



Embedded Operating Systems

Che-Wei Chang

chewei@mail.cgu.edu.tw

Department of Computer Science and Information
Engineering, Chang Gung University

Course Roadmap

Basic Concepts

- Embedded System Design Concepts
- Embedded System Developing Tools and Operating Systems
- Embedded Linux and Android Environment

Core Technology

- Real-Time System Design and Scheduling Algorithms
- System Synchronization Protocols

Real Implementation

- System Initialization and Memory Management
- Power Management Techniques and System Routine
- Embedded Linux Labs and Exercises on Android



Real-Time Operating Systems

Real Time OS

- ▶ An RTOS is an abstraction from hardware and software programming
 - Shorter development time
 - Less porting efforts
 - Better reusability
- ▶ Choosing an RTOS is important
 - High efforts when porting to a different OS
 - The chosen OS may have a high impact on the amount of resources needed

Soft Real-Time Systems

- ▶ With Soft Real-Time Systems
 - Missed deadlines are not fatal
 - Often have a human in the loop
- ▶ Example:
 - Multimedia applications
 - If the frame-rate of a video clip is lower than 30 frame/sec, the user still can watch the video
 - An automatic teller machine (ATM)
 - If the ATM takes 30 seconds longer than the ideal, the user still won't walk away

Hard Real-Time Systems

- ▶ If the deadline is missed, data is permanently lost or people might get hurt
- ▶ Often, these systems are fully autonomous
- ▶ Examples:
 - Air bag deployment
 - Anti-lock brake system
 - Nuclear power plant controller

