Interrupts (Part 1)
Responding to events

- We’ve seen two ways to detect events (like an input signal being asserted), and respond to them:
  - Polling (reading) the input signal directly
    - This wastes a lot of CPU cycles just polling
    - CPU can’t do anything else
  - Let the input signal set a hardware flag (i.e., a flip-flop), and then test the flag occasionally
    - Don’t need to poll so quickly
    - Even if you miss the actual moment that the signal came in, you can test the flag later and tell that the event happened
    - However, you still need to poll the flag; also there may be some delay between the time that the event occurred and when you actually test the flag

- A better (third) way: use interrupts
  - No time wasted polling
  - Small (and predictable) delay between event and your response
What is an interrupt?

• An event that causes the CPU to stop the program it is currently running and run a function to service the event

• When it is done servicing the event, it goes back to what it was doing

• Good analogy
  – You are reading a book and the phone rings
  – You stop reading and put a bookmark at the current place, then answer the phone
  – After you are done talking you go back to the place in the book where the bookmark was placed and resume reading
Interrupts

• A way to respond to an external event (i.e., flag being set) without polling

• Process:
  – H/W senses flag being set
  – Automatically transfers control to s/w that “services” the interrupt
  – When done, H/W returns control to wherever it left off

• Advantages
  – Transparent to user – cleaner code
  – Computer doesn’t waste time polling

• The main program can be interrupted anywhere (after the current instruction finishes)

• When the ISR is done, the main program resumes at the next instruction
Interrupts

• The main program should be completely unaware that an interrupt has occurred

• For this to happen, there are two big things that the interrupt system must do:
  – The system must save the “state” of the current program, and restore it when it returns
  – The system must return control to the main program at the instruction following the one where the interrupt occurred

• This is easier than is sounds - the “state” of the current program is just the values in the CPU registers (including the program counter)\(^1\)
  – The value in the PC is just the address of the next instruction

• So all the interrupt system needs to do is to save all the CPU registers on the stack, and restore them when done

\(^1\)Of course this assumes that the ISR doesn’t change the memory that the main program is using
Interrupts

• When an interrupt occurs, the interrupt system pushes all the CPU registers on the stack (9 bytes total)

• It then transfers control to the interrupt service routine (ISR)
  – The ISR can use all the CPU registers without fear of interfering with the main program

• The last machine code instruction in an ISR is always “RTI” (return from interrupt)
  – This pulls all the CPU registers off the stack

• Since the program counter is one of registers that is restored, execution will resume at the place where the main program was interrupted

• Note that is some overhead in using interrupts
  – In addition to the instructions in the ISR itself, it takes some time to push and pull the registers
  – If you really need to respond quickly to an event (i.e., a flag being set), polling a flag is actually faster (of course, then the CPU can only do that one thing)
Example

• Suppose that the CPU is executing the following instruction segment and an interrupt occurs when the “inca” instruction is being executed. What will be the contents of the stack?

```assembly
org   $2000
lds   #$2000
ldd   #$1234
ldx   #$5678
ldy   #$9ABC
tap   ; this does (A) => CCR
inca
nop
```
Example (continued)

- We need to determine the return address that will be pushed on the stack, when the interrupt happens.
- From the disassembler, we can find the machine code and where the instructions are stored in memory:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Address</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5485</td>
<td>a002000 CF20 00</td>
<td>5486</td>
<td>a002003 CC12 34</td>
</tr>
<tr>
<td>5487</td>
<td>a002006 CE56 78</td>
<td>5488</td>
<td>a002009 CD9A BC</td>
</tr>
<tr>
<td>5489</td>
<td>a00200C B702 00</td>
<td>5490</td>
<td>a00200E 42</td>
</tr>
<tr>
<td>5491</td>
<td>a00200F A7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This will be the return address

Interrupt occurs here
Example (continued)

- The CPU will finish the execution of the "inca" instruction.
- Contents of registers:
  - The return address to be pushed onto the stack is $200F
  - A = $12 + 1 = $13
  - B = $34
  - X = $5678
  - Y = $9ABC
  - CCR = $12 (caused by "tap" instruction)
- Initially SP = $2000
- After the interrupt system pushes 9 bytes onto the stack, SP points to $1FF7
Enabling interrupts

• The I-bit in the condition code register (bit 4) enables the interrupt system
  – Think of this as a switch that allows the interrupt signal to get through
• You should set the I-bit to zero to enable interrupts
  – e.g., using the CLI instruction

