Micrium

µC/OS-II, The Real-Time Kernels

and the ARM7 / ARM9

Jean J. Labrosse

www.Micrium.com

My Background

- Master's Degree in Electrical Engineering
- Wrote two books (µC/OS-II and ESBB)
- Wrote many papers for magazines
  - Embedded Systems Programming
  - Electronic Design
  - C/C++ User's Journal
  - ASME
  - Xcell Journal
- Have designed Embedded Systems for over 20 years
- President of Micriμm
  - Provider of Embedded Software Solutions

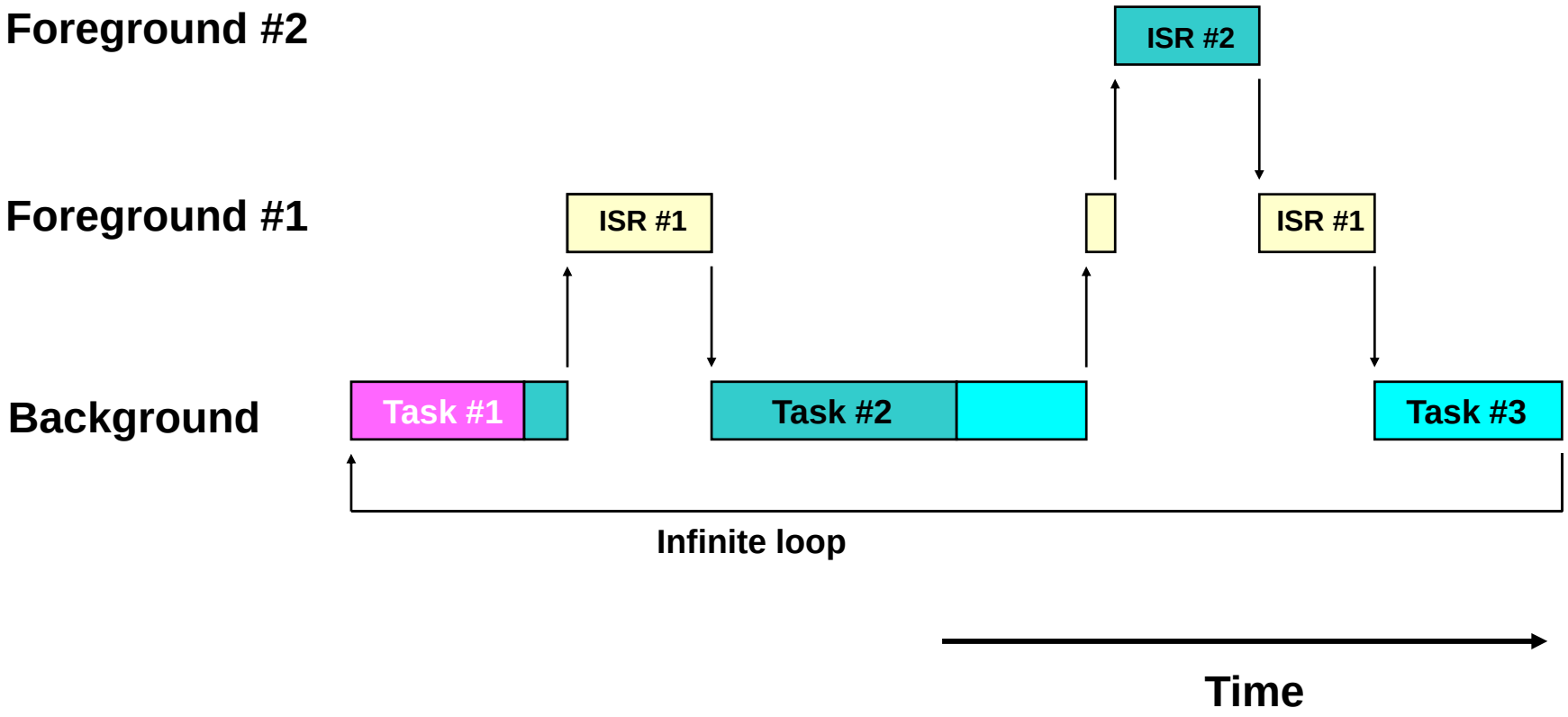
Part I

Foreground/Background Systems

µC/OS-II, The Real-Time Kernels

Task Management

Products without Kernels
(Foreground/Background Systems)



