Sequence of Events During Interrupt

- Hardware needing service makes a busy-to-done transition.
- 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).
- 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).
- Execution of the ISR.
- Return control back to the thread that was running.

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 a device means to enable the source of interrupts.
- To disarm a device means to shut off the source of interrupts.
- To enable means to allow interrupts at this time.
- To disable means to allow postpone interrupts at this time.

Stack Before and After an Interrupt

![Stack Diagram]
### 6812 Interrupts

- Each interrupt has a 16-bit vector stored in the 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 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>$FFFF$ 0</td>
<td>Reset</td>
<td>Always</td>
<td>Always highest</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 1</td>
<td>COP clk monitor fail</td>
<td>Always</td>
<td>COPCTL.CME</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 2</td>
<td>COP failure reset</td>
<td>Always</td>
<td>COP rate selected</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 3</td>
<td>Unimplemented instruction</td>
<td>Always</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 4</td>
<td>SWI</td>
<td>Always</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 5</td>
<td>XIRQ</td>
<td>X=0</td>
<td>External hardware</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 6</td>
<td>IRQ</td>
<td>I=0</td>
<td>INTCR.IRQEN</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 7</td>
<td>Real time interrupt, RTIF</td>
<td>I=0</td>
<td>CRGINT.RTIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 8</td>
<td>Timer Channel 0, C0F</td>
<td>I=0</td>
<td>TIE.C0I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 9</td>
<td>Timer Channel 1, C1F</td>
<td>I=0</td>
<td>TIE.C1I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 10</td>
<td>Timer Channel 2, C2F</td>
<td>I=0</td>
<td>TIE.C2I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 11</td>
<td>Timer Channel 3, C3F</td>
<td>I=0</td>
<td>TIE.C3I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 12</td>
<td>Timer Channel 4, C4F</td>
<td>I=0</td>
<td>TIE.C4I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 13</td>
<td>Timer Channel 5, C5F</td>
<td>I=0</td>
<td>TIE.C5I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 14</td>
<td>Timer Channel 6, C6F</td>
<td>I=0</td>
<td>TIE.C6I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 15</td>
<td>Timer Channel 7, C7F</td>
<td>I=0</td>
<td>TIE.C7I</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 16</td>
<td>Timer overflow, TOF</td>
<td>I=0</td>
<td>TIE.TOI</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 17</td>
<td>Pulse acc overflow, PAOVF</td>
<td>I=0</td>
<td>PACTL.PAOVI</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 18</td>
<td>Pulse acc input edge, PAIF</td>
<td>I=0</td>
<td>PACTL.PAI</td>
<td></td>
</tr>
</tbody>
</table>

### 6812 Interrupt Vectors and Priority (cont)

<table>
<thead>
<tr>
<th>Vector</th>
<th>CW#</th>
<th>Interrupt Source</th>
<th>Enable</th>
<th>Arm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFF$ 19</td>
<td>SPI complete, SPIF</td>
<td>I=0</td>
<td>SPICR1.SPIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 20</td>
<td>SPI transmit empty, SPTEF</td>
<td>I=0</td>
<td>SPICR1.SPTIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 21</td>
<td>SCI transmit buffer empty, TDRE</td>
<td>I=0</td>
<td>SCICR2.SIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 22</td>
<td>SCI transmit complete, TC</td>
<td>I=0</td>
<td>SCICR2.TCIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 23</td>
<td>SCI receiver buffer full, RDRF</td>
<td>I=0</td>
<td>SCICR2.RIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 24</td>
<td>SCI receiver idle, IDLE</td>
<td>I=0</td>
<td>SCICR2.ILIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 25</td>
<td>ATD sequence complete, ASCIF</td>
<td>I=0</td>
<td>ATDCTL2.ASCIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 26</td>
<td>Key wake up J, PIFJ.[7:6]</td>
<td>I=0</td>
<td>PIEJ.[7:6]</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 27</td>
<td>CAN wake up</td>
<td>I=0</td>
<td>CANRIER.WUPIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 28</td>
<td>CAN errors</td>
<td>I=0</td>
<td>CANRIER.CSIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 29</td>
<td>CAN receive</td>
<td>I=0</td>
<td>CANRIER.OVRIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 30</td>
<td>CAN transmit</td>
<td>I=0</td>
<td>CANRIER.RXFIE</td>
<td></td>
</tr>
<tr>
<td>$FFFF$ 31</td>
<td>Key wake up P, PIPI.[7:0]</td>
<td>I=0</td>
<td>PIEP.[7:0]</td>
<td></td>
</tr>
</tbody>
</table>
**External Interrupt Design Approach**

- Identify status signal indicating the busy-to-done state transition.
- Connect the I/O status signal to a microcontroller input that can generate interrupts.

