General Features of Interrupts

- All interrupting systems must have the:
  - Ability for the hardware to request action from the computer.
  - Ability for the computer to determine the interrupt source.
  - Ability for the computer to acknowledge the interrupt.
- To arm (disarm) a device means to enable (shut off) the source of interrupts.
- To enable (disable) means to allow (postpone) interrupts at this time.

Sequence of Events During Interrupt

1. Hardware needing service makes a busy-to-done transition.
2. Flag is set in one of the I/O status registers.
   - Interrupting event sets the flag (ex., TOF=1).
   - Checks that the device is armed (ex., TOI=1).
   - Checks that interrupts are enabled (i.e., I=0).
3. Thread switch.
   - Microcomputer finishes current instruction (except rev, revw, and wav).
   - All registers are pushed onto the stack.
   - Vector address is obtained and put into the PC.
   - Microcomputer disables interrupts (i.e., sets I=1).
4. Execution of the ISR.
5. Return control back to the thread that was running.

Stack Before and After an Interrupt

- Each interrupt has 16-bit vector stored in upper 128 bytes of memory.
- There are six interrupt sources that are not maskable.
  - Power-on-reset (POR) or regular hardware RESET pin
  - Clock monitor reset
  - COP watchdog reset
  - Unimplemented instruction trap
  - Software interrupt instruction (swi)
  - XIRQ signal (if X bit in CCR = 0)
- 6812 has two external requests IRQ and XIRQ.
- Other interrupt sources include:
  - 10 key wakeup interrupts (Ports J and P)
  - 8 input capture/output compare interrupts
  - An ADC interrupt
  - 4 timer interrupts (timer overflow, RTI, 2 pulse accumulators)
  - 2 serial port interrupts (SCI and SPI)
  - 4 CAN interrupts
- Interrupts have a fixed priority, but can elevate one to highest priority using hardware priority interrupt (HPRIO) register.
- XIRQ is highest-priority and has separate vector and enable bit (X).
  - Once X bit is cleared, software cannot disable it.
  - XIRQ handler sets X and I, and restores with rti.
6812 Interrupt Vectors and Priority

<table>
<thead>
<tr>
<th>Vector</th>
<th>CW#</th>
<th>Interrupt Source</th>
<th>Enable</th>
<th>Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFF</td>
<td>0</td>
<td>Reset</td>
<td>Always</td>
<td>Always highest</td>
</tr>
<tr>
<td>$FFFC</td>
<td>1</td>
<td>COP clk monitor fail</td>
<td>COPCTL.CME</td>
<td></td>
</tr>
<tr>
<td>$FFFA</td>
<td>2</td>
<td>COP failure reset</td>
<td>COP rate selected</td>
<td></td>
</tr>
<tr>
<td>$FFFF8</td>
<td>3</td>
<td>Unimplemented instruction</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>$FFFF6</td>
<td>4</td>
<td>SWI</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>$FFFF4</td>
<td>5</td>
<td>XIRQ</td>
<td>X=0 External hardware</td>
<td></td>
</tr>
<tr>
<td>$FFFF2</td>
<td>6</td>
<td>IRQ</td>
<td>I=0 INTR.IRQEN</td>
<td></td>
</tr>
<tr>
<td>$FFFF0</td>
<td>7</td>
<td>Real time interrupt, RTIF</td>
<td>CRGINT.RTIE</td>
<td></td>
</tr>
<tr>
<td>$FFE9</td>
<td>9</td>
<td>Timer Channel 1, C1F</td>
<td>I=0 TIE.C1I</td>
<td></td>
</tr>
<tr>
<td>$FEFE</td>
<td>10</td>
<td>Timer Channel 2, C2F</td>
<td>I=0 TIE.C2I</td>
<td></td>
</tr>
<tr>
<td>$FEE8</td>
<td>11</td>
<td>Timer Channel 3, C3F</td>
<td>I=0 TIE.C3I</td>
<td></td>
</tr>
<tr>
<td>$FEE6</td>
<td>12</td>
<td>Timer Channel 4, C4F</td>
<td>I=0 TIE.C4I</td>
<td></td>
</tr>
<tr>
<td>$FEE4</td>
<td>13</td>
<td>Timer Channel 5, C5F</td>
<td>I=0 TIE.C5I</td>
<td></td>
</tr>
<tr>
<td>$FEE2</td>
<td>14</td>
<td>Timer Channel 6, C6F</td>
<td>I=0 TIE.C6I</td>
<td></td>
</tr>
<tr>
<td>$FEE0</td>
<td>15</td>
<td>Timer Channel 7, C7F</td>
<td>I=0 TIE.C7I</td>
<td></td>
</tr>
<tr>
<td>$FDFE</td>
<td>16</td>
<td>Timer overflow, TOF</td>
<td>I=0 TIE.TOF</td>
<td></td>
</tr>
<tr>
<td>$FDFF</td>
<td>17</td>
<td>Pulse acc overflow, PAOVF</td>
<td>I=0 PACTL.PAOVF</td>
<td></td>
</tr>
<tr>
<td>$FDFC</td>
<td>18</td>
<td>Pulse acc input edge, PAIF</td>
<td>I=0 PACTL.PAIF</td>
<td></td>
</tr>
</tbody>
</table>

