Lecture 10: Interrupts in the 6812
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.
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 (e.g., TOF=1).
Before fetching an instruction the HCS12 (except rev, revw, and wav):

3.1 Sees that the flag is set
3.2 Checks that the device is armed (e.g., TOI=1).
3.3 Checks that interrupts are enabled (I=0).

4. Jump to ISR:

4.1 Push all registers onto the stack.
4.2 Obtain vector address and load into PC.
4.3 Disable interrupts (I=1).

5. Execution of the ISR.

6. Pop the PC and registers and return to previous context.
Stack Before and After an Interrupt

Before the interrupt:
- PC
- Main
- Handler
- SP
- Top

After the interrupt:
- PC
- Main
- Handler
- Old CC
- Old B
- Old A
- Old X
- Old Y
- Old PC
- Top
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 has two external requests \( \overline{IRQ} \) and \( \overline{XIRQ} \).

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 the highest priority interrupt 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 \texttt{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>$FFFFE</td>
<td>0</td>
<td>Reset</td>
<td>Always</td>
<td>Always highest</td>
</tr>
<tr>
<td>$FFFC</td>
<td>1</td>
<td>COP clk monitor fail</td>
<td>Always</td>
<td>COPCTL.CME</td>
</tr>
<tr>
<td>$FFFA</td>
<td>2</td>
<td>COP failure reset</td>
<td>Always</td>
<td>COP rate selected</td>
</tr>
<tr>
<td>$FFF8</td>
<td>3</td>
<td>Unimplemented instruction</td>
<td>Always</td>
<td>Always</td>
</tr>
<tr>
<td>$FFF6</td>
<td>4</td>
<td>SWI</td>
<td>Always</td>
<td>Always</td>
</tr>
<tr>
<td>$FFF4</td>
<td>5</td>
<td>XIRQ</td>
<td>X=0</td>
<td>External hardware</td>
</tr>
<tr>
<td>$FFF2</td>
<td>6</td>
<td>IRQ</td>
<td>I=0</td>
<td>INTCR.IRQEN</td>
</tr>
<tr>
<td>$FFF0</td>
<td>7</td>
<td>Real time interrupt, RTIF</td>
<td>I=0</td>
<td>CRGINT.RTIE</td>
</tr>
<tr>
<td>$FFEE</td>
<td>8</td>
<td>Timer Channel 0, C0F</td>
<td>I=0</td>
<td>TIE.C0I</td>
</tr>
<tr>
<td>$FFEC</td>
<td>9</td>
<td>Timer Channel 1, C1F</td>
<td>I=0</td>
<td>TIE.C1I</td>
</tr>
<tr>
<td>$FFEA</td>
<td>10</td>
<td>Timer Channel 2, C2F</td>
<td>I=0</td>
<td>TIE.C2I</td>
</tr>
<tr>
<td>$FFE8</td>
<td>11</td>
<td>Timer Channel 3, C3F</td>
<td>I=0</td>
<td>TIE.C3I</td>
</tr>
<tr>
<td>$FFE6</td>
<td>12</td>
<td>Timer Channel 4, C4F</td>
<td>I=0</td>
<td>TIE.C4I</td>
</tr>
<tr>
<td>$FFE4</td>
<td>13</td>
<td>Timer Channel 5, C5F</td>
<td>I=0</td>
<td>TIE.C5I</td>
</tr>
<tr>
<td>$FFE2</td>
<td>14</td>
<td>Timer Channel 6, C6F</td>
<td>I=0</td>
<td>TIE.C6I</td>
</tr>
<tr>
<td>$FFE0</td>
<td>15</td>
<td>Timer Channel 7, C7F</td>
<td>I=0</td>
<td>TIE.C7I</td>
</tr>
<tr>
<td>$FFDE</td>
<td>16</td>
<td>Timer overflow, TOF</td>
<td>I=0</td>
<td>TIE.TOI</td>
</tr>
<tr>
<td>$FFDC</td>
<td>17</td>
<td>Pulse acc overflow, PAOVF</td>
<td>I=0</td>
<td>PACTL.PAOVI</td>
</tr>
<tr>
<td>$FFDA</td>
<td>18</td>
<td>Pulse acc input edge, PAIF</td>
<td>I=0</td>
<td>PACTL.PAI</td>
</tr>
<tr>
<td>Vector</td>
<td>CW#</td>
<td>Interrupt Source</td>
<td>Enable</td>
<td>Arm</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-----------------------------------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>$FFD8</td>
<td>19</td>
<td>SPI complete, SPIF</td>
<td>I=0</td>
<td>SPICR1.SPIE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SPI transmit empty, SPTEF</td>
<td>I=0</td>
<td>SPICR1.SPTIE</td>
</tr>
<tr>
<td>$FFD6</td>
<td>20</td>
<td>SCI transmit buffer empty, TDRE</td>
<td>I=0</td>
<td>SCICR2.TIE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCI transmit complete, TC</td>
<td>I=0</td>
<td>SCICR2.TCIE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCI receiver buffer full, RDRF</td>
<td>I=0</td>
<td>SCICR2.RIE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCI receiver idle, IDLE</td>
<td>I=0</td>
<td>SCICR2.ILIE</td>
</tr>
<tr>
<td>$FFD2</td>
<td>22</td>
<td>ATD sequence complete, ASCIF</td>
<td>I=0</td>
<td>ATDCTL2.ASCIE</td>
</tr>
<tr>
<td>$FFCE</td>
<td>24</td>
<td>Key wakeup J, PIFJ.[7:6]</td>
<td>I=0</td>
<td>PIEJ.[7:6]</td>
</tr>
<tr>
<td>$FFB6</td>
<td>36</td>
<td>CAN wakeup</td>
<td>I=0</td>
<td>CANRIER.WUPIE</td>
</tr>
<tr>
<td>$FFB4</td>
<td>37</td>
<td>CAN errors</td>
<td>I=0</td>
<td>CANRIER.CSCIE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>I=0</td>
<td>CANRIER.OVRIE</td>
</tr>
<tr>
<td>$FFB2</td>
<td>38</td>
<td>CAN receive</td>
<td>I=0</td>
<td>CANRIER.RXFIE</td>
</tr>
<tr>
<td>$FFB0</td>
<td>39</td>
<td>CAN transmit</td>
<td>I=0</td>
<td>CANRIER.TXEIE[2:0]</td>
</tr>
<tr>
<td>$FF8E</td>
<td>56</td>
<td>Key wakeup P, PIFP.[7:0]</td>
<td>I=0</td>
<td>PIEP.[7:0]</td>
</tr>
</tbody>
</table>
Identify status signal indicating the busy-to-done state transition.