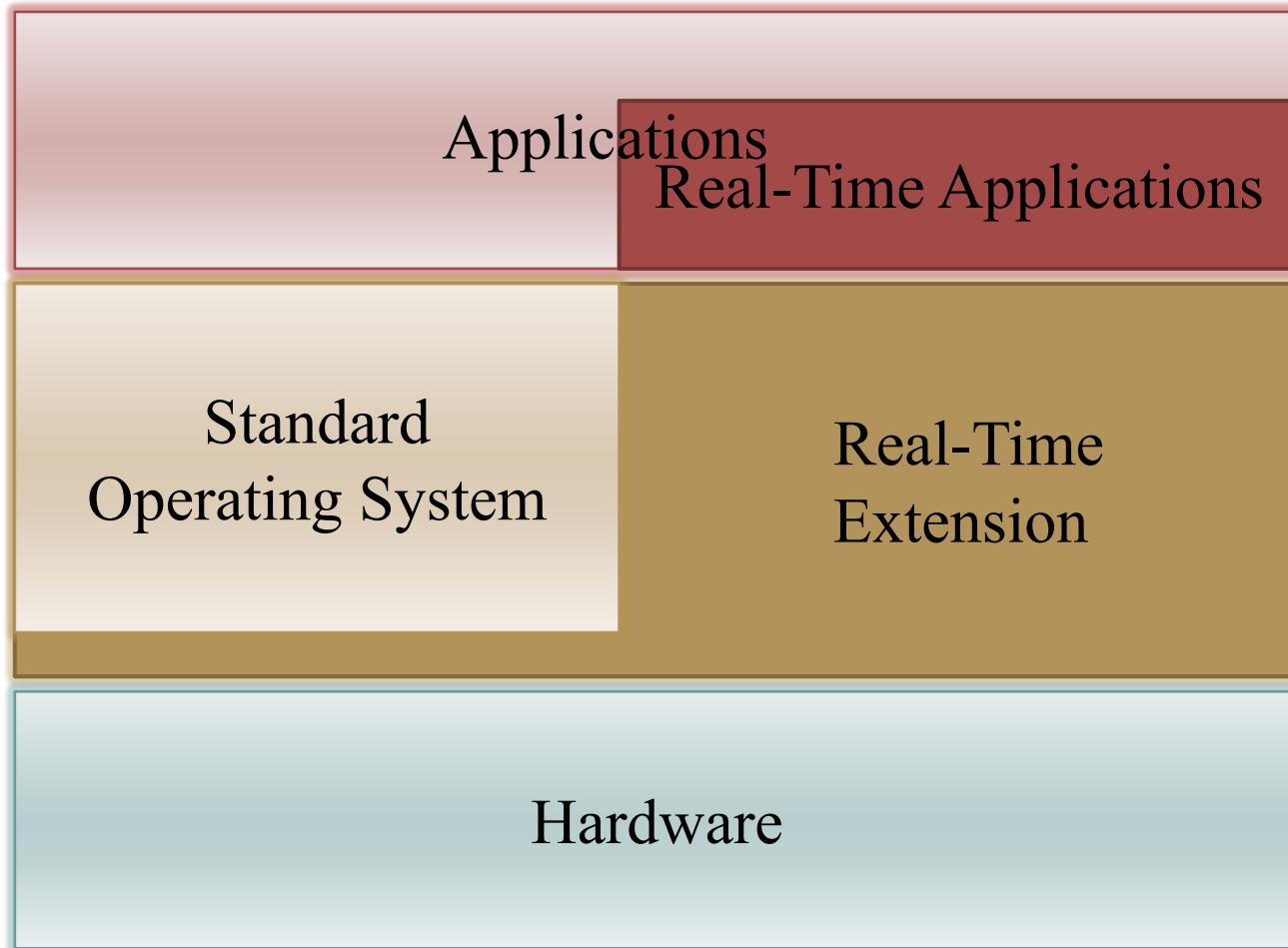
Pure Real-Time OS

- ▶ Especially designed for real-time requirements
- ▶ Completely real-time compliant
- ▶ Often usable for simple architecture
- ▶ Advantage:
 - No or little overhead of computing power and memory
- ▶ Disadvantage:
 - Limited functionality
- ▶ Examples:
 - eCos, Nucleus, VxWork, QNX, uC/OS II

Real-Time Extension of General OS

- ▶ Extension of an OS by real-time components
- ▶ Cooperation between RT-and non-RT parts
- ▶ Advantages:
 - Rich functionality
- ▶ Disadvantage:
 - No general real-time ability
 - Need more computing and memory resources
- ▶ Example:
 - RT-Linux, Solaris

Picture of Real-Time Extension



Components of a RTOS

- ▶ Process Management
 - Real-Time Scheduler
 - Synchronization Mechanism
 - Inter-Process Communication (IPC)
 - Semaphores
- ▶ Memory Management
- ▶ Interrupt Service Mechanism
- ▶ I/O Management
- ▶ Development Environments



Case Studies of RTOS

Introduction of VxWorks

- ▶ Manufacturer: Wind River System
 - Largest player on the market
 - Proprietary software
- ▶ Target Platforms:
 - x86, MIPS, PowerPC, SPARC, ARM, ...
- ▶ Application Examples:
 - Transport systems: Airbus A400M, AH-64 Apache, BMW iDrive
 - Spacecraft: Phoenix Mars Lander (2008), Curiosity Rover (2012), Yutu Rover (2013)
 - Robots and programmable controllers, networking and communication components, printers, copiers, and image processing



Writing C Code on VxWorks

- ▶ VxWorks consists of threads (called "tasks")
 - VxWorks does not start at a main function
 - Every global function can be called from the shell
- ▶ Every global function or variable is global to the whole system
- ▶ Every function can access to every memory location
 - Every other global function and variable can be accessed
 - Writing to a NULL pointer can corrupt the interrupt table
 - Stack overflow can crash the system

Introduction of Real-Time Linux

- ▶ What is RTLinux
 - It is a hard real-time RTOS microkernel
 - It runs the entire Linux operating system as a fully preemptive process
- ▶ The Key Ideas
 - To be hard real-time, the execution time of each component should be deterministic
 - Each real-time task can use only the device drivers with real-time support
 - Other tasks can use the whole functions of Linux and can not lock device without the monitoring of RTLinux

Modules of Real-Time Linux

- ▶ A priority scheduler that supports both a "lite POSIX" interface and the RTLinux API
- ▶ A timer which controls the processor clocks and exports an abstract interface for connecting handlers to clocks
- ▶ A module supports POSIX read/write/open interface to device drivers
- ▶ A module connects real-time tasks and interrupt handlers to Linux processes through a device layer so that Linux processes can read/write to RT components
- ▶ A package of semaphore which is used among real-time tasks
- ▶ A module shares memory between real-time components and Linux processes

Introduction of μC/OS-II (1 / 2)

- ▶ The name is from micro-controller operating system, version 2
- ▶ μC/OS-II is certified in an avionics product by FAA in July 2000 and is also used in the Mars Curiosity Rover
- ▶ It is a very small real-time kernel
 - Memory footprint is about 20KB for a fully functional kernel
 - Source code is about 5,500 lines, mostly in ANSI C
 - Its source is open but not free for commercial usages
- ▶ Preemptible priority-driven real-time scheduling
 - 64 priority levels (max 64 tasks)
 - 8 reserved for μC/OS-II
 - Each task is an infinite loop



Introduction of μC/OS-II (2/2)

- ▶ Deterministic execution times for most μC/OS-II functions and services
- ▶ Nested interrupts could go up to 256 levels
- ▶ Supports of various 8-bit to 64-bit platforms: x86, 68x, MIPS, 8051, etc.
- ▶ Easy for development: Borland C++ compiler and DOS (optional)
- ▶ However, uC/OS-II still lacks of the following features:
 - Resource synchronization protocol
 - Soft-real-time support

The μC/OS-II File Structure

Application Code (Your Code!)

