Lecture 8: Interrupt Synchronization
Exam 1 on March 12
Interrupts place polling loops into hardware.
Interrupts allow efficient response to rare but important events.
Periodic interrupts are highly useful for data acquisition and control.
Interrupts can be more predictable than other approaches, or they can be less predictable, depending on how you structure the system.
Interrupts add *concurrency* to your embedded software and this can make your life very difficult.
What are Interrupts?

An automatic, hardware-supported transfer of software execution that is asynchronous with respect to current software.

Hardware can be external I/O device or internal event. When hardware needs service, it requests an interrupt. Calls *interrupt service routine* (ISR) that you write. Interrupt is terminated with `rti` instruction.

Interrupts communicate with `main` using shared memory. Each potential interrupt has separate `arm` bit. *Interrupt enable bit*, I, found in condition code register.
Dedicated vs. Shared
Dedicated vs. Shared

Wire- or negative-logic interrupt requests:
Can add additional I/O devices w/o redesigning H/W.
No limit to number of interrupting I/O devices.
Microcomputer hardware is simple.

Dedicated edge-triggered interrupt requests:
Software is simpler, easier to debug, and faster.
Less coupling between software modules.
Easier to implement priority.
Interrupt Service Routines (ISR)

Software executed when hardware requests an interrupt.

*Polled 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:

- Execution of main program is suspended.
- All registers are pushed onto the stack.
- The ISR is executed.
- The ISR executes `rti` instruction.
- All registers are restored from the stack.
- The main program is resumed.
Interrupt Execution

The diagram illustrates the process of handling an interrupt in a system. It shows a timeline with the states: Busy, Done, and Busy. The main thread is interrupted by hardware, which then needs service. The main thread saves its execution state and calls the ISR (Interrupt Service Routine) which provides service. Once the ISR completes its task, it restores the execution state of the main thread and the process continues.
## When to Use Interrupts

<table>
<thead>
<tr>
<th>Gadfly</th>
<th>Interrupts</th>
<th>DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictable</td>
<td>Variable arrival times</td>
<td>Low latency</td>
</tr>
<tr>
<td>Simple I/O</td>
<td>Complex I/O</td>
<td>High bandwidth</td>
</tr>
<tr>
<td>Fixed load</td>
<td>Variable load</td>
<td></td>
</tr>
<tr>
<td>No concurrency</td>
<td>Concurrent execution</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>
Communication with Interrupts

Interrupts have logically separate registers/stack, so communication must occur through global memory.
Input Device Interrupts

![Diagram of Input Device Interrupts]

- **Busy**: Input device busy
- **Done**: Input device done
- **Input device creates new data**: Causes an interrupt
- **Software reads data asks for another**: New input is ready

**Input device**
- Busy
- Done

**Interrupt service routine**
- b

**Main program**
- a
- c
- d
- a

**Elements in FIFO**
- Empty
- 1
- Empty
- 1
- Empty

**Input device creates new data**
- Busy
- Done
- Busy
- Done
- Busy
- Done
- Busy
- Done
- Busy
- Done

**Interrupt service routine**
- b
- b
- b
- b
- b

**Main program**
- a
- c
- d
- a

**Elements in FIFO**
- Empty
- 1
- Empty
- 1
- 2
- 3
- 2

**Time**

- a) main program waits because FIFO is empty
- b) ISR reads data and puts into FIFO
- c) main program gets data from FIFO
- d) main program processes data
Output Device Interrupts

Output device completes output operation
Causes an interrupt

Busy performing last output

Interrupt serviced
Software writes new data
Asks device to output it

Output is done

Output device
Interrupt service routine
Main program
Elements in FIFO

Disarmed Done Busy Done Disarmed Done Busy Done
a b a b a b a b a b
Empty 1 Empty 1 Empty

Disarmed Done Busy Done Busy
a b a b a b a b a b
Empty 1 0 1 2 3 4 5 4 5

a) main program generates data
b) main program puts data into FIFO, arms
c) ISR gets from FIFO and writes to port
d) ISR disarms itself because FIFO is empty

Time
Periodic interrupts are essential for implementing data acquisition and control systems.
ISR should only occur when needed, come in clean, perform function, and return right away.
Gadfly loops and iterations should be avoided in ISRs.
Percent of time in ISRs should be minimized.

*Interface latency* is time between new input available and when software reads the input data.

*Device latency* is response time of external I/O device.

A *real-time system* guarantees bounds on latency of important operations.
Reentrant Programming

A program segment is reentrant if it can be concurrently executed by two (or more) interrupts and/or main
Reentrant software must place local variables on stack, or else use mutual exclusion.
A nonreentrant subroutine has a section of code called a vulnerable window; an error occurs if:
   Main or an interrupt calls a nonreentrant subroutine and is executing inside the vulnerable window
   An interrupt calls same subroutine

Mutual exclusion is often implemented by disabling interrupts.
```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);
}
```
Reentrant or Not?

Must be able to recognize potential sources of bugs due to nonreentrant code in high-level languages.

Is the following atomic?

```c
#include <cassert>

int main()
{
    int time = 0;
    assert(time++ == 1);
    return 0;
}
```

Yes, if the compiler generates:

```c
inc time
```

No, if the compiler generates:

```c
ldd time
addd #1
std time
```
Atomic operation is one that once started is guaranteed to finish.

In most computers, machine instructions are atomic.

The following is atomic:

```plaintext
inc counter       where counter is global variable
```

The following is nonatomic:

```plaintext
ldaa counter      where counter is global variable
inca
staa counter
```
Read-Modify-Write Example

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

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

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

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

```c
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 */
}
```

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
addd temp    RegD=RegX+2*RegD
xgdx       RegX=RegX+2*RegD
rts
Nonatomic Multistep Write

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

```c
int info[2]; /* 32-bit global */

void set(int x, int y){
  info[0]=x;
  info[1]=y;
}
```

**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
Disabling Interrupts in C

```c
int Empty; /* -1 means empty, 0 means it contains something */
int Message; /* data to be communicated */
int SEND(int data){ int OK;
    char SaveSP;
    asm tpa
    asm staa SaveSP
    asm sei /* make atomic, entering critical */
    OK=0;     /* Assume it is not OK */
    if(Empty){
        Message=data;
        Empty=0; /* signify it is now contains a message*/
        OK=-1;}   /* Successfull */
    asm ldaa SaveSP
    asm tap     /* end critical section */
    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
We already talked about how device synchronization follows a protocol that permits the software and hardware state machines to interact.

Device protocols are written down in the hardware documentation.

Interrupts also have state machines, and the interaction of interrupts and main also follow a protocol.

However, since you are writing both sides of the protocol, it is probably not documented and you will be tempted to not think it through carefully, since you’re designing both sides of the protocol.

This is why interrupts are hard.