![Diagram of MC9S12C32 and I/O Device Status]

**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

```
MC9S12C32
PJ7
I/O Device Status

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

```
MC9S12C32
PJ7
PJ6
I/O Devices

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

```
MC9S12C32
PJ7
PJ6
Keyboard
STROBE
STROBE DATA available

Until the next key

Input character
acknowledge
Put character
into data structure

IRQ

Yes

Poll

No

STROBE causes
interrupt

Interrupt from
unknown source

RTI
```

Interrupting Keyboard Ritual

```
// 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

```c
void interrupt 24 ExtHan(void){
    if((PIFJ&0x80)==0){
        asm swi
    }
    PIFJ = 0x80;  // clear flag
    Fifo_Put(PTT);
}
```

Printer Interface Helper Routines

```c
// PT6-PT0 outputs = printer DATA
// PJ7=READY interrupt on rise, PJ6=START pulse out
unsigned char OK;  // 0=busy, 1=done
unsigned char Line[20];  //ASCII data
unsigned char *Pt;  // pointer to line
void Fill(unsigned char *p){
    Pt=&Line[0];
    while(*Pt++=*p++);  // copy
    Pt=&Line[0];  // initialize pointer
    OK=0;
}
unsigned char Get(void){
    return(*Pt++);
}
void Out(unsigned char data){
    PTJ |= 0x40;  // START=0
    PTT=data;  // write DATA
    PTJ |= 0x40;  // START=1
}
```

Printer Interface Using IRQ Interrupts

```c
void Print_Init(unsigned char *thePt){
    asm sei  // make atomic
    Fill(thePt);  // copy data into global
    DDRT = 0xFF;  // PT6-0 output DATA
    DDRJ = 0x40;  // PJ7=START output
    PPSJ |= 0x80;  // rise on PJ7
    PPEJ |= 0x80;  // arm PJ7
    PIFJ = 0x80;  // clear flag7
    Out(Get());  // start first
    asm cli
}
```
Printer Interface ISR

```c
void interrupt 24 ExtHan(void){
    if((PIFJ&0x80)==0) asm("swi");
    PIFJ = 0x80; // clear flag
    if(data=Get())
        Out(data); // start next
    else{
        PIEJ &= ~0x80; // disarm
        OK=1; // line complete
    }
}
```

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 (for example: 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.

Power System Interface

```c
/* XIRQ requested on a rise of TooLow
PB0, negative logic pulse, will acknowledge XIRQ
PB1=1 will activate backup power */
void Ritual(void){
    DDRB=0xFF;
    PORTB=0; PORTB=1; // Make XIRQ=1
    asm("ldaa #0x10\n     tap");
    void interrupt 5 PowerLow(void){
        PORTB=2;
        PORTB=3; } /* Ack, turn on backup power */
```

Linked List Data Structure for Polling
Interrupt Polling Using Linked Lists

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

void PJ2Han(void){
    KWIFJ=0x04;
    Counter2++;
}
void PJ1Han(void){
    KWIFJ=0x02;
    Counter1++;
}
void PJ0Han(void){
    KWIFJ=0x01;
    Counter0++;
}
```

Fixed Priority Using One Interrupt Line

```c
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 Implemented Using XIRQ

```c
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
NodeType sys[3]=
    {0x04, PJ2Han, &sys[1]},
    {0x02, PJ1Han, &sys[2]},
    {0x01, PJ0Han, 0};
```
Round-Robin Polling

- Sometimes we want to have no priority which gives a 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.

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.

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

Periodic Polling
Peri odi c Inter rupt Using RTI

unsigned short Time;
void RTI_Init(void){
    asmsei     // Make atomic
    RTICTL = 0x73; // 30.517Hz
    CRGINT = 0x80; // Arm
    Time = 0;     // Initialize
    asmcli
}

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

Peri odi c Inter rupt Using Timer Ov erflo w

unsigned short Time;
void TOF_Init(void){
    asmsei     // Make atomic
    TSCR1 = 0x80; // enable counter
    TSCR2 = 0x81; // Arm, 30.517Hz
    Time = 0;     // Initialize
    asmcli     // enable interrupts
}

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

Peri odi c Inter rupt Using Output Compare

#define PERIOD 1000
unsigned short Time;
void OC6_Init(void){
    asmsei     // 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
    asmcli  // enable IRQ
void interrupt 14 OC6handler(void){
    TC6 = TC6+PERIOD; // next in 1 ms
    TFLG1 = 0x40; // acknowledge C6F
    Time++;  }