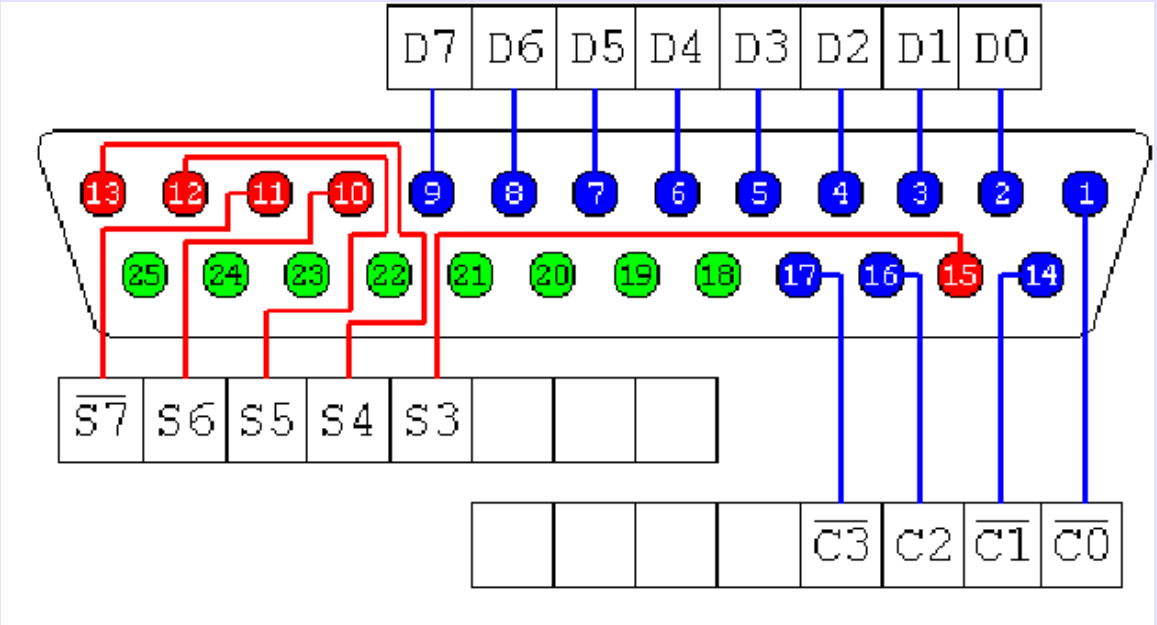
Day 3
Device Drivers Abstractions

Joel Isaacson
Ascender Technologies Ltd.
http://ascender.com

Today's Lectures

- Today we will address how user applications access devices that are connected to the “real” world.
- We will explore this via a series of increasingly more complex and abstract models (device drivers) of interaction with the physical world.

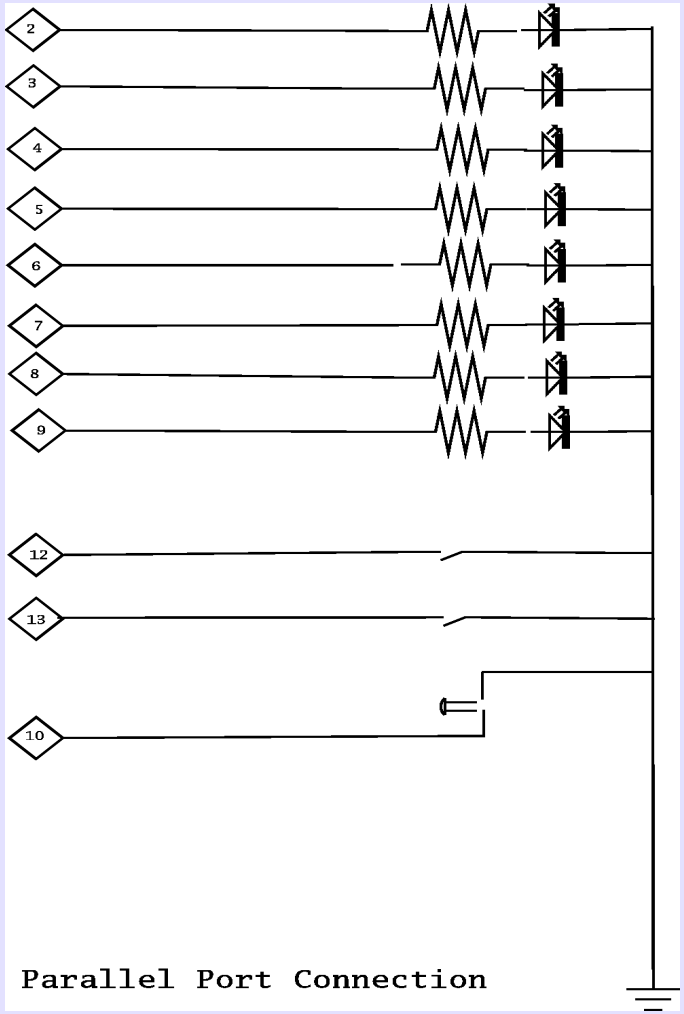
Parallel Port Hardware



- Three registers
  - Data
  - Status
  - Control

The Circuit

- This circuit is connected to the parallel port.
- Eight Leds
- Two switches
- One button



