EEL 4744C: Microprocessor Applications

Lecture 7

Part 2

M68HC12 Interrupt

Reading Assignment

- Software and Hardware Engineering (New version): Chapter 12

or

- SHE (old version) Chapter 8

And

- CPU12 Reference Manual Chapter 7

And

- HC12 Data Sheet Chapter 4

Interrupts (IRQ)

- What is an Interrupt?
  - an asynchronous event that stops normal program execution
  - performs a Service Route (i.e., executes some code)
  - returns program to where it left off
- Why have IRQ's?

1) Good for Asynchronous Events
   - that must be serviced
   - don't want to use main CPU cycles to continually check

2) Good for Synchronizing with external processes
   - if CPU is faster than peripherals (printer, A/D, etc…)
   - peripherals notify CPU when they're ready
   - CPU doesn't waste time "waiting"

3) Good for timed events
   - "Real Time Systems" events happen periodically
   - Multi-tasking

Interrupt 101

- Priority
  - interrupts can occur at the same time
  - some interrupts are more important than others (ex, fire alarm)
  - to handle this, all interrupts are given a default priority
  - some priorities can be changed
- Interrupt Service Route (ISR)
  - this is the code to be executed when an IRQ occurs
  - each individual IRQ has an ISR if it is being used
  - this is similar to a subroutine, it is code written by us
- Interrupt Vector Address
  - each IRQ has a unique Vector Address
  - this holds the address of ISR to be executed if that IRQ occurs

HC12B32 Interrupt Overview

- Uses vectored interrupts, but polling used when multiple external sources on IRQ* line
- Features h/w priority resolution that can be customized via s/w
- IRQ* and XIRQ* (NMI), plus other signals w/ timer subsystem, serial interface, and A/D converter that we'll see in later chapters
- Special interrupts including s/w, illegal opcode, watchdog timer, and clock failure interrupts

External

Internal

Interrupt Vectors

- Interrupt vector is address of start of particular ISR
- When interrupt generated, CPU fetches this address from vector location
- Interrupt vector table for HC12B32 →

Vector Address	Interrupt Source
$FF8D:FFCF	Reserved
$FFD0:FFD1	BDLC
$FFD2:FFD3	A/D Converter
$FFD4:FFD5	Reserved
$FFD6:FFD7	SCI.0 Serial System
$FFD8:FFD9	SPI Serial Transfer Complete
$FFDA:FFDB	Pulse Accumulator Input Edge
$FFDC:FFDD	Pulse Accumulator Overflow
$FFDE:FFDF	Timer Overflow
$FFE0:FFE1	Timer Channel 7
$FFE2:FFE3	Timer Channel 6
$FFE4:FFE5	Timer Channel 5
$FFE6:FFE7	Timer Channel 4
$FFE8:FFE9	Timer Channel 3
$FFEA:FFEB	Timer Channel 2
$FFEC:FFED	Timer Channel 1
$FFEE:FFEF	Timer Channel 0
$FFF0:FFF1	Real Time Interrupt
$FFF2:FFF3	IRQ* Pin
$FFF4:FFF5	XIRQ* Pin
$FFF6:FFF7	SWI
$FFF8:FFF9	Unimplemented Opcode Trap
$FFFA:FFFB	COP Failure (Reset)
$FFFC:FFFD	Clock Monitor Fail (Reset)
$FFFE:FFFF	RESET*

What Happens When an IRQ Happens

- IRQ detected
- CPU finished current instruction
- CPU pushes all registers onto the STACK
  - this includes the Program Counter, which is where the program will return to after the ISR
- CPU grabs the ISR address from the "Vector Table" and loads into the PC
- ISR is executed and completes
- CPU pulls all registers from the STACK
  - this includes the Program Counter
- CPU returns to executing code as it was before

Terminology

- Request: IRQ occurs and indicates to CPU that it needs attention
- Pending: IRQ that is waiting to be serviced
- Service: Process of executing the ISR
- Latency: The delay between the Request and the beginning of Service

Finish executing current instruction
Storing CPU registers to STACK
Retrieve ISR address from Vector Table
+_____
Latency

Enabling Interrupts

- Upon External Reset, IRQ's are disabled
- "Maskable" IRQ's are enabled by:

  1) A Global Interrupt Mask

  - these are the X and I bits in the CCR
  - initially set, we clear then in our code if we want IRQ's
  - this enables a group of IRQ's

  2) A Local Interrupt Mask

  - each individual IRQ has an enable bit
  - this must be enabled in the configuration registers
  - also called "device interrupt enable“

- Some IRQ's cannot be disabled, these are called "Non-Maskable" IRQ's (e.g. RESET)

