MicroC/OS-II

The Kernel, Inside Story……

Santosh Sam Koshy

National Ubiquitous Computing Research Centre

C-DAC, Hyderabad

MicroC-OS/II

- MicroC/OS-II is the acronym for Micro-Controller Operating Systems Version 2
- It is a **priority-based pre-emptive** real-time multitasking operating system kernel
- Memory footprint is about **20KB** for a fully functional kernel
- It is ported to more than **100** micro-processors and micro-controllers
- Source code is written mostly in ANSI C
- Currently, MicroC-OS/III is released and is commercially distributed

uCOS-II Features

- **Portable**
  - *Portable ANSI C, minimum microprocessor-specific assembler*
- **ROMable**
  - *Designed for Embedded Applications, and with the proper tool chain, it can be embedded to any part of the product*
- **Scalable**
  - *Can be scaled to target various target applications based on the services required by that application*
- **Pre-emptive**
  - *uCOS-II is a fully pre-emptive real-time kernel*

uCOS-II Features

- **Multitasking**
  - *uCOS-II (latest version v2.80) can manage up to 256 tasks, uCOS-III can manage unlimited tasks*
- **Deterministic *(time)***
  - *Execution time of most uCOS-II functions and services are deterministic*
- **Deterministic *(space)***
  - *Each task maintains its own, different stack size*
- **Services**
  - *Mailboxes, Queues, Semaphores, Fixed-sized memory partitions, Time-related functions*

uCOS-II Features

- **Interrupt Management**
  - *Interrupts can cause higher priority tasks to be READY*
  - *Interrupts can be nested 255 levels deep*
- **Robust and Reliable**
  - *Has been developed and deployed on hundreds of commercial applications since 1992*
- **Task Stacks**
  - *Each task requires its own stack.*
  - *Micro C/OS-II allows tasks to maintain variable sized stacks*
  - *This allows applications the flexibility of making an efficient use of the available RAM*

uCOS-II Features

- uCOS-II supports **priority inheritance**.
- With uCOS-II, all tasks must have a unique priority.
- It is a **pre-emptive** Kernel
- Does not support Round Robin Scheduling (Unique Priority)

The MicroC/OS-II Kernel Details

Critical Sections

- RT Kernels needs to **disable Interrupts** in order to **access critical sections** of the code and **re-enable Interrupts** when done
- *This allows UCOS-II to protect critical code from being entered simultaneously from either multiple tasks or ISRs*
- The **Interrupt disable time** is one of the most **important specifications** that has to be provided by vendor *(responsiveness of system)*

Critical Sections

- UCOS-II defines two MACROS to disable & enable Interrupts
  - **OS_ENTER_CRITICAL()**
  - **OS_EXIT_CRITICAL()**

- Because these are processor specific, they are found in the file called **OS_CPU.H**

- *Each processor thus has its own OS_CPU.H*

Critical Sections

- The application can also use OS_ENTER_CRITICAL() and OS_EXIT_CRTITCAL() to protect critical sections between two user defined tasks.

- *This is prone to errors and care should be taken in implementing as such. For example, on protecting a critical section between two tasks, calls to timer interrupts such as OSTimeDly should not be issued.*

Critical Sections

- OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() can be implemented in three ways depending on the capabilities rendered by the processor and compiler used

- The method used is selected by the #define constant OS_CRITICAL_METHOD which is defined in OS_CPU.h

Critical Sections - Method I

- **# define OS_CRITICAL_METHOD == 1**
  - Invoke the processor instruction to disable and enable interrupts in the OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL macros respectively
    - *#define OS_ENTER_CRITICAL() asm volatile 'CLI'*
    - *#define OS_EXIT_CRITICAL() asm volatile 'STI'*

Using OS_CRITICAL_METHOD = 1

Task Function ()

```
{
    OS_ENTER_CRITICAL();

    Kernel Function ();

    OS_EXIT_CRITICAL()
}
```

Kernel Function ()

```
{
    OS_ENTER_CRITICAL();
    Do Something
    OS_EXIT_CRITICAL();
}
```

1

2

3

Critical Section is Vulnerable

Critical Sections -Method II

- **OS_CRITICAL_METHOD == 2**
  - The solution of the above problem is in saving the status of the interrupt disable onto the stack, before disabling the interrupts.
  - *#define OS_ENTER_CRITICAL() *
    - *asm ("PUSH PSW"); *
    - *asm (" DI");*
  - *#define OS_EXIT_CRITICAL() *
    - *asm (" POP PSW");*

Critical Sections -Method III

- **OS_CRITICAL_METHOD == 3**
  - Some compilers provide extensions to access the PSW from functions in the code. This may be stored as a local variable in the function and used later.

```
void my_function(arguments)
{
   OS_CPU_SR local_PSW;
   local_PSW = get_processor_PSW();
   disable interrupts
      Critical Section
   set_processor_PSW(local_PSW);
}
```

Tasks

- Task is typically a **infinite loop function**
- A task looks like a C function containing a return type & an argument, but it never returns
- The return type should always be ***void***
- The argument is a void pointer that allows a user to pass different types depending on the context

Task Example

```
void    ourTask(void *pData)
{
        for(;;){
                /* USER CODE */
                ….
                /* UCOS-II Services */
                ….
                /* USER CODE */
        }
}
```

Tasks

- A task can delete itself upon completion
- Note that the code is not actually deleted
- UCOS-II simply doesn’t know about the task anymore so the task code will not run
- Also if task calls **OSTaskDel()**, the task never returns

Task that deletes itself

```
void  ourTask(void *pData)
{
    /* USER CODE */

    OSTaskDel(OS_PRIO_SELF) ;
}
```