External Interrupt Design Approach

- First, identify status signal that indicates the busy-to-done state transition.
- Next, connect the I/O status signal to a microcomputer input that can generate interrupts.

Setting Interrupt Vectors in Assembly

```assembly
org $FFF0
fdb RTIHAN Ptr to real time interrupt handler
org $FFF2
fdb IRQHAN Ptr to external IRQ and STRA handler
org $FFF4
fdb XIRQHAN Ptr to external XIRQ handler
org $FFFE
fdb RESETHAN Ptr to reset handler
```

Interrupting Software

- **Ritual** - executed once, disable interrupts during, initialize globals, set port direction, set port interrupt control register, clear interrupt flag, arm device, and enable interrupts.
- **Main program** - initialize SP, execute ritual, interacts with ISRs via global data (ex. FIFO queue).
- **ISR(s)** - determine interrupt source, implement priority, acknowledge (clear the flag) or disarm, exchange info with main program via globals, execute rti to exit.
- **Interrupt vectors** - in general purpose processors vectors in RAM, in embedded systems usually in ROM.

Setting Interrupt Vectors in C

```c
unsigned short Time;
void RTI_Init(void){
    asm sei // Make atomic
    RTICTL = 0x73; // 30.517Hz
    CRGINT = 0x80; // Arm
    Time = 0; // Initialize
    asm cli
}

void interrupt 7 RTIHAn(void){
    CRGFLG = 0x80; // Acknowledge
    Time++;
}
```
Polled Versus Vectored Interrupts

- Vectored interrupts - each interrupt source has a unique interrupt vector address.
- Polled interrupts - multiple interrupt sources share the same interrupt vector address.
  - Minimal polling - check flag bit that caused interrupt.
  - Polling for 0s and 1s - verify entire status register.

Example of a Vectored Interrupt

```assembly
TimeHan movb #$80, TFLG2 ; clear TOF
; *Timer interrupt calculations*
rti
ExtHan movb #$80, PIFJ ; clear flag
; *External interrupt calculations*
rti
org $FFDE ; timer overflow
fdb TimeHan
org $FFCE ; Key wakeup J
fdb ExtHan
```

Example of a Polled Interrupt

```assembly
ExtHan brset PIFJ, $80, KJ7Han
brset PIFJ, $40, KJ6Han
swi ; error
KJ7Han movb #$80, PIFJ ; clear flag0
; *KJ7 interrupt calculations*
rti
KJ6Han movb #$40, PIFJ ; clear flag1
; *KJ6 interrupt calculations*
rti
org $FFCE ; Key wakeup J
fdb ExtHan
```

Keyboard Interface Using Interrupts

```assembly
// PT6-Pt0 inputs = keyboard DATA
// PJ7=STROBE interrupt on rise
void Key_Init(void){
  asm sei
  DDRT = 0x80; // PT6-0 DATA
  DDRJ &= ~0x80;
  PPSJ |= 0x80; // rise on PJ7
  PIEJ |= 0x80; // arm PJ7
  PIFJ = 0x80; // clear flag7
  Fifo_Init();
  asm cli
}
```

Interrupting Keyboard ISR

```assembly
void interrupt 24 ExtHan(void){
  if((PIFJ&0x80)==0){
    asm swi
  }
  PIFJ = 0x80; // clear flag
  Fifo_Put(PTT);
}
```
Printer Interface Using IRQ Interrupts

Printer Interface Helper Routines

Printer Interface Ritual

Printer Interface ISR

Power System Interface

Interrupt Polling Using Linked Lists

 ISR using polled interrupts must check status of all devices that may have caused the interrupt.  
 Must poll when two devices share the same interrupt vector (ex., SCI).  
 Sometimes poll anyway to verify status of the device to help detect software or hardware errors.  
 Polling using a linked list makes it easier to debug, change the polling order, add devices, or subtract devices.
Linked List Data Structure for Polling

```
const struct Node{
  unsigned char Mask; /* And Mask */
  void (*Handler)(void); /* Handler for this task */
  const struct Node *NextPt; /* Link to Next Node */
};
```

