IRQ pin interrupt
```

```
irq_isr inc    count      ; increment count
        movb   count,portb ; and display it
        rti
```

```
          org    RAM_addr_IRQ          ; establish IRQ interrupt vector entry
          dc.w    irq_isr
          end
```

Example 3 – Initialise IRQ, ISR, Interrupt Vector

```
#include "c:\miniide\hcs12.inc"
```

```
RAM_addr_IRQ equ $3E72      ; = $3E00 + UserIRQ*2 = $3E00 + $39*2
```

```
count    org    $1000
         ds.b    1      ; reserve one byte for count
         org    $1500
         lds     #$1500 ; set up the stack pointer
         clr     count
         movb    #$FF,ddrb ; configure port B for output
         movb    count,portb ; display the count value on LEDs
```

A counter is incremented by pressing a push button that is connected to IRQ. Counter's content (binary) is displayed on LED's

```
movw     #irq_isr, RAM_addr_IRQ ; establish IRQ interrupt vector entry in RAM
movb     #$C0,irqcr           ; enable IRQ pin interrupt & select edge-triggering
cli                                           ; enable all maskable interrupts (CCR: I=0)
```

```
forever  nop
         bra     forever      ; wait for IRQ pin interrupt
```

```
irq_isr inc     count      ; increment count
         movb    count,portb ; and display it
         rti
end
```

Port P Interrupts - Enabling

Address Offset: \$__1E

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	PIEP7	PIEP6	PIEP5	PIEP4	PIEP3	PIEP2	PIEP1	PIEP0
Write:								
Reset:	0	0	0	0	0	0	0	0


 = Reserved or unimplemented

Figure 3-28 Port P Interrupt Enable Register (PIEP)

Read:Anytime.

Write:Anytime.

This register disables or enables on a per pin basis the edge sensitive external interrupt associated with port P.

PIEP[7:0] — Interrupt Enable Port P

1 = Interrupt is enabled.

0 = Interrupt is disabled (interrupt flag masked).

- Port P is also connected to PWM (pulse width modulator) and SPI (synchronous protocol interface)

Interrupts Port P – Control (1)

Address Offset: \$__1C

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	PERP7	PERP6	PERP5	PERP4	PERP3	PERP2	PERP1	PERP0
Write:								
Reset:	0	0	0	0	0	0	0	0



= Reserved or unimplemented

Figure 3-26 Port P Pull Device Enable Register (PERP)

Read: Anytime.

Write: Anytime.

This register configures whether a pull-up or a pull-down device is activated, if the port is used as input. This bit has no effect if the port is used as output. Out of reset no pull device is enabled.

PERP[7:0] — Pull Device Enable Port P

1 = Either a pull-up or pull-down device is enabled.

0 = Pull-up or pull-down device is disabled.

Interrupts Port P – Control (2)

Address Offset: \$__1D

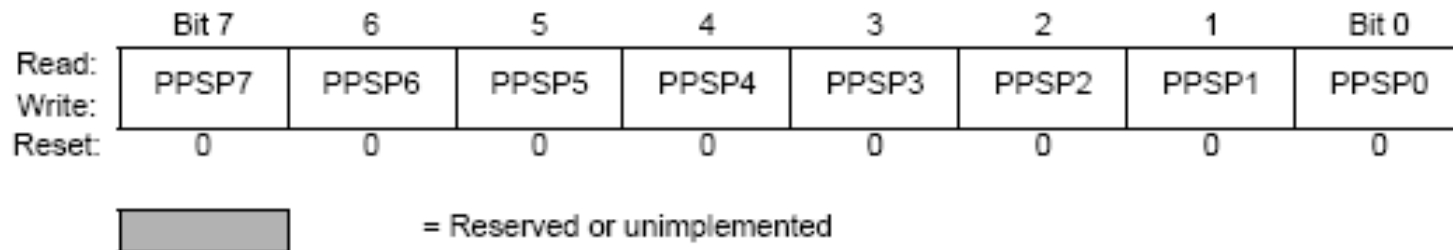


Figure 3-27 Port P Polarity Select Register (PPSP)

Read:Anytime.

Write:Anytime.

This register serves a dual purpose by selecting the polarity of the active interrupt edge as well as selecting a pull-up or pull-down device if enabled.

PPSP[7:0] — Polarity Select Port P

- 1 = Rising edge on the associated port P pin sets the associated flag bit in the PIFP register. A pull-down device is connected to the associated port P pin, if enabled by the associated bit in register PERP and if the port is used as input.
- 0 = Falling edge on the associated port P pin sets the associated flag bit in the PIFP register. A pull-up device is connected to the associated port P pin, if enabled by the associated bit in register PERP and if the port is used as input.

Interrupts Port P – Flags

Address Offset: \$__1F



Figure 3-29 Port P Interrupt Flag Register (PIFP)

Read:Anytime.

Write:Anytime.

Each flag is set by an active edge on the associated input pin. This could be a rising or a falling edge based on the state of the PPSP register. To clear this flag, write “1” to the corresponding bit in the PIFP register. Writing a “0” has no effect.

PIFP[7:0] — Interrupt Flags Port P

1 = Active edge on the associated bit has occurred (an interrupt will occur if the associated enable bit is set).

Writing a “1” clears the associated flag.

0 = No active edge pending.

Writing a “0” has no effect.

Port P and Port J Interrupts

- Port J provides 2 pins for interrupts
 - Registers similar to Port P registers are available to enable, configure and control Port J interrupts.
- Setting PIFP (and PIFJ)