Processor Independent Implementations

- Scheduling policy
- Event flags
- Semaphores
- Mailboxes
- Event queues
- Task management
- Time management
- Memory management

Application Specific Configurations

- OS_CFG.H
- Max # of tasks
- Max Queue length
- ...

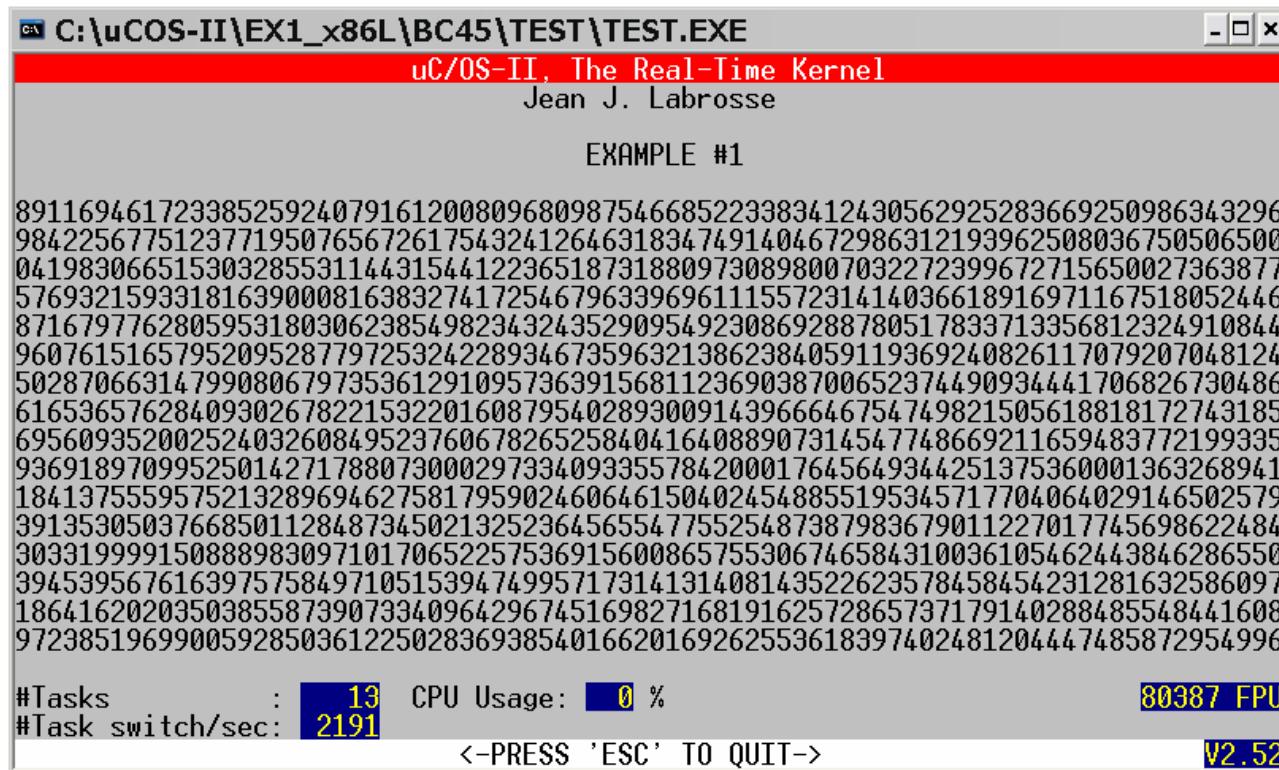
uC/OS-II Port for Processor Specific Codes

