Interrupts
Interrupts

• An interrupt is typically an external event that causes the Microcontroller to suspend its current task

• A small piece of software (an interrupt service routine) is executed, then the microcontroller resumes what it was previously doing

• Interrupts are useful for performing actions within a very short time period of the event occurring. Most true ‘real-time’ systems are interrupt driven
Normal(?) Program Operation

```
Power On RESET
(Initialization Routine)

Main Program

Cycle continuously
```
Reset Interrupt

Power On RESET (Initialization Routine)

and start over again.

Main Program

Event triggers reset signal.

Stop program sequence

Monday, 25 March 13
Interrupt Service Routine

Figure 3.7 interrupt
Interrupt Handling

An Interrupt signal can be defined as a hardware signal that initiate an event. It can be generated by external devices or internal processes.

Interrupt Process:

1. Finish current instruction.
2. Save PC in the interrupt return location
3. Load PC with the contents of interrupt handler location
Multiple Interrupt

Single Interrupt

1. Finish current instruction.
2. Save PC in the interrupt return location
3. Load PC with the contents of interrupt handler location

Interrupt

CPU

1

PC

2

Interrupt Return Loc.

Multiple Interrupts

Interrupt n

CPU

1

Interrupt Handler Loc.

1. Check for interrupt enable
2. Check to see if higher priority interrupt is running.
3. Finish current instruction.
4. Save PC in the interrupt return location
5. Load PC with the contents of interrupt handler location

Interrupt

Return Loc.

Interrupt

Handler Loc.

PC

1

2

3
Interrupt Vectors

• All resets and interrupts use vectors

• The vector table lists the start address of the corresponding interrupt service routine

• When an interrupt occurs, the CPU fetches the corresponding vector (address) and puts it in the PC. The interrupt service routine is therefore executed
Pseudo Vectors

• The vector table is fixed (it is part of the 68HC11 chip)

• But any address can be loaded into the vector table (to point to an ISR)

• However, there are circumstances under which the absolute ISR address may be unknown, or not fixed (e.g. during development)

• In these cases, a pseudo-vector may be used. This is a jump table located in RAM which has the potentially variable ISR addresses stored in it. This is not necessary for the EVBPlus board
Memory Map
### TABLE B.2 INTERRUPT VECTOR ASSIGNMENTS

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Interrupt Source</th>
<th>CCR Mask Bit</th>
<th>Local Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFC0, C1 – FFD4, D5</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFD6, D7</td>
<td>SCI serial system • SCI receive data register full • SCI receiver overrun • SCI transmit data register empty • SCI transmit complete • SCI idle line detect</td>
<td>I</td>
<td>RIE RIE TIE TCIE ILIE</td>
</tr>
<tr>
<td>FFD8, D9</td>
<td>SPI serial transfer complete</td>
<td>I</td>
<td>SPIE</td>
</tr>
<tr>
<td>FFDA, DB</td>
<td>Pulse accumulator input edge</td>
<td>I</td>
<td>PAII</td>
</tr>
<tr>
<td>FFDC, D0</td>
<td>Pulse accumulator overflow</td>
<td>I</td>
<td>PAOVI</td>
</tr>
<tr>
<td>FFDE, DF</td>
<td>Timer overflow</td>
<td>I</td>
<td>TOI</td>
</tr>
<tr>
<td>FFE0, E1</td>
<td>Timer input capture 4/output compare 5</td>
<td>I</td>
<td>I4/O5I</td>
</tr>
<tr>
<td>FFE2, E3</td>
<td>Timer output compare 4</td>
<td>I</td>
<td>OC4I</td>
</tr>
<tr>
<td>FFE4, E5</td>
<td>Timer output compare 3</td>
<td>I</td>
<td>OC3I</td>
</tr>
<tr>
<td>FFE6, E7</td>
<td>Timer output compare 2</td>
<td>I</td>
<td>OC2I</td>
</tr>
<tr>
<td>FFE8, E9</td>
<td>Timer output compare 1</td>
<td>I</td>
<td>OC1I</td>
</tr>
<tr>
<td>FFEA, EB</td>
<td>Timer input capture 3</td>
<td>I</td>
<td>IC3I</td>
</tr>
<tr>
<td>FFEC, ED</td>
<td>Timer input capture 2</td>
<td>I</td>
<td>IC2I</td>
</tr>
<tr>
<td>FFEF, EF</td>
<td>Timer input capture 1</td>
<td>I</td>
<td>IC1I</td>
</tr>
<tr>
<td>FFF0, F1</td>
<td>Real-time interrupt</td>
<td>I</td>
<td>RTI</td>
</tr>
<tr>
<td>FFF2, F3</td>
<td>IRQ (external pin)</td>
<td>I</td>
<td>None</td>
</tr>
<tr>
<td>FFF4, F5</td>
<td>XIRQ pin</td>
<td>X</td>
<td>None</td>
</tr>
<tr>
<td>FFF6, F7</td>
<td>Software interrupt</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>FFF8, F9</td>
<td>Illegal opcode trap</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>FFFA, FB</td>
<td>COP failure</td>
<td>None</td>
<td>NOCOP</td>
</tr>
<tr>
<td>FFFC, FD</td>
<td>Clock monitor fail</td>
<td>None</td>
<td>CME</td>
</tr>
<tr>
<td>FFFE, FF</td>
<td>RESET</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
Interrupt Vectors & Pseudo Vectors

RAM JUMP TABLE
00EE
7E
C1
5A

JUMP TO PSEUDO VECTOR

RAM SERVICE ROUTINE
C15A

JMP $C15A

ROM VECTORS
FFFF2
00
FFFF3
EE

Fixed by system

Modifiable by User

00EE
PC
Interrupt Execution

• When an interrupt triggers, all CPU registers are saved to the stack (not just the PC)

• At the end of an ISR, you should always have an RTI instruction. This restores all CPU registers, and continues the main task
Interrupt Priority

- In the 68HC11, each interrupt is given a predefined ‘priority’

- This is used in the case where two interrupts fire simultaneously, or while an ISR is executing

- The interrupts are queued in order of priority, and executed as soon as the previous ISR terminates
Interrupt Masks

• You can tell the 68HC11 to ignore certain interrupts. This is known as ‘masking’ the interrupt. Interrupts that can be disabled in this way are known as ‘maskable’

• Each maskable interrupt has its own mask bit. A value of 0 tells the CPU to ignore the interrupt

• Interrupt masks are set in various control and status registers

• The I bit in the CCR can mask many interrupts
An Example

start:   JSR INIT_SERIAL
         CLI
         BRA *

serial_isr:   LDAA SCSR
              ANDA #$20
              BEQ end_isr
              LDAA SCDR
              STAA PORTB

desc:   ORG $FFD6
         FDB serial_isr

desc:   RTI

* Load the address of the ISR into the interrupt vector table.