 - Write a 1 into the bit to deactivate the interrupt, i.e., sets the bit to 0
 - Use `MOVB #00000001, PIFJ`
 - Or `BCLR #11111110, PIFJ`
 - Complements the 1 in the mask
 - Changes the 0 to a 1 and applies an AND with the 1 in the register..
 - Thus 1 is written into bit 0
 - In the other bits, zero's will be written which has not effect.

Non-Maskable Interrupts

- RESET
- Clock Monitor Failure
- Computer Operating Properly (COP)
- Unimplemented Instruction Opcode Trap
- SWI
- Non Maskable Interrupt Request (XIRQ)

Resets

- Initial values for certain registers, flip-flops, I/O peripheral control registers are initialized for proper CPU functioning
- The reset sets these initial conditions
- Power on reset and manual resets
- Power on reset initializes all the CPU
- The manual reset allows the CPU to exit an error state
- Resets are non-maskable

Power On Reset (POR)

- The HCS12 contains a circuit to detect a transition on the power pin V_{DD}
- After a delay (that allows the clock signal to stabilize), an internal reset signal is invoked
- Interrupt vector: \$FFFE:\$FFFF

Manual Reset (pin RESET)

- The HCS12 contains a circuit to distinguish internal from external resets (see page 302 of Cady text)
 - At a reset, the HCS12 places 0 V on the reset pin for 16 clock cycles and then releases the pin
 - 8 cycles after the release, the RESET pin is checked: 0V indicates an external RESET, and 1V indicates an internal RESET (COP or clock monitor reset)

Clock Monitor Reset

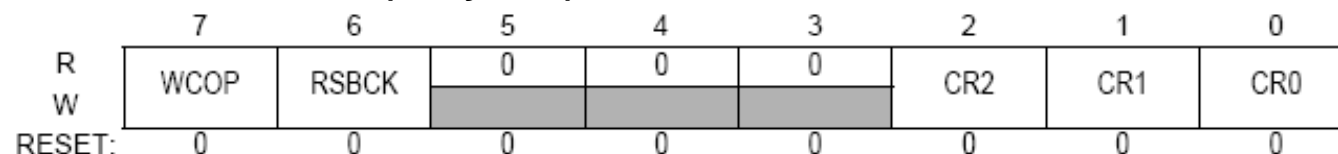
- Checks the clock frequency using an RC circuit
- If no edge is detected within a certain delay set by the RC circuit, the clock monitor can initiate a RESET
 - ☐ Only when the clock is re-established with the interrupt be serviced
- This operation is enabled using the CME and SCME bits in the PLLCTL register (Cady, 303)

Computer Operating Properly COP RESET

- The **COP watchdog** provides recovery from software failure (ex. infinite loop) or power surges
- Can predict execution time of properly functioning software
- When COP is enabled, the application program must write \$55 followed by \$AA in the ARMCOP register (address \$34) within a preset time to prevent COP timer overflow
- If the software is not running properly, the ARMCOP register will not be updated within the preset time – and a COP RESET is generated
- The COPCTL register is used to configure and control the COP function

COP Watchdog Rates

Address Offset: \$003C (Cady, 304)




 = Unimplemented or Reserved

Figure 3-9 CRG COP Control Register (COPCTL)

Table 3-3 COP Watchdog Rates¹

1.024 ms =>

4.096 ms =>

16.384ms=>

CR2	CR1	CR0	OSCCLK cycles to time-out
0	0	0	COP disabled
0	0	1	2^{14}
0	1	0	2^{16}
0	1	1	2^{18}
1	0	0	2^{20}
1	0	1	2^{22}
1	1	0	2^{23}
1	1	1	2^{24}

SWI Interrupt

- Interrupt without a hardware signal
- Often used by a debugging monitor (D-Bug12) to implement breakpoints
- Places SWI at breakpoint addresses
 - ☐ The monitor maintains a table of these points with the original instructions
- When the SWI executes, the monitor examines the PC value on the stack and checks for the value in the table of breakpoints
 - ☐ If the PC value is not part of the table, replace the PC value on the stack with the address of the monitor code and RTI
 - ☐ If the value of the PC is on the stack, replace the SWI with the original instruction code, replace the PC value with the monitor address, display the register values on the screen, and then RTI
- Interrupt vector address: \$FFF6:\$FFF7

Illegal Opcode Reset

- There exists 202 illegal opcodes (16 bit)
- An interruption is generated when such an opcode is encountered
- The RTI will bring the microcontroller to the following instruction
- Interrupt Vector: \$FFF8:\$FFF9

XIRQ Interrupt

- An interrupt on the XIRQ pin is not allowed during a reset
- After a minimum of initialisation, the CCR X bit can be cleared (ANDCC #\$BF) to allow XIRQ interrupts
- Once cleared, the X bit cannot be changed
- When an XIRQ interrupt occurs, the I and X bits are set (preventing maskable and other nested XIRQ interrupts to be taken into consideration during servicing XIRQ)
- The RTI instruction will restore (clears) the X and I bit values, re-arming the interrupt system
- XIRQ Interrupt Vector: \$FFF4:\$FFF5