The Main Program

```
#include <stdio.h>
#include <time.h>
#include "pp.h"
main()
{
        static int t;
        struct timespec req = { 0, 1000*1000*100 };
        int i=0;
        int fd;

        fd= pp_init();
        while(1){
                int tt;
                pp_out(fd, 1<<(i++%8));
                tt= pp_in(fd);
                if(t != tt){
                        printf("%x %x\n", 0xff&t, 0xff&tt);
                        t=tt;
                }
                nanosleep(&req, NULL);
        }
}
```

In/Out Implementation

This implementation uses the inb/outb instructions to read/ write the parallel port registers

```
int pp_init()
{
  ioperm(0x378, 3, 1);
  outb(0x0,0x379);
  outb(0x0,0x37a);
  outb(0xff,0x378);
  return(0);
}

void pp_out(int fd,unsigned
char c)
{
  outb(c, 0x378);
}

unsigned char pp_in(int fd)
{
  return(inb(0x379));
}
```

The /dev/port Implementation

The /dev/port file is a file that when read/written will perform in/out operations.

```
int pp_init()
{
  return(open("/dev/port", O_RDWR));
}

void pp_out(int fd, unsigned char c)
{
  lseek(fd, 0x378, SEEK_SET);
  write(fd,&c,1);
}

unsigned char pp_in(int fd)
{
  unsigned char c;
  lseek(fd, 0x379, SEEK_SET);
  read(fd, &c, 1);
  return c;
}
```

The pp Driver Implementation

The device /dev/pp is installed as a module and it is device specific for the parallel port device.

```
int pp_init()
{
  return(open("/dev/pp", O_RDWR));
}

void pp_out(int fd, unsigned char c)
{
  write(fd,&c,1);
}

unsigned char pp_in(int fd)
{
  unsigned char c[10];
  read(fd, c, 1);
  return c[0];
}
```

The Device Driver - In Parts

```
/*
  Parallel port driver
 */

#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/init.h>
#include <linux/ioport.h>
#include <asm/io.h>

#include <asm/irq.h>
#include <asm/uaccess.h>
#include "ppreg.h"
```

The Device Driver - In Parts

```
static int pp_open(struct inode *inode, struct file *file)
{
  return 0;
}

static int pp_release(struct inode *inode, struct file *file)
{
  return 0;
}

static ssize_t pp_write(struct file *file, const char *data,
                        size_t len, loff_t *ppos)
{
  char c;
  int l= len;
  while(l-->0){
    get_user(c, data++);
    outb(c, PP_IOPORT);
  }
  return len;
}
```

The Device Driver - In Parts

```
static ssize_t pp_read(struct file *file, char *data,
                               size_t len, loff_t *ppos)
{
  char c;
  int l=len;
  while(l-->0){
    c=inb(PP_IOPORT+1);
    put_user(c,data++);
  }
  return len;
}

static struct file_operations pp_fops=
{
        owner:          THIS_MODULE,
        write:          pp_write,
        read:           pp_read,
        open:           pp_open,
        release:        pp_release,
};
```

The Device Driver - In Parts

```
static int __init pp_init(void)
{
  printk("pp driver initialized\n");
  if(! request_region(PP_IOPORT, 4, "parallel port")) {
    printk("pp: Can't allocate %x\n", PP_IOPORT);
    return(-EBUSY);
  }
  register_chrdev(150, "pp", &pp_fops);
  return(0);
}

static void __exit pp_exit(void)
{
  printk("pp driver exited\n");
  unregister_chrdev(150, "pp");
  release_region(PP_IOPORT, 4);
}
```

The Device Driver - In Parts

```
MODULE_AUTHOR("Joel Isaacson (joel@ascender.com)");
MODULE_DESCRIPTION("parallel port driver");
MODULE_LICENSE("GPL");

module_init(pp_init);
module_exit(pp_exit);
```

Loading the driver

- The driver can be loaded with the insmod command.
- The rmmod command unloads the driver.
- The special char file is made via the command:

```
mknod /dev/pp c 150 0
```