Using an IRQ

- There are ALWAYS 3 things that must be done to use an IRQ

  1) Initialize the Vector Address for that IRQ

  2) Write ISR

  3) Enable the IRQ

Using an IRQ

- Step 1: Initialize “Interrupt Vector Table”
- Step 2: Write Interrupt Service Routine
  - We can put them after our main code or of a place of the programmers choosing

```
My_ISR
        code…..
        RTI
```

- RTI = "Return from Interrupt" - indicates end of service routine (similar to RTS). However it indicates that all of the registers must be pulled from STACK

- an ISR can call subroutines

Interrupt Vector Initialization

- Interrupt vector is address of start of particular ISR and must be initialized
- An example:

```
invectlc.asm            Assembled with CASM   05/28/1998  01:33  PAGE 1

                    1  ; Initializing M68HC12 interrupt vectors
                    2  ; Initialize Timer Channel 0 Interrupt vector
  0000              3  TC0:       EQU     $FFEE   ; Address of the vector
  0000              4  PROG:      EQU     $E000   ; Program location
                    5  ; . . .
  E000              6             org     PROG
                    7  ; Main program
  E000 A7           8             nop
                    9  ; . . .
                   10  ; The interrupt service routine starts with a label
                   11  ; at the first op code to be executed.
                   12  TC0ISR:
  E001 A7          13             nop
                   14  ; . . .
                   15  ; The isr ends with a RTI
  E002 0B          16             rti
                   17  ; . . .
                   18  ; Locate the vector
  FFEE             19             ORG     TC0
  FFEE E001        20             DW      TC0ISR  ; The label is the address
                   21                             ; of the isr
```

Using an IRQ

- **Step 3: Enable Interrupt**

  I-bit - this is a Global Mask for all "Maskable" Interrupts in the HC12

  I=1 (Maskable IRQ's are Disabled, default @ reset)
  I=0 (Maskable IRQ's are Enabled, we manually clear this bit to enable IRQ's

  to alter the I-bit, we write code to clear the bit:

  to clear/enable `ANDCC #%11101111`
  or
  `CLI`

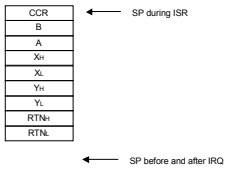
  to set/disable `ORCC #%00010000`
  or
  `SEI`

Enable Interrupt (Contd.)

- **When an ISR is called, the I-bit is automatically set so that other interrupts don't occur**
- **We can choose to allow interrupts by enabling them first thing in the ISR, but BE CAREFUL**
- **"Non-Maskable" Interrupts are always enabled**
- **Local Enable: each of the "Maskable IRQ's" have a local enabled (aka device IRQ enable)**
  - See Tables 8-1, 8-2 (SHE, old version) "Local enable bit" for value

The Stack and IRQ's

- **Upon IRQ, the following 9 bytes are pushed onto the stack:**



- **The RTI command PULLS this information off the stack**

Interrupt Priorities

- **Interrupt priorities fixed in h/w; *see Table 8-4 (SHE, old version)***
- **However, any single interrupting source can be elevated to highest-priority status**
  - **Set by writing low byte of desired vector to HPRIO control register**
  - **But, should only be done when interrupts masked via I=1 in CCR**
  - **Default on CPU reset is HPRIO = $F2 giving IRQ preference**

Setting Interrupt Priorities

Priority	MC68HC912B32 Maskable Interrupt Source	HPRIO Value to promote
1	Highest according to HPRIO	
2	IRQ* Pin	$F2
3	Real Time Interrupt	$F0
4	Timer Channel 0	$EE
5	Timer Channel 1	$EC
6	Timer Channel 2	$EA
7	Timer Channel 3	$E8
8	Timer Channel 4	$E6
9	Timer Channel 5	$E4
10	Timer Channel 6	$E2
11	Timer Channel 7	$E0
12	Timer Overflow	$DE
13	Pulse Accumulator Overflow	$DC
14	Pulse Accumulator Input Edge	$DA
15	SPI Serial Transfer Complete	$D8
16	SCI-0 Serial System	$D6
17	Reserved	$D4
18	Analog-to-Digital Converter	$D2
19	BDLC	$D0
20	Reserved	$CE

Raise Timer Channel 2 to the highest priority position

```
HPRIO:    EQU     $1F         ; HPRIO address
TC2VECT:  EQU     $FFEA       ; Channel 2 Vector
;         . . .
; Mask interrupts while setting HPRIO
          sei                 ; Set I-bit
; Raise Timer Channel 2 to the highest priority
          ldd     #TC2VECT
          stab    HPRIO
          cli
; Clear interrupt mask
```