Software
Hardware

CPU

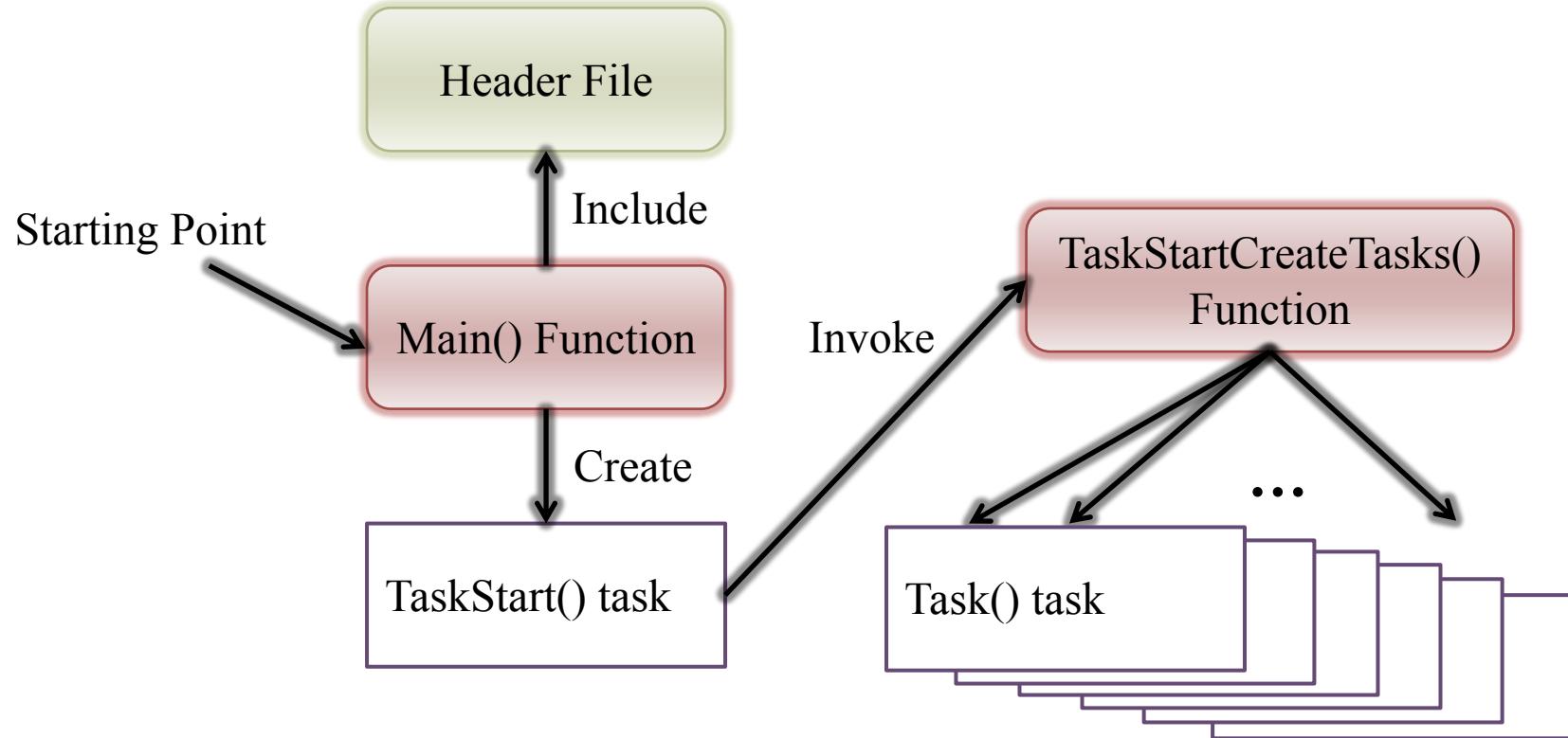
Timer

An Example on μC/OS-II: Multitasking



- ▶ Three System Tasks
- ▶ Ten Application Tasks Randomly Print Its Number

Multitasking: Workflow



Multitasking: Header File

```
#include "includes.h"
/*
*****
CONSTANTS
*****
*/
#define TASK_STK_SIZE 512
#define N_TASKS 10
/*
*****
VARIABLES
*****
*/
OS_STK TaskStk[N_TASKS][TASK_STK_SIZE];
OS_STK TaskStartStk[TASK_STK_SIZE];
char TaskData[N_TASKS];
OS_EVENT *RandomSem;
```

Multitasking: Main()

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + ISP_BGND_BLACK);
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    RandomSem = OSSemCreate(1);
    OSTaskCreate( TaskStart,
                  (void *)0,
                  (void *)&TaskStartStk[TASK_STK_SIZE-1],
                  0);
    OSStart();
}
```

Entry point of the task
(a pointer to function)

User-specified data

Top of stack

Priority (0=highest)

Multitasking: TaskStart()