Adding Interrupts - driver2.c

We can add interrupt processing to the driver. The only thing that the interrupt routine does is print a string.

```
irqreturn_t pp_interrupt(int irq,
void *dev_id)
{
  static int i;
  inb(PP_IOPORT+1);
  printk("interrupt\n");
  return IRQ_HANDLED;
}

...

  request_irq(PP_IRQ, pp_interrupt,
                      0, "pp", NULL);

....
```

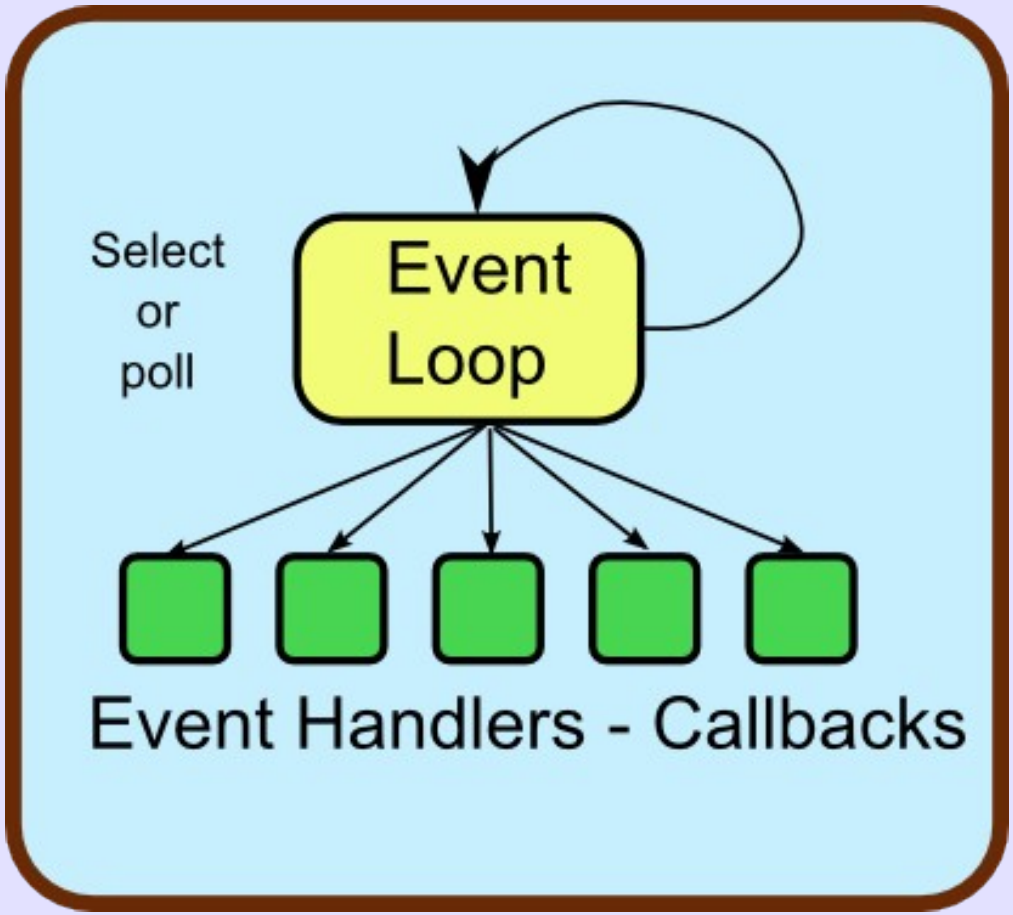
Too Many Interrupts

- We now get more than one interrupt for each button press.
- The problem is ringing in the button switch. This leads to “bouncing”.
- We can debounce the interrupt signal by setting a timer allow masking of multiple interrupts.
- This is done in driver3.c

Blocking API - driver4.c

- We have now changed the API to block on reads.
- The read only unblocks when an interrupt is generated.
- The poll operation is implemented.
- The program sel.c selects for both stdin and pp input.

Event Loop



ioctl - driver5.c

- driver5.c introduces an ioctl call to set a LED output mask.
- The program test_mask will test this option.

/proc - driver6.c

- The driver6.c file adds support for a file /proc/pp.
- This file contains the state of status lines.

Work Queue - driver7.c

- Work queue are a way of deferring kernel work into a kernel thread.
- This allows work that might block to be performed in a process context.
- By default work queue are executed by the kernel threads “event/n”.
- The debouncing is done on the work queue.

Work Queue - driver8.c

- In this example the write to the LEDS is performed in a work queue.