ECE/CS 5780/6780: Embedded System Design

Chris J. Myers

Lecture 9: Interrupts in the 6812

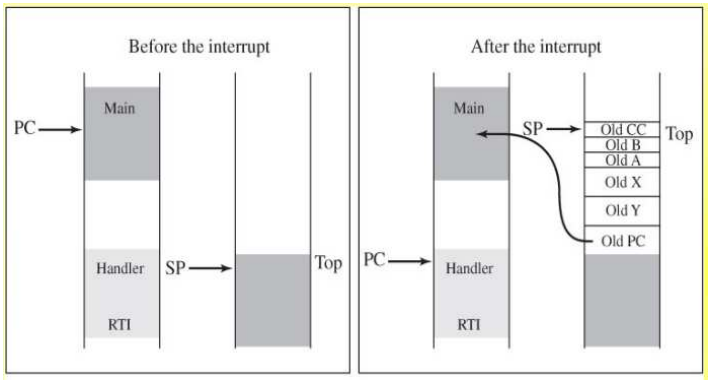
General Features of Interrupts

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



6812 Interrupts

- Each interrupt has 16-bit vector stored in upper 128 bytes of memory.
- There are six interrupt sources that are not maskable.
  1. Power-on-reset (POR) or regular hardware RESET pin
  2. Clock monitor reset
  3. COP watchdog reset
  4. Unimplemented instruction trap
  5. Software interrupt instruction (`swi`)
  6. XIRQ signal (if X bit in CCR = 0)

6812 Interrupts (cont)

- 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 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

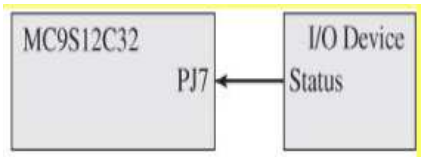
Vector	CW#	Interrupt Source	Enable	Arm
$FFFE	0	Reset	Always	Always highest
$FFFC	1	COP clk monitor fail	Always	COPCTL.CME
$FFFA	2	COP failure reset	Always	COP rate selected
$FFF8	3	Unimplemented instruction	Always	Always
$FFF6	4	SWI	Always	Always
$FFF4	5	XIRQ	X=0	External hardware
$FFF2	6	IRQ	I=0	INTCR.IRQEN
$FFF0	7	Real time interrupt, RTIF	I=0	CRGINT.RTIE
$FFEE	8	Timer Channel 0, C0F	I=0	TIE.C0I
$FFEC	9	Timer Channel 1, C1F	I=0	TIE.C1I
$FFEA	10	Timer Channel 2, C2F	I=0	TIE.C2I
$FFE8	11	Timer Channel 3, C3F	I=0	TIE.C3I
$FFE6	12	Timer Channel 4, C4F	I=0	TIE.C4I
$FFE4	13	Timer Channel 5, C5F	I=0	TIE.C5I
$FFE2	14	Timer Channel 6, C6F	I=0	TIE.C6I
$FFE0	15	Timer Channel 7, C7F	I=0	TIE.C7I
$FFDE	16	Timer overflow, TOF	I=0	TIE.TOI
$FFDC	17	Pulse acc overflow, PAOVF	I=0	PACTL.PAOVI
$FFDA	18	Pulse acc input edge, PAIF	I=0	PACTL.PAI

6812 Interrupt Vectors and Priority (cont)

Vector	CW#	Interrupt Source	Enable	Arm
$FFD8	19	SPI complete,SPIF	I=0	SPICR1.SPIE
		SPI transmit empty, SPTEF	I=0	SPICR1.SPTIE
$FFD6	20	SCI transmit buffer empty, TDRE	I=0	SCICR2.TIE
		SCI transmit complete, TC	I=0	SCICR2.TCIE
		SCI receiver buffer full, RDRF	I=0	SCICR2.RIE
		SCI receiver idle, IDLE	I=0	SCICR2.ILIE
$FFD2	22	ATD sequence complete, ASCIF	I=0	ATDCTL2.ASCIE
$FFCE	24	Key wakeup J, PIFJ.[7:6]	I=0	PIEJ.[7:6]
$FFB6	36	CAN wakeup	I=0	CANRIER.WUPIE
$FFB4	37	CAN errors	I=0	CANRIER.CSCIE
			I=0	CANRIER.OVRIE
$FFB2	38	CAN receive	I=0	CANRIER.RXFIE
$FFB0	39	CAN transmit	I=0	CANRIER.TXEIE[2:0]
$FF8E	56	Key wakeup P, PIFP.[7:0]	I=0	PIEP.[7:0]

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.