Foreground/Background

```
/* Background */
void main (void)
{
  Initialization;
  FOREVER {
    Read analog inputs;
    Read discrete inputs;
    Perform monitoring functions;
    Perform control functions;
    Update analog outputs;
    Update discrete outputs;
    Scan keyboard;
    Handle user interface;
    Update display;
    Handle communication requests;
    Other...
  }
}
```

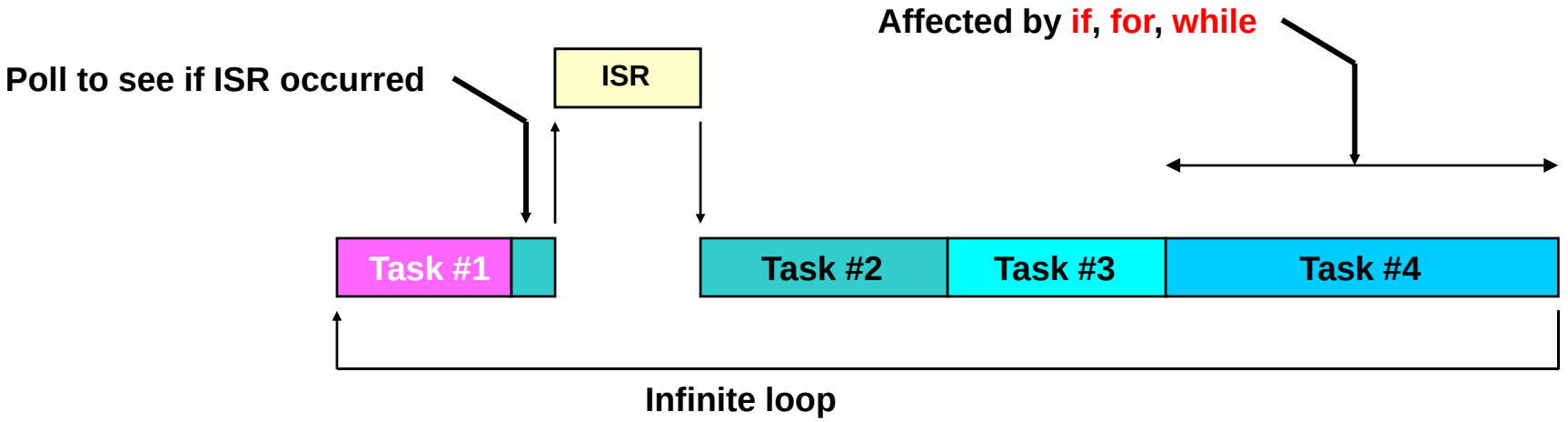
```
/* Foreground */
ISR (void)
{
    Handle asynchronous event;
}
```

Foreground/Background
Advantages

- Used in low cost Embedded Applications
- Memory requirements only depends on your application
- Single stack area for:
  - Function nesting
  - Local variables
  - ISR nesting
- Minimal interrupt latency
- Low Cost
  - No royalties to pay to vendors

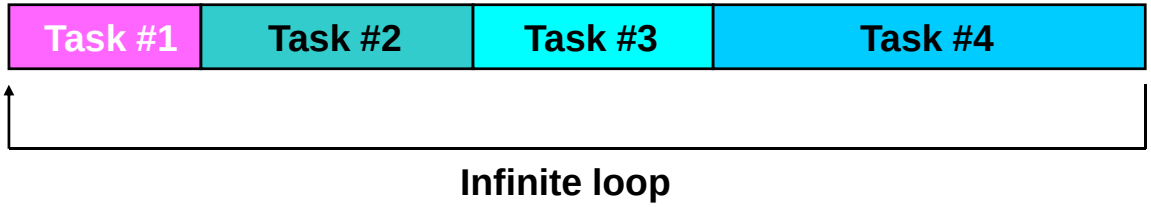
Foreground/Background
Disadvantages

- Background response time is the background execution time
  - Non-deterministic
    - Affected by **if**, **for**, **while** ...
  - May not be responsive enough
  - Changes as you change your code



Foreground/Background
Disadvantages

- All ‘tasks’ have the same priority!
  - Code executes in sequence
  - If an important event occurs it’s handled at the same priority as everything else!
  - You may need to execute the same code often to avoid missing an event.



