Introduction
- Interrupts provide guarantee on response time.
- Interrupts allow response to rare but important events.
- Periodic interrupts used for data acquisition and control.
- Interrupts can provide a way to buffer I/O data.

What are Interrupts?
- An automatic transfer of software execution in response to hardware that is asynchronous with current software.
- Hardware can be external I/O device or internal event.
- When hardware needs service, it requests an interrupt.
- Calls interrupt service routine as a background thread.
- Thread is terminated with rti instruction.
- Threads may communicate using FIFO queues and synchronize using semaphores.
- Threads share global variables while processes do not.
- Each potential interrupt has separate arm bit.
- Interrupt enable bit, I, found in condition code.
Slide 5

Shared versus Dedicated

- Wire-or negative-logic interrupt requests:
  1. Can add additional I/O devices w/o redesigning H/W.
  2. No limit to number of interrupting I/O devices.
  3. Microcomputer hardware is simple.
- Dedicated edge-triggered interrupt requests:
  1. Software is simpler, easier to debug, and faster.
  2. Less coupling between software modules.
  3. Easier to implement priority.

Slide 6

Interrupt Service Routines (ISR)

- Software executed when hardware requests an interrupt.
- Pollled interrupts - one large ISR handles all requests.
- Vectored interrupts - many small, specific ISRs.
- When the device is armed, the I bit is zero, and an interrupt is requested, it is serviced as follows:
  1. Execution of main program is suspended.
  2. All registers are pushed onto the stack.
  3. The ISR, or background thread, is executed.
  4. The ISR executes rti instruction.
  5. All registers are restored from the stack.
  6. The main program is resumed.

Slide 7

Interrupt Execution

Slide 8

When to Use Interrupts

<table>
<thead>
<tr>
<th></th>
<th>Pollled interrupts</th>
<th>Vectored interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadfly</td>
<td>Predicable</td>
<td></td>
</tr>
<tr>
<td>Simple I/O</td>
<td>Complex I/O</td>
<td>High bandwidth</td>
</tr>
<tr>
<td>Single thread</td>
<td>Multithread</td>
<td></td>
</tr>
<tr>
<td>Nothing else to do</td>
<td>Infrequent alarms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program errors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Overflow, illegal op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Illegal memory access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Machine/memory errors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Power failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Real-time clocks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data acquisition/control</td>
<td></td>
</tr>
</tbody>
</table>
Interthread Communication

- Interrupt threads have logically separate registers/stack, so communication must occur through global memory.

Input Device Interrupts

Output Device Interrupts

Other Interrupt Issues

1. Periodic interrupts are neither input or output.
2. Essential for data acquisition and control systems.
3. ISR should only occur when needed, come in clean, perform function, and return right away.
4. Gadfly loops and iterations should be avoided in ISRs.
5. Percent of time in ISRs should be minimized.
6. Interface latency is time between new input available and when software reads the input data.
7. Device latency is response time of external I/O device.
8. A real-time system guarantees bound on interface latency.
Reentrant Programming

- A program segment is reentrant if it can be concurrently executed by two (or more) threads.
- Reentrant software must place local variables on stack.
- A nonreentrant subroutine has a section of code called a vulnerable window or critical section, and error occurs if:
  1. One thread calls the nonreentrant subroutine,
  2. That thread is executing in the “vulnerable” window when interrupted by a second thread, and
  3. Second thread calls same sub or shared variable.
- Need to implement mutual exclusion which is typically done by disabling interrupts.

Nonreentrant Subroutine in Assembly

```
Second rmb 2 Temporary global variable
* Input parameters: Reg X,Y contain 2 16 bit numbers
* Output parameter: Reg X is returned with the average
AVE   sty Second Save the second number in memory
      xgd x Reg D contains first number
add Second Reg D=First+Second
lsrd (First+Second)/2
adcb #0 round up?
      xgd x
rts
```

Nonreentrant Subroutine in C

```c
int Result; /* Temporary global variable */
int AVE(int x,y){
    Result = y; /* Save second number */
    Result = (Result + x) >> 1; /* (1st+2nd)/2 */
    return(Result);
}
```
Atomic Operations

- Atomic operation is one that once started is guaranteed to finish.
- In most computers, machine instructions are atomic.
- The following is atomic:
  \begin{verbatim}
  inc counter  where counter is global variable
  \end{verbatim}
- The following is nonatomic:
  \begin{verbatim}
  ldd counter  where counter is global variable
  inc
  staa counter
  \end{verbatim}