Interrupting Software

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

Setting Interrupt Vectors in 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
```

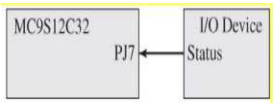

Setting Interrupt Vectors in C

```
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

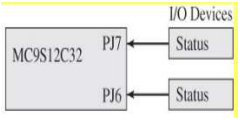
- Vectored interrupts - each interrupt source has a unique interrupt vector address.
- Polled interrupts - multiple interrupt sources share the same interrupt vector address.
  - Minimal polling - check flag bit that caused interrupt.
  - Polling for 0s and 1s - verify entire status register.

Example of a Vectored Interrupt



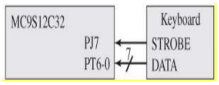

```
TimeHan movb #$80,TFLG2 ;clear TOF
;*Timer interrupt calculations*
        rti

ExtHan  movb #$80,PIFJ ;clear flag
;*External interrupt calculations*
        rti

        org  $FFDE  ;timer overflow
        fdb  TimeHan
        org  $FFCE  ;Key wakeup J
        fdb  ExtHan
```

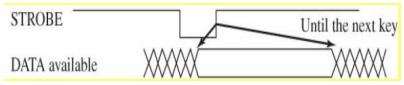

Example of a Polled Interrupt

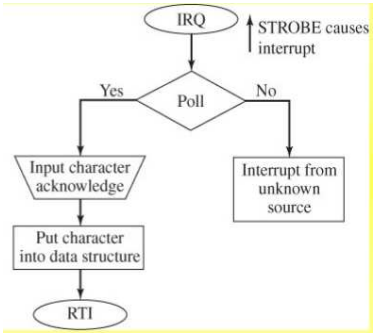



```
ExtHan brset PIFJ,$80,KJ7Han
       brset PIFJ,$40,KJ6Han
       swi         ;error
KJ7Han movb #$80,PIFJ ;clear flag0
;*KJ7 interrupt calculations*
       rti
KJ6Han movb #$40,PIFJ ;clear flag1
;*KJ6 interrupt calculations*
       rti
       org  $FFCE  ;Key wakeup J
       fdb  ExtHan
```


Keyboard Interface Using Interrupts







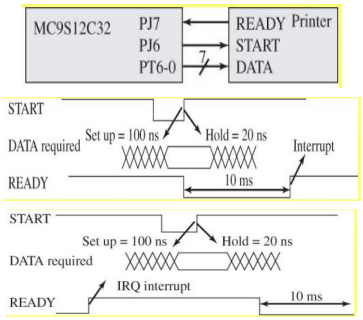
Interrupting Keyboard Ritual

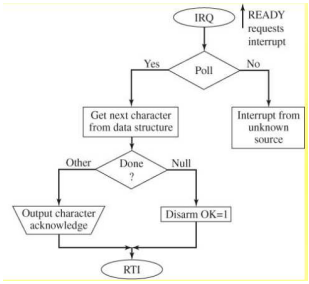
```
// PT6-Pt0 inputs = keyboard DATA
// PJ7=STROBE interrupt on rise
void Key_Init(void){
asm sei
  DDRT = 0x80;   // PT6-0 DATA
  DDRJ &= ~0x80;
  PPSJ |= 0x80;  // rise on PJ7
  PIEJ |= 0x80;  // arm PJ7
  PIFJ = 0x80;   // clear flag7
  Fifo_Init();
asm cli
}
```


Interrupting Keyboard ISR

```
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

```
// 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
```

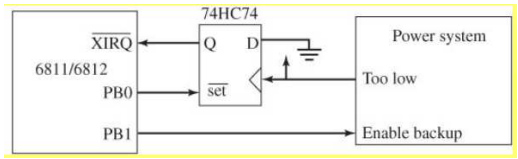

Printer Interface Ritual