NMI Sources

- **Six sources (from highest to lowest priority)**

	Priority	Nonmaskable Interrupt Source	Vector Address	Enable Bit
High	1	RESET*	$FFFE:FFFF	None
	2	Clock Monitor Fail	$FFFC:FFFD	CME, DISR
	3	COP Failure	$FFFA:FFFB	DISR
	4	Unimplemented Opcode Trap	$FFF8:FFF9	None
	5	Software Interrupt SWI	$FFF6:FFF7	None
Low	6	XIRQ*	$FFF4:FFF5	X

NMI Sources

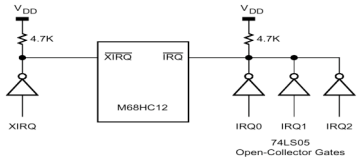
- 1. RESET*
- 2. Clock monitor fail (if CPU's clock signals slow down or fail)
- 3. Computer Operating Properly (COP) failure
  - Watchdog timer used in dedicated applications to help when power surges or programming errors lead to program getting "lost";
  - When COP enabled, program must periodically pulse the COPRST reg. by writing $55 followed by $AA before at specified interval expires;
  - Timeout period controllable in COPCTL control register, from ~1 ms to ~1 sec.
  - Often disabled in development board s/w, but potent for real embedded systems.

NMI Sources

- 4. Unimplemented opcode trap (happens when "lost" program starts executing data)
- 5. SWI (one-byte indirect branch to ISR via fixed vector location for SWI)
- 6. XIRQ* (more later)
  - Initially masked to allow time to initialize SP, etc.
  - Then can unmask by clearing X bit in CCR via ANDCC #%10111111
  - Once unmasked, cannot be masked again except on reset or during execution of XIRQ*'s ISR (when masked and unmasked automatically to avoid nested NMI)

External Interrupt Sources

- IRQ* signal generated by external device; may choose level-activate (default: good for shared wired-OR line) or negative-edge-triggered response (unshared) via setting in INTCR reg.
- Polling can be used w/ multiple devices on IRQ* line; HC12 has no INTA signal to reset interrupt request, so if needed by external device then output port bit may be used



IRQ* External Interrupt

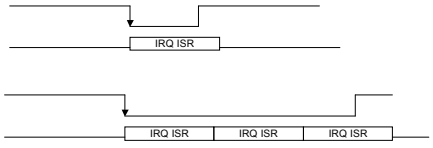
- An external IRQ available to the HC12
  - is a physical pin on the device
  - can be used for sensors, external signals, buttons
- Sensitivity
  - can be "Level" or "Negative Edge" sensitive
  - we use the IRQE bit in the INTCR register to configure this

    IRQE = 0 "Active Low"

    IRQE = 1 "Negative Edge Active"

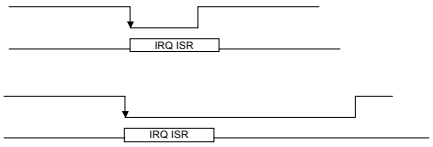
IRQ* Active Low Operation

- When the ISR is executing, IRQ's are disabled
- Sometimes, the IRQ* line will still be low when the completed
  - this can be caused by a human pushed button (slow) or a slower peripheral
  - this can cause the IRQ* to trigger again when the ISR is completed



IRQ* Negative Edge Operation

- IRQ Request on falling edge will only occur once on edge
  - sometimes better for a one-time event like a button



IRQ* Details

"Global Enable" - I-bit in CCR

I-bit 1 = Disabled (default or SEI)
I-bit 0 = Enabled (manually clear with CLI)

"Local Enable" - IRQEN bit of the INTCR Register

IRQEN 0 = Disabled (default)
IRQEN 1 = Enabled

"Vector Address"

$FFF2:$FFF3

"Priority"

highest of Maskable IRQ's, default value in HPRIO

IRQ* Example 1

- **Latch the contents of Port A on the falling edge of IRQ* and store in "Temp" (this will make the HC12 into an 8-bit D-flip-flop)**

Step 1 : Initialize the Vector Table

```
IRQ_Vector      EQU     $FFF2
…

                ORG IRQ_Vector
                DW My_ISR
```

Step 2 : Write ISR (Registers are already preserved so don't need to push)

```
My_ISR:
        LDAA    PortA
        STAA    Temp
        RTI
```