- You have to implement all services:
  - Time delays and timeouts
  - Timers
  - Message passing
  - Resource management
- Code is harder to maintain and can become messy!

Part I

Foreground/Background Systems

µC/OS-II, The Real-Time Kernels

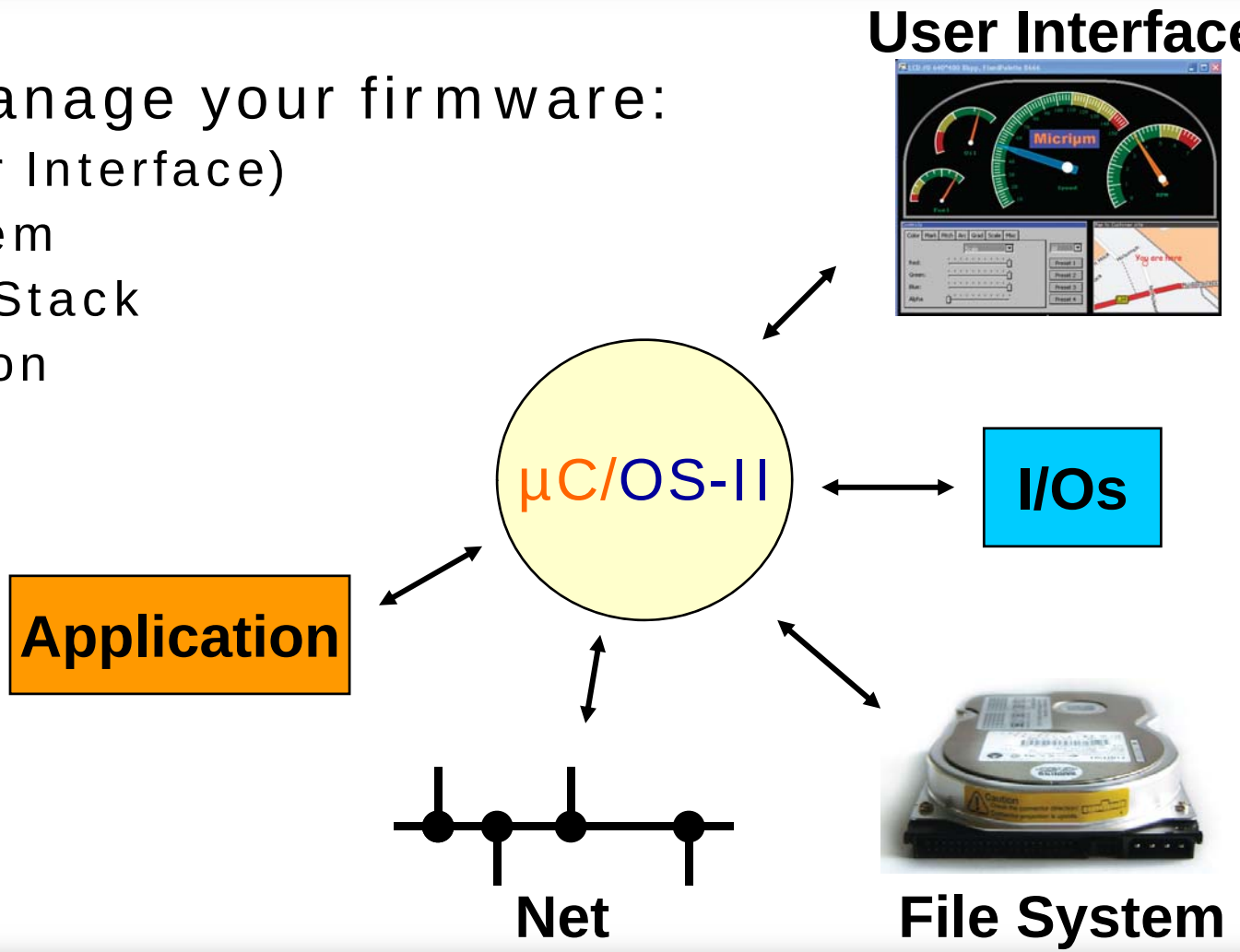
Task Management

What is µC/OS-II?