Interrupt Polling Using Linked Lists

```
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
NodeType sys[3] = {
  {0x04, PJ2Han, &sys[1]},
  {0x02, PJ1Han, &sys[2]},
  {0x01, PJ0Han, 0} };
void interrupt 23 KWJHan(void){
    NodePtr Pt;
    unsigned char Status;
    Pt=sys[0];
    while(Pt){ // executes each device handler
      if(KWIFJ&(Pt->Mask)){
        (*Pt->Handler)();} /* Execute handler */
      Pt=Pt->NextPt; } } // returns after all devices polled
```

Fixed Priority Using One Interrupt Line

```
void interrupt 23 KWJHan(void){
    NodePtr Pt;
    unsigned char Status;
    Pt=sys[0];
    while(Pt){ // executes each device handler
      if(KWIFJ&(Pt->Mask)){
        (*Pt->Handler)();} /* Execute handler */
      Pt=Pt->NextPt; } } // returns after all devices polled
```

Fixed Priority Implemented Using XIRQ

```
void interrupt 23 KWJHan(void){
    NodePtr Pt;
    unsigned char Status;
    Pt=sys[0];
    while(Pt){ // executes each device handler
      if(KWIFJ&(Pt->Mask)){
        (*Pt->Handler)();} /* Execute handler */
      Pt=Pt->NextPt; } } // returns after all devices polled
```

Round-Robin Polling

- Sometimes we want to have no priority.
- Gives service guarantee under heavy load to equally important devices.
- Round-robin polling rotates the polling order to allow all devices an equal chance of getting service.
- Does not apply to vectored interrupts.
- Example sequence of events:
  - Interrupt, poll A, B, C
  - Interrupt, poll B, C, A
  - Interrupt, poll C, A, B
  - Interrupt, poll A, B, C, etc.
Round-Robin Polling

NodeType sys[3]={
    {0x04, PJ2Han, &sys[1]},
    {0x02, PJ1Han, &sys[2]},
    {0x01, PJ0Han, &sys[0]} }; 

NodePtr Pt=&sys[0]; // points to one polled first last time
void interrupt 23 KWJHan(void){
    unsigned char Counter,Status;
    Counter=3; // quit after three devices checked
    Pt=Pt->NextPt; // rotates ABC BCA CAB polling orders
    while(Counter--){
        if(KWIFJ&(Pt->Mask)){
            (*Pt->Handler)();} /* Execute handler */
            Pt=Pt->NextPt; } } // returns after all devices polled

Real-Time Interrupts and Periodic Polling

- A real-time interrupt (RTI) is one that is requested on a fixed time basis.
- Required for data acquisition and control systems because servicing must be performed at accurate time intervals.
- RTIs also used for intermittent or periodic polling.
- In gadfly, I/O devices polled continuously.
- With periodic polling, I/O devices polled on regular basis.
- If no device needs service, interrupt simply returns.
- Use periodic polling if the following conditions apply:
  - The I/O hardware cannot generate interrupts directly.
  - We wish to perform I/O functions in the background.

Periodic Polling

![Periodic Polling Diagram]

Periodic Interrupt Using RTI

```c
unsigned short Time;
void RTI_Init(void) {
    asm sei // Make atomic
    RTICTL = 0x73; // 30.517Hz
    CRGINT = 0x80; // Arm
    Time = 0; // Initialize
    asm cli }

void interrupt 7 RTIHan(void) {
    CRGFLG = 0x80; // Acknowledge
    Time++;
}
```

Periodic Interrupt Using Timer Overflow

```c
#define PERIOD 1000
unsigned short Time;
void TOF_Init(void){
    asm sei // Make atomic
    TSCR1 = 0x80; // enable counter
    TSCR2 = 0x81; // Arm, 30.517Hz
    Time = 0; // Initialize
    asm cli // enable interrupts
}

void interrupt 16 TOFHan(void){
    TFLG2 = 0x80; // Acknowledge
    Time++;
}
```

Periodic Interrupt Using Output Compare

```c
#define PERIOD 1000
unsigned short Time;
void OC6_Init(void){
    asm sei // Make atomic
    TSCR1 = 0x80;
    TSCR2 = 0x02; // 1 MHz TCNT
    TIOS |= 0x40; // activate OC6
    TIE |= 0x40; // arm OC6
    TC6 = TCNT+50; // first in 50us
    Time = 0; // Initialize
    asm cli } // enable IRQ

void interrupt 14 OC6Handler(void) {
    TC6 = TC6+PERIOD; // next in 1 ms
    TFLG1 = 0x40; // acknowledge C6F
    Time++; }
```