**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 with 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

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

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 them 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:

```assembly
; My_ISR

; RTI

; An example

; void My_ISR(void)

; This example shows how to initialize an ISR

; The interrupt vector is located at the address 0x0003

; The interrupt vector must be initialized before the ISR is called

; The ISR is called when an interrupt occurs

; The ISR should return to the calling routine

; The calling routine must save the registers before the ISR is called

; The ISR must restore the registers before returning to the calling routine

; The ISR must not modify the stack pointer

; The ISR must not modify the program counter

; The ISR must not modify the interrupt mask

; The ISR must not modify the interrupt vector

; The ISR must not modify the interrupt vector table
```
**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

  Dr. Tao Li 13

**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

  Dr. Tao Li 14

---

**The Stack and IRQ’s**

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

  - SP before: L Y H X L X H A B
  - SP during ISR: RTN L RTN H
  - RTI command PULLS this information off the stack

  Dr. Tao Li 15

**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

  Dr. Tao Li 16
### Setting Interrupt Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Interrupt Source</th>
<th>HPRI 16-bit Value</th>
<th>HPRI 8-bit Value</th>
<th>HPRI Address</th>
<th>HPRIO</th>
<th>TC2VECT</th>
<th>sets</th>
<th>Clears</th>
<th>Vector 24-bit Address</th>
<th>EQU</th>
<th>Vector 8-bit Value</th>
<th>EQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESET*</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>2</td>
<td>Clock Monitor Fail</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>3</td>
<td>COP Failure</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>4</td>
<td>Unimplemented Opcode Trap</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>5</td>
<td>SWI</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>6</td>
<td>XIRQ*</td>
<td>0000</td>
<td>FF</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>FF000000000000000000000000000000</td>
<td>0000</td>
<td>FFFF</td>
<td>FFFF</td>
</tr>
</tbody>
</table>

### NMI Sources

- **Six sources (from highest to lowest priority)**
  - **High**
    - 1. `RESET*` - Unmaskable, address FFFF FFFF
    - 2. Clock Monitor Fail - Unmaskable, address FFFF FFFF
    - 3. COP Failure - Unmaskable, address FFFF FFFF
    - 4. Unimplemented Opcode Trap - Unmaskable, address FFFF FFFF
    - 5. SWI - Unmaskable, address FFFF FFFF
  - **Low**
    - 6. XIRQ* - Maskable, address FFFF FFFF

**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.

- **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

  \[ \text{IRQE} = 0 \quad \text{"Active Low"} \]
  \[ \text{IRQE} = 1 \quad \text{"Negative Edge Active"} \]

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**

**Step 1 : Initialize the Vector Table**

<table>
<thead>
<tr>
<th>IRQ_Vector EQU</th>
<th>$FFF2</th>
</tr>
</thead>
</table>

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

- **My_ISR:**
  - LDAA PortA
  - STAA Temp
  - ORG IRQ_Vector
  - DW My_ISR
  - JSR Delay
  - LDAA #LED1
  - STAA PortA
  - RTI

**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*

**Example Code**

- ORG IRQ_Vector
- DW My_ISR
- RTI

---

**IRQ* Example 2**

**Step 1 : Initialize the Vector Table**

<table>
<thead>
<tr>
<th>IRQ_Vector EQU</th>
<th>$FFF2</th>
</tr>
</thead>
</table>

**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

**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*

**Example Code**

- ORG IRQ_Vector
- DW My_ISR
- RTI

---

**Remarks**

- **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)**

- **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

- **We can use IRQ* in "Level Sensitive" Mode**
  - IRQ will continue to trigger as long as the signal is asserted
### IRQ* Example 2

**Step 3 : Enable Interrupt**

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

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

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

(See S1E 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
- Pseudo 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* 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**

```assembly
XIRQ_Vector EQU $FFF4
org XIRQ_Vector
```

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

```assembly
My_ISR:
; Code
RTI
```

**Step 3 : Enable Interrupt**

```assembly
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
• 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 - "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

• 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