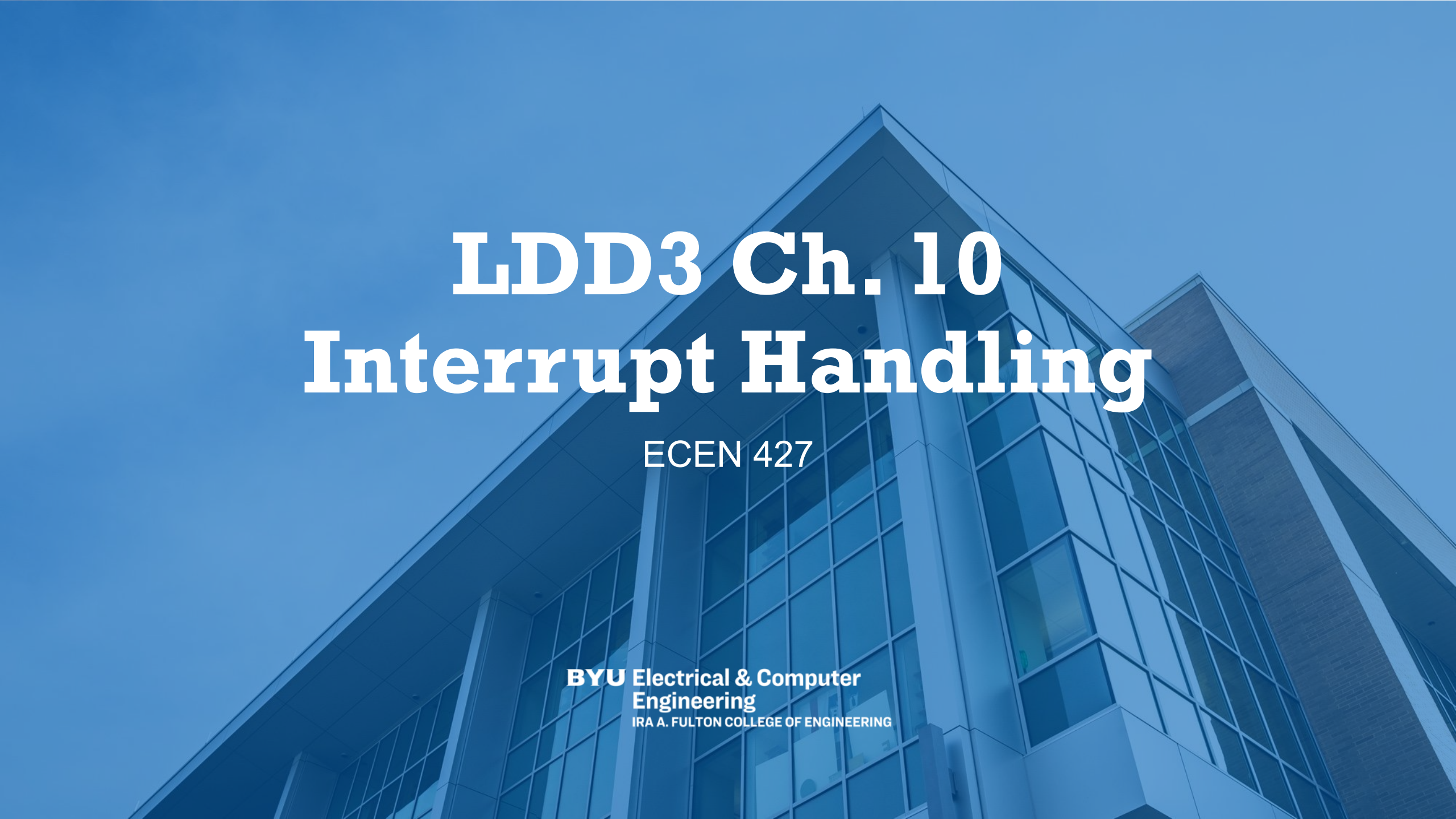




LDD3 Ch. 10

Interrupt Handling

ECEN 427

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

“Although some devices can be controlled using nothing but their I/O regions, most real devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different from, and far slower than, that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened. “

“That way, of course, is interrupts. An interrupt is simply a signal that the hardware can send when it wants the processor’s attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device’s interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.”

Register an interrupt handler

```
int request_irq(unsigned int irq,  
               irqreturn_t (*handler)(int, void *, struct pt_regs *),  
               unsigned long flags,  
               const char *dev_name,  
               void *dev_id);
```

```
void free_irq(unsigned int irq, void *dev_id);
```

unsigned int irq

The interrupt number being requested.

[We will look at where to get this next lecture](#)

```
irqreturn_t (*handler)(int, void *, struct pt_regs *)
```

The pointer to the handling function being installed.

const char *dev_name

The string passed to *request_irq* is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

void *dev_id

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, dev_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure. We'll see a practical use for dev_id in the section "Implementing a Handler."

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs  
                                *regs)  
{
```

- The interrupt number (int irq) is useful as information you may print in your log messages, if any.
- The second argument, void *dev_id, is a sort of client data
 - a void * argument is passed to request_irq, and this same pointer is then passed back as an argument to the handler when the interrupt happens.
 - You usually pass a pointer to your device data structure so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.
- Interrupt handlers should return **IRQ_HANDLED**
 - If you are sharing the interrupt line between multiple devices, only return this if your device was generating the interrupt. Otherwise return IRQ_NONE

Rules for an Interrupt Handler

- Runs in *interrupt context*, not on behalf of any process.
 - No **current**, no user address space.
- Cannot sleep
 - Cannot call functions that might sleep: `mutex_lock`, `msleep`, `wait_event`, `copy_from_user`, or `kmalloc`.
- Runs with interrupts disabled on this CPU.
 - Everything else on the board waits for you. Keep it short.
- You run in interrupt context, so you only get:
 - The argument to your handler: `dev_id`, which should be your device struct.
 - Global variables in the driver/module.
- Make the device stop asserting before you return. An interrupt you did not acknowledge fires again the instant you return.
 - Endless interrupt loop!!
- Shares data with other driver functions (e.g. read/write)?
 - You might have been interrupted in the middle of one of your driver functions
 - Your driver functions could be called from a process on another CPU core
 - You will need to synchronize/lock your data (more on this in Chapter 5)

Interrupt Ordering

- The correct place to call `request_irq` is when the device is first opened, before the hardware is instructed to generate interrupts.
- The place to call `free_irq` is the last time the device is closed, after the hardware is told not to interrupt the processor any more.

The /proc Interface

Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected. Reported interrupts are shown in `/proc/interrupts`. The following snapshot was taken on a two-processor Pentium system:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
```

	CPU0	CPU1		
0:	4848108	34	IO-APIC-edge	timer
2:	0	0	XT-PIC	cascade
8:	3	1	IO-APIC-edge	rtc
10:	4335	1	IO-APIC-level	aic7xxx
11:	8903	0	IO-APIC-level	uhci_hcd
12:	49	1	IO-APIC-edge	i8042
NMI:	0	0		
LOC:	4848187	4848186		
ERR:	0			
MIS:	0			

Let's look at `/proc/interrupts` on the PYNQ...