Operating systems for embedded systems

- Embedded operating systems
  - How do they differ from desktop operating systems?
- Programming model
  - Process-based
  - Event-based
  - How is concurrency handled?
  - How are resource conflicts managed?
- Programming languages
  - C/C++
  - Java/C#
  - Memory management
  - Atomicity in the presence of interrupts

Embedded Operating Systems

- Features of all operating systems
  - Abstraction of system resources
  - Managing of system resources
  - Concurrency model
  - Launch applications
- Desktop operating systems
  - General-purpose – all features may be needed
  - Large-scale resources – memory, disk, file systems
- Embedded operating systems
  - Application-specific – just use features you need, save memory
  - Small-scale resources – sensors, communication ports

System Resources on Typical Sensor Nodes

- Timers
- Sensors
- Serial port
- Radio communications
- Memory
- Power management

Abstraction of System Resources

- Create virtual components
  - E.g., multiple timers from one timer
- Allow them to be shared by multiple threads of execution
  - E.g., two applications that want to share radio communication
- Device drivers provide interface for resource
  - Encapsulate frequently used functions
  - Save device state (if any)
  - Manage interrupt handling

Very simple device driver

- Turn LED on/off
- Parameters:
  - port pin
- API:
  - on(port_pin) - specifies the port pin (e.g., port D pin 3)
  - off(port_pin)
- Interactions:
  - only if other devices want to use the same port

Simple device driver

- Turning an LED on and off at a fixed rate
- Parameters:
  - port pin
  - rate at which to blink LED
- API:
  - on(port_pin, rate)
    - specifies the port pin (e.g., port D pin 3)
    - specifies the rate to use in setting up the timer (what scale?)
  - off(port_pin)
- Internal state and functions:
  - keep track of state (on or off for a particular pin) of each pin
  - interrupt service routine to handle timer interrupt

Interesting interactions

- What if other devices also need to use timer (e.g., PWM device)?
  - timer interrupts now need to be handled differently depending on which device's alarm is going off
- Benefits of special-purpose output compare peripheral
  - output compare pins used exclusively for one device
  - output compare has a separate interrupt handling routine
- What if we don't have output compare capability or run out of output compare units?

Sharing timers

- Create a new device driver for the timer unit
  - Allow other devices to ask for timer services
  - Manage timer independently so that it can service multiple requests
- Parameters:
  - Time to wait, address to call when timer reaches that value
- API:
  - set_timer(time_to_wait, call_back_address)
    - Set call_back_address to correspond to time+time_to_wait
    - Compute next alarm to sound and set timer
    - Update in interrupt service routine for next alarm
- Internal state and functions:
  - How many alarms can the driver keep track of?
  - How are they organized? FIFO? priority queue?

Concurrency

- Multiple programs interleaved as if parallel
- Each program requests access to devices/services
  - e.g., timers, serial ports, etc.
- Exclusive or concurrent access to devices
  - allow only one program at a time to access a device (e.g., serial port)
  - arbitrate multiple accesses (e.g., timer)
- State and arbitration needed
  - keep track of state of devices and concurrent programs using resource
  - arbitrate their accesses (order, fairness, exclusivity)
  - monitors/locks (supported by primitive operations in ISA - test-and-set)
- Interrupts
  - disabling may effect timing of programs
  - keeping enabled may cause unwanted interactions

Handling concurrency

- Traditional operating system
  - multiple threads or processes
  - file system
  - virtual memory and paging
  - input/output (buffering between CPU, memory, and I/O devices)
  - interrupt handling (mostly with I/O devices)
  - resource allocation and arbitration
  - command interface (execution of programs)
- Embedded operating system
  - lightweight threads
  - input/output
  - interrupt handling
  - real-time guarantees

Embedded operating systems

- Lightweight threads
  - basic locks
  - fast context-switches
- Input/output
  - API for talking to devices
  - buffering
- Interrupt handling (with I/O devices and UI)
  - translate interrupts into events to be handled by user code
  - trigger new tasks to run (reactive)
- Real-time issues
  - guarantee task is called at a certain rate
  - guarantee an interrupt will be handled within a certain time
  - priority or deadline driven scheduling of tasks

Some Examples

embedded operating systems typically reside in ROM (flash) - changed rarely

- Pocket PC/WindowsCE/WindowsMobile
  - PDA operating system
  - spin-off of Windows NT
  - portable to a wide variety of processors (e.g., Xscale)
  - full-featured OS modularized to only include features as needed
- Wind River Systems VxWorks
  - one of the most popular embedded OS kernels
  - highly portable to an even wider variety of processors (tiny to huge)
  - modularized even further than the ones above (basic system under 50K)
- TinyOS
  - Open-source development environment specificall for small sensors
  - Simple (and tiny) operating system
    - Scheduler/event model of concurrency
    - Software components for efficient modularity
    - Software encapsulation for resources of sensor networks
  - Programming language and model – nesC