Read-Modify-Write Example

1. Software reads global variable, producing a copy of the data.
2. Software modifies the copy.
3. Software writes modification back into global variable.

\begin{verbatim}
unsigned int Money; /* bank balance (global) */
/* add 100 dollars */
void more(void){
  Money += 100;}
\end{verbatim}

Money rmb 2  bank balance implemented as a global
* add 100 dollars to the account
more ldd Money  where Money is a global variable
add #100
std Money Money=Money+100
rts

Write Followed by Read Example

1. Software writes to a global variable.
2. Software reads from global variable expecting original data.

\begin{verbatim}
int temp;  /* global temporary */
/* calculate x+2*d */
int mac(int x, int d){
temp = x+2*d;  /* write to a global variable */
  return (temp);  /* read from global */
}
\end{verbatim}

temp rmb 2  global temporary result
* calculate RegX=RegX+2*RegD
mac stx temp  Save X so that it can be added
lsld  RegD=2*RegD
add temp  RegD=RegX+2*RegD
xgdx  RegX=RegX+2*RegD
rts

Nonatomic Multistep Write

1. Software write part of new value to a global variable.
2. Software write rest of new value to a global variable.

\begin{verbatim}
int info[2];  /* 32-bit global */
void set(int x, int y){
  info[0]=x;
  info[1]=y;)
\end{verbatim}

Info rmb 4  32-bit data implemented as a global
* set the variable using RegX and RegY
set stx Info  Info is a 32 bit global variable
sty Info+2
rts
**Make a Subroutine Reentrant Using a Stack Variable**

* Input parameters: Reg X, Y contain 2 16 bit numbers
* Output parameter: Reg X is returned with the average

```
AVE    pshy    Save the second number on the stack
tsy    Reg Y points the Second number
xgdx   Reg D contains first number
add 0,Y Reg D=First+Second
lsrd   (First+Second)/2
adcb #0 round up?
adca #0
xgdx
puly
rts
```

**A Nonreentrant Subroutine**

Status  rmb 1  0 means empty, -1 otherwise
Message  rmb 1  data to be communicated

* Input param: Reg B contains an 8 bit message
* Output param: Reg CC (C bit) is 1 for OK, 0 for busy error

```
SEND tst Status check if mailbox is empty
bmi Busy full, can't store, so return C=0
staa Message store
sec Status signify now contains a message
stored OK, so return with C=1
Busy rts
```

**Make a Subroutine Reentrant by Disabling Interrupts**

* Input param: Reg B contains an 8 bit message
* Output param: Reg CC (C bit) is 1 for OK, 0 for busy error

```
SEND   clc Initialize carry=0
tpa    save current interrupt state
psha   sei disable interrupts when vulnerable
tst Status check if mailbox is empty
bmi Busy full, so return with C=0
staa Message store
dec Status signify it is now contains a message
pula   OK, so return with C=1
psha   Busy pula restore interrupt status
```

**Disabling Interrupts in C**

```c
int Empty; /* -1 means empty, 0 otherwise */
int Message; /* data to be communicated */
int SEND(int data){ int OK;
char SaveSP;
asm("tpa\n staa %SaveSP\n sei"); /* critical */
OK=0; /* Assume it is not OK */
if(Empty){
    Message=data;
    Empty=0; /* Now contains a message*/
    OK=-1; /* Successfull */
    asm("ldaa %SaveSP\n tap"); /* end critical */
    return(OK);
}
```
A Binary Semaphore

* Global parameter: Semi4 is the mem loc to test and set
* If the location is zero, it will set it (make it -1)
  * and return Reg CC (Z bit) is 1 for OK
* If location is nonzero, return Reg CC (Z bit) = 0

Semi4 fcb 0  Semaphore is initially free
TAS  tst  Semi4  check if already set
 bne Out  busy, operation failed, return Z=0
 dec  Semi4  signify it is now busy
 bita #0  operation successful, return Z=1
Out  rts

Reentrant or Not?

- Must be able to recognize potential sources of bugs due to nonreentrant code in high-level languages.
- Is the following atomic?
  
  ```
  time++;  
  ```
  
- Yes, if the compiler generates:
  ```
  inc time  
  ```
  
- No, if the compiler generates:
  ```
  ldd time
  addd #1
  std time
  ```