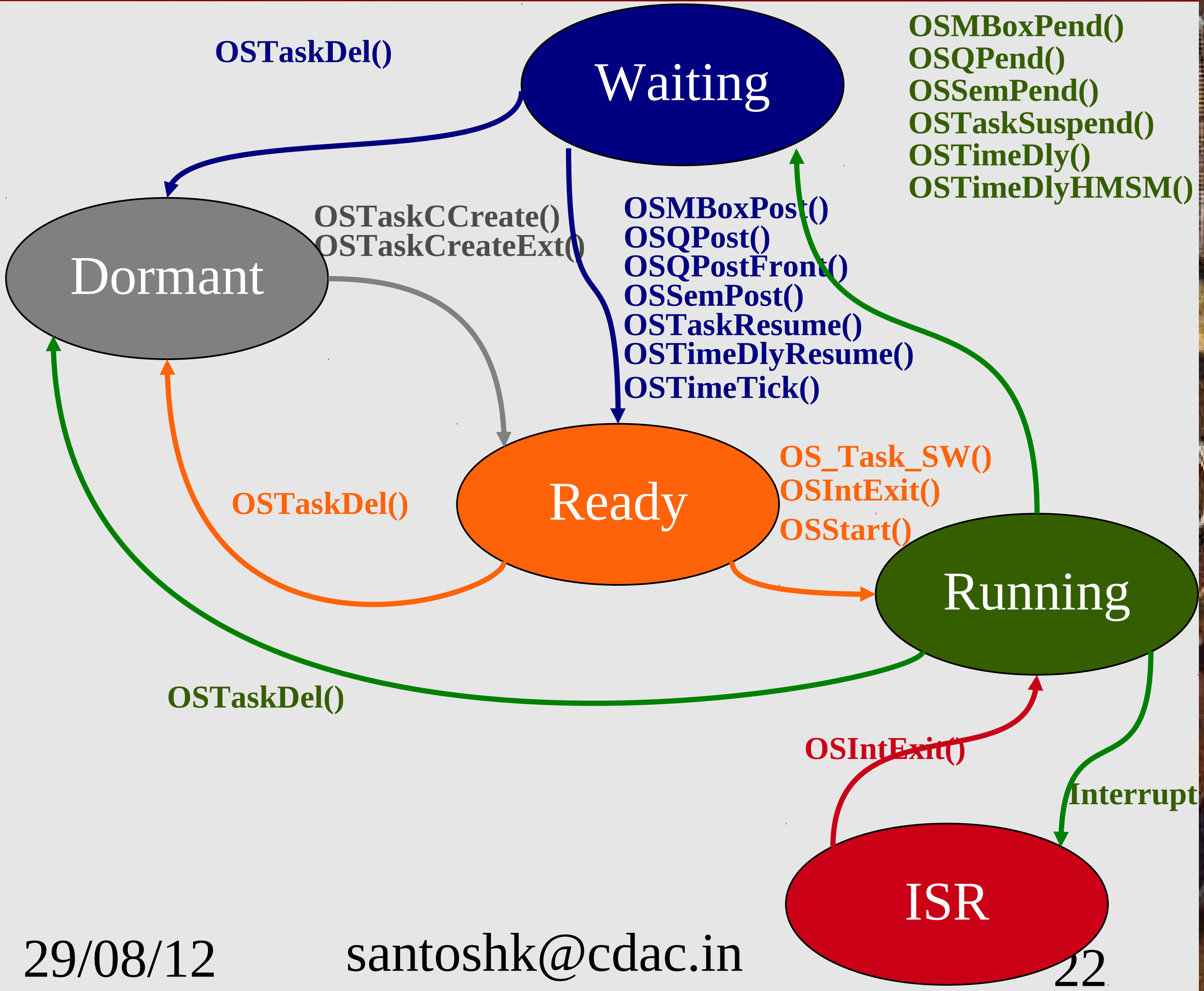
Tasks

- UCOS-II *(v2.50)* can manage upto **64 tasks** *(and v2.53 onwards supports 256 tasks)*
- Current version uses **two** tasks as **system tasks**
- *Reserved priorities (for future use) are 0, 1, 2, 3, OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1 and OS_LOWEST _PRIO*
- OS_LOWEST_PRIO defined in **OS_CFG.H**

Task Priority

- Each task in our application should be assigned unique priority level from 0 to OS_LOWEST_PRIO-2
- The lower the priority number the highest is the priority
- Priority number also serves as task Identifier

Task States


Waiting
OSTaskDel()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
Dormant
OSTaskCCreate()
OSTaskCreateExt()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
Ready
OSTaskDel()
OS_Task_SW()
OSIntExit()
OSStart()
Running
OSTaskDel()
OSIntExit()
Interrupt
ISR

Task Creation

- A task is created by calling either

    INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);

  - **OSTaskCreateExt();**

- When a task is created, it is made **READY** to run

- *Task can be created before Multitasking starts or dynamically by a running task*

Important

- If task is created dynamically by another task and if the created task has higher priority the new task is given control over the CPU immediately

Delaying a Task

- The running task may delay itself for a certain amount of time by calling either
  - **OSTimeDly();**
  - **OSTimeDlyHMSM();**
  - This task is **WAITING** for some time to expire & the next high priority task that is ready to run is given the control of the CPU immediately

Starting the OS

- Multitasking is started by calling **OSStart()**. This function must be called only once during startup.
- OSStart() runs the highest priority task that is ready to run. This task is thus placed in the **RUNNING** State
- A ready task will not run until all higher priority task are placed in either the wait state or are deleted

The Main Function

```
INT16S main (void)
{
   HardwareInit(); //Initialize Timers, Ports,  Etc..
   OSInit(); //Initialize OS services
   OSTaskCreate (); //Create the Tasks
   OSStart();   // Start multitasking   kernel
   return 0;
}
```

The End