```
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

```
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
  }
}
```


Power System Interface

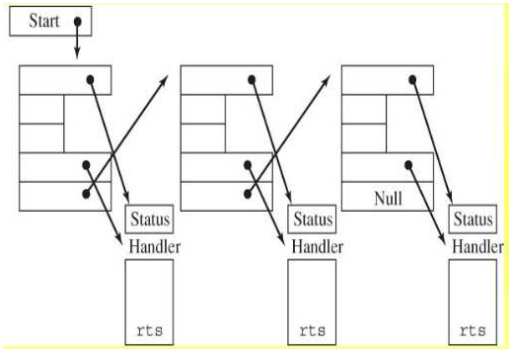



```
/* XIRQ requested on a rise of TooLow
   PB0, negative logic pulse, will acknowledge XIRQ
   PB1=1 will activate backup power */
void Ritual(void){
  DDRB=0xFF;        // Port B outputs
  PORTB=0; PORTB=1; // Make XIRQ=1
asm(" ldaa #0x10\n tap"); }
void interrupt 5 PowerLow(void){
  PORTB=2;
  PORTB=3;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


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



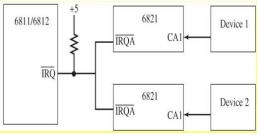
Interrupt Polling Using Linked Lists

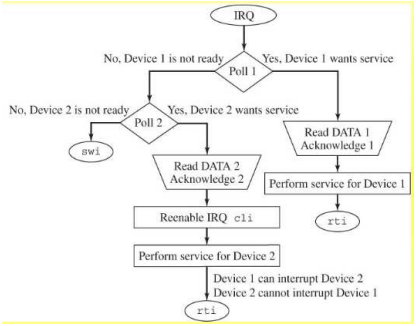
```
const struct Node{
  unsigned char Mask;        /* And Mask */
  void (*Handler)(void);     /* Handler for this task */
  const struct Node *NextPt; /* Link to Next Node */
};
unsigned char Counter2,Counter1,Counter0;
void PJ2Han(void){
    KWIFJ=0x04;
    Counter2++;}
void PJ1Han(void){
    KWIFJ=0x02;
    Counter1++;}
void PJ0Han(void){
    KWIFJ=0x01;
    Counter0++;}
```


Interrupt Polling Using Linked Lists

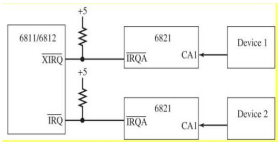
```
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
NodeType sys[3]={
    {0x04, PJ2Han, &sys[1]},
    {0x02, PJ1Han, &sys[2]},
    {0x01, PJ0Han,   0    } };
void interrupt 23 KWJHan(void){
  NodePtr Pt;
  unsigned char Status;
  Pt=&sys[0];
  while(Pt){            // executes each device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

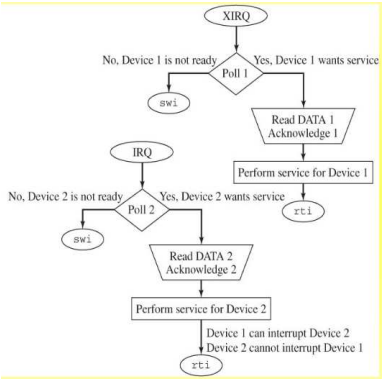

Fixed Priority Using One Interrupt Line





Fixed Priority Implemented Using XIRQ





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.

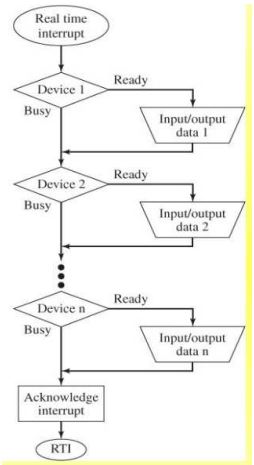
Round-Robin Polling

```
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 Interrupt Using 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++;
}
```


Periodic Interrupt Using Timer Overflow

```
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

```
#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++; }
```