• When an interrupt occurs
  – After pushing the CPU registers, the interrupt system disables additional interrupts (i.e., sets I=1)
  – This prevents other interrupts from occurring till the ISR is done
  – When the ISR finishes, interrupts are re-enabled (this is done automatically by restoring the CCR)
Interrupt signal path

When enabled, takes control of CPU, and does these things:
1. Save registers on stack
2. Fetch address of ISR
3. Disable interrupts

• There are many devices that can cause interrupts
• Each sets a flag
• Each has its own interrupt enable switch

The I-bit is the “gatekeeper” for all the maskable interrupts

In addition to the internal subsystems, /IRQ is an external pin that can generate an interrupt

The “/” means that the signal is asserted when low

device interrupt enables
device flags

devices or subsystems

e.g., RTI system
e.g., A/D converter

Microcomputer Architecture and Interfacing  Colorado School of Mines  Professor William Hoff
Example – RTI system

RTIF is set when RTI event occurs ... write a 1 to RTIF to clear it.

Write a 1 to RTIE to enable RTI interrupts.

Reset

CRGFLG

CRGINT
ISR Address

• How does the interrupt system know where to transfer control to when an interrupt occurs?
  – Namely, what is the starting address of the interrupt service routine (ISR)?

• There are two approaches
  – One type of microcontroller architecture (e.g., PIC) simply uses a single fixed address for every interrupt
    • The system always transfers control to that ISR
    • The programmer has to figure out which device caused the interrupt (by checking flags, for example)
  
  – In the other type of architecture, the interrupt system knows which device caused the interrupt (since its interrupt request signal was asserted)
    • The system can transfer control to a predefined (ISR) address for each type of interrupt (the method used by the Intel 8051)
    • Or, the system looks up the address of the ISR in a predefined location in a table (the method used by the HCS12)
Interrupt Vector Table

- The system looks up the address of the ISR in a predefined location in a table
  - The address is called the “vector”
  - The programmer must put the address of the ISR in that location

- In the HCS12, the vector table is in locations $FF80 through $FFFF
- Example – RTI system
  - The vector for the RTI system is at $FFF0:$FFF1

When an RTI interrupt occurs, the interrupt system fetches the ISR address from locations $FFF0:$FFF1.

Control is transferred to this address (e.g., $4103)

Starting address of ISR (e.g., $36 is the machine code for the 1st instruction in the ISR)
Complete Vector Table for the HCS12

- Interrupt sources are prioritized, with the highest priority source, Reset, at the top of the table.

- Must initialize vector with your ISR address, e.g.:

  ```
  org $FFF0
  dc.w myRTIISR
  ```

