EEL 4744C: Microprocessor Applications

Lecture 7

Part 2

M68HC12 Interrupt

Dr. Tao Li

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

Interruptions (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

Dr. Tao Li

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

Dr. Tao Li

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

Dr. Tao Li

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

Dr. Tao Li
What Happens When an IRQ Happens

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

Enabling Interrupts

1. Upon External Reset, IRQ's are disabled
2. "Maskable" IRQ's are enabled by:
   1) A Global Interrupt Mask
      - these are the X and L 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"
3. Some IRQ's cannot be disabled, these are called "Non-Maskable" IRQ's (e.g. RESET)

Using an IRQ

1. Step 1: Initialize "Interrupt Vector Table"
2. 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

Terminology

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

Using an IRQ

1. 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

Interrupt Vector Initialization

1. Interrupt vector is address of start of particular ISR and must be initialized
2. An example:

```
My_ISR code:
    RTI

// Interrupt Vector Table
interrupt_vector:  dw 003Fh

// ISR
interrupt_service:  proc
    ; The interrupt service routine starts with a label
    ; It can finish up code to be initialized.
    ; Offset 0,0
    ret
\interrupt_service
```

2
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

<table>
<thead>
<tr>
<th>Priority</th>
<th>Interrupt Name</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESET*</td>
<td>1FFE FFFF</td>
</tr>
<tr>
<td>2</td>
<td>Clock Monitor Fail</td>
<td>1FFC FFFF</td>
</tr>
<tr>
<td>3</td>
<td>COP Failure</td>
<td>1FFB FFFF</td>
</tr>
<tr>
<td>4</td>
<td>Unimplemented Opcode Trap</td>
<td>1FFA FFFF</td>
</tr>
<tr>
<td>5</td>
<td>Software Interrupt SWI</td>
<td>1FF8 FFFF</td>
</tr>
<tr>
<td></td>
<td>XIRQ*</td>
<td>1FF4 FFFF</td>
</tr>
</tbody>
</table>

NMI Sources

- Six sources (from highest to lowest priority)

[Table with details of NMI Sources]
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 \texttt{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

\begin{center}
\begin{tikzpicture}
\node (irq1) at (0,0) {IRQ1};
\node (irq2) at (0,-1) {IRQ2};
\node (interrupt) at (0,-2) {Interrupt};
\draw [->] (irq1) -- (interrupt);
\draw [->] (irq2) -- (interrupt);
\end{tikzpicture}
\end{center}

- Sensitivity
  - can be "Level" or "Negative Edge" sensitive
  - we use the IRQE bit in the INTCR register to configure this
    \begin{itemize}
    \item IRQE = 0 “Active Low”
    \item IRQE = 1 “Negative Edge Active”
    \end{itemize}

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

\begin{center}
\begin{tikzpicture}
\node (irq1) at (0,0) {IRQ1};
\node (irq2) at (0,-1) {IRQ2};
\node (interrupt) at (0,-2) {Interrupt};
\draw [->] (irq1) -- (interrupt);
\draw [->] (irq2) -- (interrupt);
\end{tikzpicture}
\end{center}

\section*{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

\begin{center}
\begin{tikzpicture}
\node (irq1) at (0,0) {IRQ1};
\node (irq2) at (0,-1) {IRQ2};
\node (interrupt) at (0,-2) {Interrupt};
\draw [->] (irq1) -- (interrupt);
\draw [->] (irq2) -- (interrupt);
\end{tikzpicture}
\end{center}
**IRQ* Details**

"Global Enable" - I-bit in CCR

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

"Local Enable" - IRQEN bit of the INTCR Register

- IRQEN 0 = Disabled (default)
- IRQEN 1 = Enabled

"Vector Address"

- $FFF2:$FFFF

"Priority"

- highest of Maskable IRQ's, default value in HPRIO

Dr. Tao Li

---

**IRQ* Example 1**

**Step 1 : Initialize the Vector Table**

```
IRQ_Vector EQU $FFF2
```

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

```
My_ISR:
  LDAA PortA
  STAA Temp
  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
CLI ; Global enable for IRQ*
```

```
INTCR EQU $001E
IRQE EQU %10000000
IRQEN EQU %01000000
(See SHE pp 202 for more information on INTCR)
```

Dr. Tao Li

---

**IRQ* Example 2**

**Step 1 : Initialize the Vector Table**

```
IRQ_Vector EQU $FFF2
```

**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
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)
```

Dr. Tao Li

---

**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

Dr. Tao Li
**XIRQ* External Interrupt**

- **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"

**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

**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

**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**

```asm
; 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