Embedded Linux

- iMote2 supports Linux, TinyOS, and SOS
  - Linux is the Familiar release originally developed for iPAQs (actually the DEC Itsy PDA by DEC Western Research Lab and then by Compaq's Cambridge lab)
- Linux kernel provides many utilities
  - Timer abstractions
  - File system
  - Serial communication
  - IP network communication
  - Memory management
- We can extend the kernel by registering new modules
  - These can control the internal registers of the XScale microcontroller

A Simple Application

- Blinking an LED at 1Hz – Lab 1 revisited
- Module to control LED GPIO pin
  - Sets state of LED
  - Uses timer
  - API allows setting of blink rate
- Register module in kernel
  - Assigned a "device number" by the Linux OS
- User-level application calls module API to start/stop/set

Kernel module

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/errno.h>
#include <linux/interrupt.h>
#include <asm/hardware.h>
#include <asm-arm/arch-pxa/pxa-regs.h> // This include file lets us access memory-mapped I/O registers
#include "blink.h"

#define OIER_E4         (1<<4)
#define RED             (1 << 7)

dev_t devId; // Contains the major and minor device numbers
struct cdev *cdev; // A kernel character device struct
int blink_ioctl(struct inode *, struct file *, unsigned int, unsigned long);
static void __exit unload_function(void);
struct file_operations blink_fops = {.owner = THIS_MODULE, .ioctl = blink_ioctl};

int state = 0; // State of the LED (on or off)
int delay = 16384; // Period is 1/16384 seconds * delay

int blink_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg) {
     switch (cmd) {
            case BLINK_SET_RATE:
                        OSMR4 = arg; // Update the match register
                        OSCR4 = 0; // Reset the counter
                        break;
            default:
                        return -ENOTTY;
     }
     return 0;
}
```


Kernel module (cont'd)

```
irqreturn_t blink_irq_handler(int irq, void *dev_id, struct pt_regs *regs) // called by kernel
{    // Check to see if this interrupt is for us
     if (!(OSSR & OIER_E4))
            return IRQ_RETVAL(IRQ_NONE);
     OSSR |= OIER_E4; // Acknowledge this interrupt
     state = !state;
     if (state)
            GPSR3 = RED;
     else
            GPCR3 = RED;
     return IRQ_RETVAL(IRQ_HANDLED);
}
static int __init init_function(void) // discardable by kernel
{
     int result;
     // Allocate a major device number for this driver
     result = alloc_chrdev_region(&devId, 0, 1, "blink");
     if (result < 0) return result;
     // Allocate a character devicw and set the owner and file operations of this new character device
     cdev = cdev_alloc();
     cdev->owner = THIS_MODULE;
     cdev->ops = &blink_fops;
     // Register the character device
     // Note at this point the device is live!
     result = cdev_add(cdev, devId, 1);
     result = request_irq(IRQ_OST_4_11, blink_irq_handler, 0, "blink", cdev);
     if (result < 0) {
            unload_function();
            return result;
     }
     // At this point, everything should succeed, so initialize the hardware
     OMCR4 = 0xC9; // Match against channel 4, periodic timer, reset on match,
                                        // period is 1 microsecond.
     OSMR4 = delay;
     OIER |= OIER_E4; // Enable interrupts for channel 4
     OSCR4 = 0; // Start the counter
     return 0; // SUCCESS
}
```

Kernel module (cont'd)

```
static void __exit unload_function(void)
{
    // Turn off interrupts
    OIER &= ~OIER_E4;
    free_irq(IRQ_OST_4_11, cdev);

    // Free the character device
    cdev_del(cdev);
    // Unregister the major device number
    unregister_chrdev_region(devId, 1);
}

// These two macros let the compiler and kernel know which functions
// should be called when loading and unloading the kernel.

module_init(init_function);
module_exit(unload_function);
```


Application

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "blink.h"

int main(int argc, char *argv)
{
    int rate, fd;

    if (argc != 2) {
          printf("Usage: %s <Hz>\n", argv[0]);
          return 0;
    }

    rate = (int)(16384 / atof(argv[1]));

    fd = open("/dev/blink", O_WRONLY);
    if (fd < 0) {
          printf("Unable to open /dev/blink\n");
          return 0;
    }

    ioctl(fd, BLINK_SET_RATE, rate);
    close(fd);
}
```

Labs 5/6

- Redo Lab 2 (blink instead of count)
- Modify module to do color of LED instead of rate of blink
  - Use timer to generate PWM signals instead of just on/off
  - Use the single timer to do R, G, and B
    (other timers used by other modules – Linux does not provide general timer utilities at that fine resolution)
- Redo Lab 3
  - Accelerometer module (mostly already there)
  - You'll write a GPIO interrupt handler to decode accel signals
  - Change color of LED accordingly (very similar code)
    - Use 3rd dimension of accel to do V instead of pot

Labs 5/6 (cont'd)

- Implement accel decoder and LED driver on two separate iMote2s
  - Radio communication between them
  - Send RGB or HSV values from one node to the other
  - Use 802.15.4 radio packets (format and API provided)
    - Define your own payload