```
void TaskStart (void *pdata)
{
    /*skip the details of setting*/
    OSStatInit();
    TaskStartCreateTasks();
    for (;;)
    {
        if (PC_GetKey(&key) == TRUE)
        {
            if (key == 0x1B) { PC_DOSReturn(); }
        }
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
```

Call the function to
create the other tasks

See if the ESCAPE
key has been pressed

Wait one second

Multitasking: TaskStartCreateTasks()

```
static void TaskStartCreateTasks (void)
```

```
{
```

```
    INT8U i;
```

```
    for (i = 0; i < N_TASKS; i++)
```

```
{
```

```
        TaskData[i] = '0' + i;
```

```
        OSTaskCreate(
```

```
            Task,
```

```
            (void *)&TaskData[i],
```

```
            &TaskStk[i][TASK_STK_SIZE - 1],
```

```
            i + 1 );
```

Top of stack

Priority

Entry point of the task
(a pointer to function)

Argument:
character to print

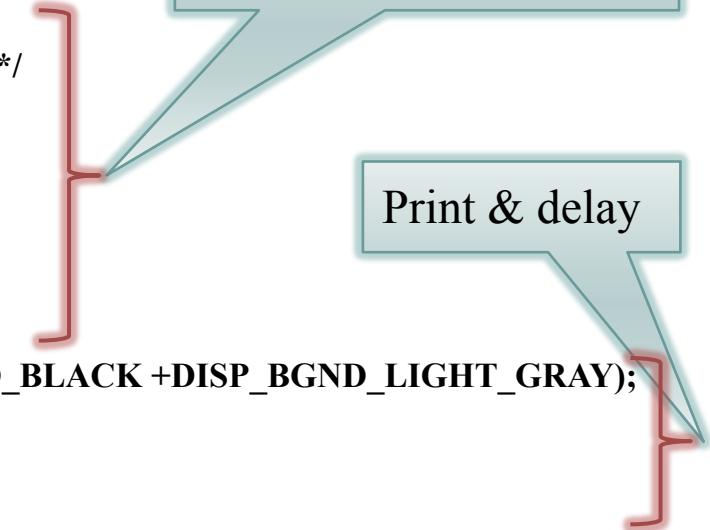
```
}
```

Multitasking: Task()

```
void Task (void *pdata)
{
    INT8U x;
    INT8U y;
    INT8U err;
    for (;;)
    {
        OSSemPend(RandomSem, 0, &err);
        /* Acquire semaphore to perform random numbers */
        x = random(80);
        /* Find X position where task number will appear */
        y = random(16);
        /* Find Y position where task number will appear */
        OSSemPost(RandomSem);
        /* Release semaphore */
        PC_DisPChar(x, y + 5, *(char *)pdata, DISP_FGND_BLACK +DISP_BGND_LIGHT_GRAY);
        /* Display the task number on the screen */
        OSTimeDly(1);
        /* Delay 1 clock tick */
    }
}
```

Randomly pick up the position to print its data

Print & delay



OSinit()

(\SOFTWARE\uCOS-II\EX1_x86L\BC45\SOURCE\OS_CORE.C)

- ▶ Initialize the internal structures of μC/OS-II and MUST be called before any services
- ▶ Internal structures of μC/OS-2
 - Task ready list
 - Priority table
 - Task control blocks (TCB)
 - Free pool
- ▶ Create housekeeping tasks
 - The idle task
 - The statistics task

PC_DOSSaveReturn()

(\SOFTWARE\BLOCKS\PC\BC45\PC.C)

- ▶ Save the current status of DOS for the future restoration
 - Interrupt vectors and the RTC tick rate
- ▶ Set a global returning point by calling setjump()
 - μC/OS-II can come back here when it terminates.
 - PC_DOSReturn()

PC_VectSet(uCOS,OSCtxSw)

(\SOFTWARE\BLOCKS\PC\BC45\PC.C)

- ▶ Install the context switch handler
- ▶ Interrupt 0x08 (timer) under 80x86 family
 - Invoked by INT instruction

OSStart()

(SOFTWARE\uCOS-II\EX1_x86L\BC45\SOURCE\CORE.C)

- ▶ Start multitasking of μC/OS-2
- ▶ It never returns to main()
- ▶ μC/OS-II is terminated if PC_DOSReturn() is called

Conclusion of μC/OS-II

- ▶ Operating System Contents
 - Data structure of each OS component
 - Basic functions of task scheduling and resource management
 - Other fundamental supports of OS
- ▶ Application Format
 - Each task is an infinite loop
 - Ready tasks execute according to their priorities
- ▶ Porting Efforts
 - CPU and timer setting
 - Interrupt handler