  This will put the starting address of your ISR routine into location $FFF0:$FFF1

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Interrupt source</th>
<th>CCR mask</th>
<th>Local Enable</th>
<th>HPRIO value to elevate to highest I bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFFE</td>
<td>Reset</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFC</td>
<td>Clock monitor reset</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFA</td>
<td>COP failure reset</td>
<td>none</td>
<td>COP rate selected</td>
<td>-</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Unimplemented instruction trap</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFA</td>
<td>SWI</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFA</td>
<td>XIRQ</td>
<td>X bit</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFB</td>
<td>Real time interrupt</td>
<td>I bit</td>
<td>INTCR(IRQEN)</td>
<td>$F2</td>
</tr>
<tr>
<td>$FFFC</td>
<td>Timer channel 0</td>
<td>I bit</td>
<td>RTICTL(RTIE)</td>
<td>$F0</td>
</tr>
<tr>
<td>$FFEC</td>
<td>Timer channel 1</td>
<td>I bit</td>
<td>TMSK1(C0I)</td>
<td>$EE</td>
</tr>
<tr>
<td>$FFEA</td>
<td>Timer channel 2</td>
<td>I bit</td>
<td>TMSK1(C1I)</td>
<td>$EC</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer channel 3</td>
<td>I bit</td>
<td>TMSK1(C2I)</td>
<td>$EA</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer channel 4</td>
<td>I bit</td>
<td>TMSK1(C3I)</td>
<td>$E8</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer channel 5</td>
<td>I bit</td>
<td>TMSK1(C4I)</td>
<td>$E4</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer channel 6</td>
<td>I bit</td>
<td>TMSK1(C5I)</td>
<td>$E2</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer channel 7</td>
<td>I bit</td>
<td>TMSK1(C6I)</td>
<td>$E0</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Timer overflow</td>
<td>I bit</td>
<td>TMSK2(T0I)</td>
<td>$DE</td>
</tr>
<tr>
<td>$FFFA</td>
<td>Pulse accumulator overflow</td>
<td>I bit</td>
<td>PACTL(PAOVI)</td>
<td>$DC</td>
</tr>
<tr>
<td>$FFFA</td>
<td>SPI serial transfer complete</td>
<td>I bit</td>
<td>PACTL(PAII)</td>
<td>$DA</td>
</tr>
<tr>
<td>$FFFD</td>
<td>SCI0</td>
<td>I bit</td>
<td>SCOCR2(TIE,TCIE,RIE,ILIE)</td>
<td>$D6</td>
</tr>
<tr>
<td>$FFFD</td>
<td>SCI1</td>
<td>I bit</td>
<td>SC1CR2(TIE,TCIE,RIE,ILIE)</td>
<td>$D4 (1,3,4)</td>
</tr>
<tr>
<td>$FFFA</td>
<td>ATD0 or ATD1</td>
<td>I bit</td>
<td>ATDxCTL2(ASCIE)</td>
<td>$D2</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 0 wakeup</td>
<td>I bit</td>
<td>C0IER(WUPIE)</td>
<td>$D0 (1*,2,2*)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>Key wakeup J or H</td>
<td>I bit</td>
<td>KWIEM[7:0] and KWIEM[7:0]</td>
<td>$CE (1,3,4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>Modulus down counter underflow</td>
<td>I bit</td>
<td>MCCTL(MCZI)</td>
<td>$CC</td>
</tr>
<tr>
<td>$FFFD</td>
<td>Pulse accumulator B overflow</td>
<td>I bit</td>
<td>PBCTL(PBOVI)</td>
<td>$CA</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 0 errors</td>
<td>I bit</td>
<td>C0IER(RWRNIE,TWRNIE, RERRIE,TERRIE,BOFFIE,OVRIE)</td>
<td>$C8 (2*,3,4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 0 receive</td>
<td>I bit</td>
<td>C0IER(RXFIE)</td>
<td>$C6 (2*,3,4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 0 transmit</td>
<td>I bit</td>
<td>C0TCR(TXIE[2:0])</td>
<td>$C4 (2*,3,4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>CGK lock and limp home</td>
<td>I bit</td>
<td>PLLCR(LOCKIE, LHIE)</td>
<td>$C2 (3,4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>IIC Bus</td>
<td>I bit</td>
<td>IBCR(IBIE)</td>
<td>$CO (4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 1 wakeup</td>
<td>I bit</td>
<td>C1IER(WUPIE)</td>
<td>$BE (4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 1 errors</td>
<td>I bit</td>
<td>C1IER(RWRNIE,TWRNIE, RERRIE,TERRIE,BOFFIE,OVRIE)</td>
<td>$BC (4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 1 receive</td>
<td>I bit</td>
<td>C1IER(RXFIE)</td>
<td>$BA (4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>MSCAN 1 transmit</td>
<td>I bit</td>
<td>C1TCR(TXIE[2:0])</td>
<td>$B8 (4)</td>
</tr>
<tr>
<td>$FFFD</td>
<td>Reserved</td>
<td>I bit</td>
<td>Reserved</td>
<td>$B6</td>
</tr>
<tr>
<td>$FFFD</td>
<td>Reserved</td>
<td>I bit</td>
<td>Reserved</td>
<td>$80-SB4</td>
</tr>
</tbody>
</table>
Interrupt Sequence

• When the CPU receives an interrupt:
  – Finishes current instruction
  – Pushes CPU registers on stack
  – Disables further interrupts by setting the I bit
  – Identify source of the interrupt – if more than one is pending, figure out which is highest priority
  – Fetch corresponding ISR address from the vector table
  – Start executing the ISR

• When the ISR is done
  – Execute the RTI instruction, which is the last instruction in the ISR
  – This causes CPU registers to be restored from the stack
  – This automatically re-enables interrupts and transfers control back to the main program
Summary / Questions

- Interrupts use special hardware to detect an interrupt event and then transfer control to an “interrupt service routine”.

- How does the CPU know where to return to the main program?

- Is it possible to implement an embedded system without using interrupts?