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., IRQ and STAF, 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 in assembly

```assembly
start fdb 11STAF place to start polling
Sreg equ 0 Index to Status Register
Amask equ 2 and mask
Cmask equ 3 compare mask
DevHan equ 4 device handler
NextPt equ 6 next pointer
num fcb 3 number of devices
11STAF fdb $1002 address of PI0C
fcb $ff look at all the bits in PI0C
fcb $C0 expect exact match with $C0
fdb STAFhan device handler
fdb 11CA1 ptr to next device to poll
```
Interrupt Polling Using Linked Lists in Assembly

Slide 5

```
11CA1 fdb $2011 addr 6821 Port A Ctrl/Stat
fcb $87 look at bits 7,2,1,0
fcb $85 expect to be 1,1,0,1
fdb CA1han device handler for CA1
fdb 11CB2 ptr to next device to poll
11CB2 fdb $2013 addr of 6821 Port B Ctrl/Stat
fcb $7C look at bits 6,5,4,3,2
fcb $4C expect to be 1,0,0,1,1
fdb CB2han device handler for CB2
fdb 0 no more
```

Interrupt Polling Using Linked Lists in C

Slide 7

```
const struct Node{
    unsigned char *StatusPt; /* Ptr to status reg */
    unsigned char Amask; /* And Mask */
    unsigned char Cmask; /* Compare Mask */
    void (*Handler)(void); /* Handler */
} const struct Node *sys[3] = {
    {&PIOC, 0xFF, 0xC0, STRAHan, &sys[1]},
    {&ACNT, 0x87, 0x85, PIAHanA, &sys[2]},
    {&BCNT, 0x7C, 0x4C, PIAHanB, 0} 
};
```

Interrupt Polling Using Linked Lists in Assembly

Slide 6

```
ldx start Reg X points to linked list
ldab num number of possible devices
next ldy Sreg,x Reg Y points to status reg
ldaa,y read status
anda Amask,x clear indeterminate bits
cmpa Cmask,x expected value if active
beq Notyet skip if not requesting
ldy DevHan,x Reg Y points to handler
jsr,y call device handler
Notyet ldx NextPt,x Reg X points to next entry
decb device counter
bne next check next device
rti
```

Interrupt Polling Using Linked Lists in C

Slide 8

```
typedef const struct Node NodeTypp;
typedef NodeTypp *NodePtr;
NodeTypp sys[3] = {
    {&PIOC, 0xFF, 0xC0, STRAHan, &sys[1]},
    {&ACNT, 0x87, 0x85, PIAHanA, &sys[2]},
    {&BCNT, 0x7C, 0x4C, PIAHanB, 0} 
};
#define interrupt_handler IRQHan()
void IRQHan(void){ // regular functions that
    CLdata=PORTCL; /* return (rts) when done
    void PIAHanA(void){
        PIAdata=ADATA;
    }
    void PIAHanB(void){
        PIABdata=BDATA;
    }
    while(1);}
```
Fixed Priority Using One Interrupt Line

Fixed Priority Implemented Using XIRQ

Round-Robin Polling

- Sometimes we want to have no priority.
- This gives a guarantee of service under a 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 in C

```c
NodeType sys[3]=
    {&PIOC, 0xFF, 0xC0, STRAHan, &sys[1]},
    {&ACNT, 0x87, 0x85, PIAHanA, &sys[2]},
    {&BCNT, 0x7C, 0x4C, PIAHanB, &sys[0]};
NodePtr Pt=&sys[0];//polled first last interrupt
#pragma interrupt_handler IRQHan()
void IRQHan(void){
    unsigned char Counter, Status;
    Counter=3; // quit after 3 devices checked
    Pt=Pt->NextPt; // rotates polling orders
    while(Counter--){
        Status=*Pt->StatusPt;
        if(((Status&Pt->Amask))==(Pt->Cmask)){
            (*Pt->Handler)();} /* Execute handler */
        Pt=Pt->NextPt; }
    }`
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:
  1. The I/O hardware cannot generate interrupts directly.
  2. We wish to perform I/O functions in the background.

Real Time Interrupt Using a 6811 STRA

Periodic Polling

Slide 13
### Periodic Interrupt Using RTI in Assembly

- **RITUAL**
  - sei: disable interrupts during RITUAL
- ldaa #3: Set RTR1, RTR0 = 11
- staa PACTL: Interrupt period = 32.768ms
- ldaa #$40: Set RTII = 1
- staa TMSK2: Enable IRQ interrupts
- cli:还不是
- rts:
- **RTIHAN**
  - ldaa TFLG2: expect=X1XX0000
  - anda #$4F: ignore TOF, PADVF, and PAIF
  - cmpa #$40: RTIF should equal 1
  - beq OK: Error
  - svi: OK
- OK
  - ldaa #$40: RTIF is cleared by writing to TFLG2
  - staa TFLG2: with bit 6 set
- * service occurs every 32.768ms or about 30.517Hz

### Periodic Interrupt Using RTI in C

```c
unsigned int Time;
#pragma interrupt_handler RTIHAn()
void RTIHAn(void){
   if((TFLG2&0x4F)!=0x40) asm("swi"); /*Illegal*/
   TFLG2=RTIF; /*Acknowledge by clearing RTIF */
   Time++;
}
void Ritual(void){
   asm("sei"); /*Make ritual atomic */
   PACTL=(0xFC&TMSK2)|2; /*Set RTR to 2, 61.035Hz*/
   TMSK2|=RTII; /*Arm RTI */
   Time=0; /*Initialize global data */
   asm("cli");
}
```

### 6811 RTI

<table>
<thead>
<tr>
<th>RTR1</th>
<th>RTR0</th>
<th>Divide E by</th>
<th>Period (us)</th>
<th>Frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$2^{25}$</td>
<td>4,096</td>
<td>244.14</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$2^{24}$</td>
<td>8,192</td>
<td>122.07</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$2^{35}$</td>
<td>16,384</td>
<td>61.035</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$2^{16}$</td>
<td>32,768</td>
<td>30.517</td>
</tr>
</tbody>
</table>

- **Periodic Interrupt Using RTI**
  - disable interrupts during RITUAL
  - Set RTR1, RTR0 = 11
  - Interrupt period = 32.768ms
  - Set RTII = 1
  - Enable IRQ interrupts
  - Make ritual atomic
  - Set RTR to 2, 61.035Hz
  - Arm RTI
  - Initialize global data

- **RTIHAn**
  - RTIF is cleared by writing to TFLG2
  - with bit 6 set
  - service occurs every 32.768ms or about 30.517Hz