Connect the I/O status signal to a microcontroller input that can generate interrupts.
Writing an Interruptable Program

“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, interact 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.
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

```
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

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
PT6-0

Keyboard
STROBE
DATA

STROBE
DATA available

Until the next key

IRQ
STROBE causes interrupt

Poll

Yes
Input character acknowledge

No
Interrupt from unknown source

Put character into data structure

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

[Diagram of printer interface using IRQ interrupts]

- MC9S12C32, PJ7, PJ6, PT6-0
- READY Printer
  - START
  - DATA
- START
- DATA required: Set up = 100 ns, Hold = 20 ns
- READY
- IRQ interrupt
- Ready requests interrupt

[Flowchart of IRQ handling]

- IRQ
- Poll
- Get next character from data structure
- Other
- Done
- Null
- Output character acknowledge
- Disarm OK=1
- RTI
Printer Interface Helper Routines

// 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
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
}

void interrupt 24 ExtHan(void) {
    if((PIFJ&0x80)==0) asm(" swi");
    PIFJ = 0x80; // clear flag7
    if(data=Get())
        Out(data); // start next
    else{
        PIEJ &= ~0x80; // disarm
        OK=1; // line complete
    }
}
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.
const struct Node{
    unsigned char Mask;    /* And Mask */
    void (*Handler)(void); /* Handler for this task */
    const struct Node *NextPt; /* Link to Next Node */
};
unsigned char Counter2, Counter1, Counter0;
void PJ2Han(void) {
    KWIFJ=0x04;
    Counter2++;
}
void PJ1Han(void) {
    KWIFJ=0x02;
    Counter1++;
}
void PJ0Han(void) {
    KWIFJ=0x01;
    Counter0++;
}
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

Diagram:
- IRQ wire connected to +5
- 6811/6812
- IRQA
- CA1
- Device 1
- Device 2

Flowchart:
- IRQ
  - No, Device 1 is not ready
    - Poll 1
      - Yes, Device 1 wants service
        - Read DATA 1
          - Acknowledge 1
          - Perform service for Device 1
        - Reenable IRQ
          - cli
          - Perform service for Device 2
            - Device 1 can interrupt Device 2
            - Device 2 cannot interrupt Device 1
  - Yes, Device 2 wants service
    - Poll 2
      - No, Device 2 is not ready
        - Poll 1
          - Yes, Device 1 wants service
            - Read DATA 1
              - Acknowledge 1
              - Perform service for Device 1
        - Reenable IRQ
          - cli
          - Perform service for Device 2
            - Device 1 can interrupt Device 2
            - Device 2 cannot interrupt Device 1
  - swi

Fixed Priority Implemented Using XIRQ
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.
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
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

1. Real time interrupt
2. Device 1
   - Ready
   - Busy
   - Input/output data 1
3. Device 2
   - Ready
   - Busy
   - Input/output data 2
4. Device n
   - Ready
   - Busy
   - Input/output data n
5. Acknowledge interrupt
6. RTI
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++;
}
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++;
}
#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++; }
Today we looked at interfacing using interrupts
This is a big topic, we just scratched the surface (quite a bit amore on this in 5785)

To write correct interrupt code:

Follow the basic patterns given in this lecture
Make all data shared between interrupts and main volatile
In main, only touch this data from inside a critical section
Match begins and ends of critical sections
Data shared with an interrupt must be ready for that interrupt to fire any time interrupts are enabled
Never enable interrupts from inside an interrupt handler

All of these rules can be broken by advanced programmers