IRQ* Example 1

Step 3 : Enable Interrupt

a) configure IRQ
b) Local enable
c) Global enable

```
        SEI                     ; disable Maskable IRQ's
        BSET    INTCR, IRQE     ; setup IRQ* to edge sensitive
        BSET    INTCR, IRQEN    ; Local enable IRQ*
        CLI                     ; Global enable for IRQ*

INTCR   EQU     $001E
IRQE    EQU     %10000000
IRQEN   EQU     %01000000
```

(See SHE pp 202 for more information on INTCR)

IRQ* Example 2

- **We have a sensor at a door and want to flash an LED if the door is open**
  - **We can use IRQ* in "Level Sensitive" Mode**
  - **IRQ will continue to trigger as long as the signal is asserted**

Step 1 : Initialize the Vector Table

```
IRQ_Vector      EQU     $FFF2
…

                ORG IRQ_Vector
                DW My_ISR
```

Step 2 : Write ISR (Registers are already preserved so don't need to push)

```
My_ISR:
        LDAA    #~LED1
        STAA    PortA
        JSR     Delay
        LDAA    #LED1
        STAA    PortA
        RTI
```

IRQ* Example 2

Step 3 : Enable Interrupt

a) configure IRQ
b) Local enable
c) Global enable

```
    SEI                         ; disable Maskable IRQ's
    BCLR    INTCR, IRQE         ; setup IRQ* to level sensitive
    BSET    INTCR, IRQEN        ; Local enable IRQ*
    CLI                         ; Global enable for IRQ*

INTCR   EQU     $001E
IRQE    EQU     %10000000
IRQEN   EQU     %01000000
```

(See SHE pp 202 for more information on INTCR)

XIRQ* External Interrupt

- **XIRQ* Details**
  - **An external IRQ available to the HC12**
  - **Is a physical pin on the device**
  - **Psuedo Non-Maskable Interrupts: we CAN enable it, but only once. After we enable it, it is always going**
  - **Global EN = X-bit**
    X=1, disabled (default) **(ANDCC #%1011 1111)**
    X=0, enabled
  - **Local EN = NONE**
  - **during XIRQ ISR, CPU will disable other XIRQ Interrupts (X=1), we can't control this**
  - **after XIRQ ISR, CPU will enabled XIRQ interrupts (X=0) automatically**

XIRQ* External Interrupt

- **XIRQ* Details**
  - **Vector = $FFF4:$FFF5**
  - **Priority = #6 : non-Maskable > Maskable, Priority can't be changed or promoted**
- **XIRQ_L Sensitivity**
  - **Level Sensitive Only (active LOW)**
  - **Typically used for gross failures like "low power" or "fire alarm"**

XIRQ* Example 1

Step 1 : Initialize the Vector Table

```
XIRQ_Vector     EQU     $FFF4
…

                ORG XIRQ_Vector
                DW My_ISR
```

Step 2 : Write ISR (Registers are already preserved so don't need to push)

```
My_ISR:
        ; Code
        RTI
```

Step 3 : Enable Interrupt

```
ANDCC #%1011 1111
```

Note: a) no configuration needed
b) Local enable = NONE
c) Global enable = X-bit

NOTE: Do this right before your main program loop. Once you enable it, interrupts may start occurring immediately

Advanced Interrupt

- **What to do in s/w when waiting on an interrupt:**
  - **Spin loop spin:bra spin**
  - **Use WAI instruction to wait on interrupt (pushes regs. on stack in prep. for later interrupt to reduce its latency, then puts CPU into WAIT mode to conserve power)**
  - **Use STOP instruction to stop HC12 clocks and dramatically conserve power; S bit in CCR must be 0 for instruction to operate**

WAI

- **WAI - "Wait for Interrupt"**

  - Low Power Mode

  - This instruction : 1) Stacks the Return Address; 2) Stacks the CPU Registers; 3) Stops executing instructions and waits for an IRQ

  - System Clocks are all still running

  - Good for Real Time programs where main loop does nothing, all functionality is handled by IRQs

WAI Example

```
; Example of the WAI instruction in
; a foreground job.
;
foreground:
; Here is the code to be done in the
; foreground.  When all is complete,
; wait for the next interrupt.
;           . . .
          wai
; When come out of the interrupt branch
; back to the foreground job.
          bra       foreground
```


STOP

- **STOP - "Stop CPU"**

  - Low Power Mode, similar to WAI

  - This instruction :1) Stacks the Return Address; 2) Stacks the CPU Registers; 3) Stops executing instructions and waits for an IRQ or Reset

  - CPU Clock is STOPPED

  - Takes more time to recover after IRQ/RESET due to waiting for X-tal to reach full-speed

  - We enabled this in the CCR, S-bit: S=1, disabled (default), instruction treated as NOP; S=0, enabled