- Software that manages the time of a microprocessor or microcontroller.
  - Ensures that the most important code runs first!
- Allows Multitasking:
  - Do more than one thing at the same time.
  - Application is broken down into multiple tasks each handling one aspect of your application
  - It's like having multiple CPUs!
- Provides valuable services to your application:
  - Time delays
  - Resource sharing
  - Intertask communication and synchronization

Why use µC/OS-II?

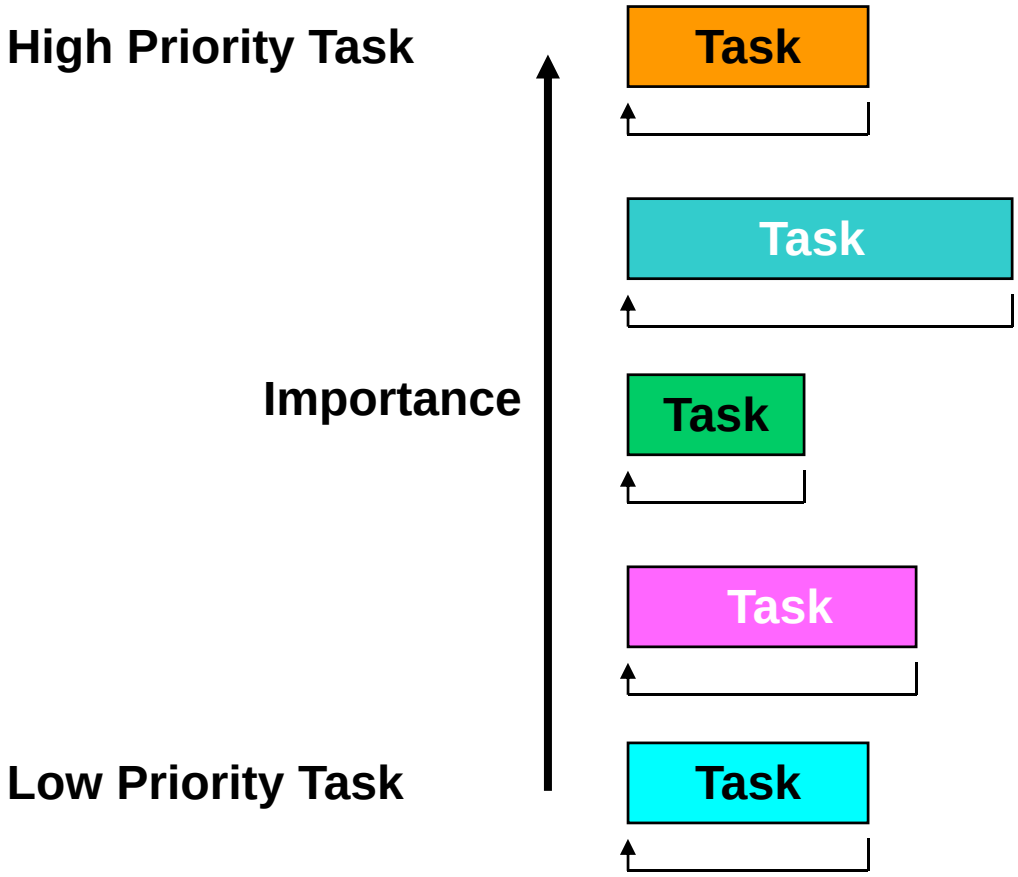
- To help manage your firmware:
  - GUI (User Interface)
  - File System
  - Protocol Stack
  - Application
  - I/Os



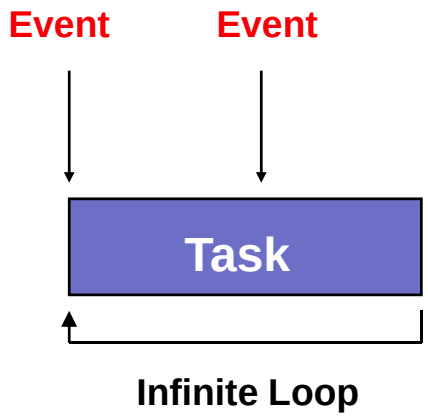
Why use µC/OS-II?

- To be more responsive to real-time events
- To prioritize the work to be done by the CPU
- To simplify system expansion
  - Adding low-priority tasks generally does not change the responsiveness to higher priority tasks!
- To reduce development time
- To easily split the application between programmers
  - Can simplify debugging
- To get useful services from the kernel
  - Services that you would want to provide to your application code

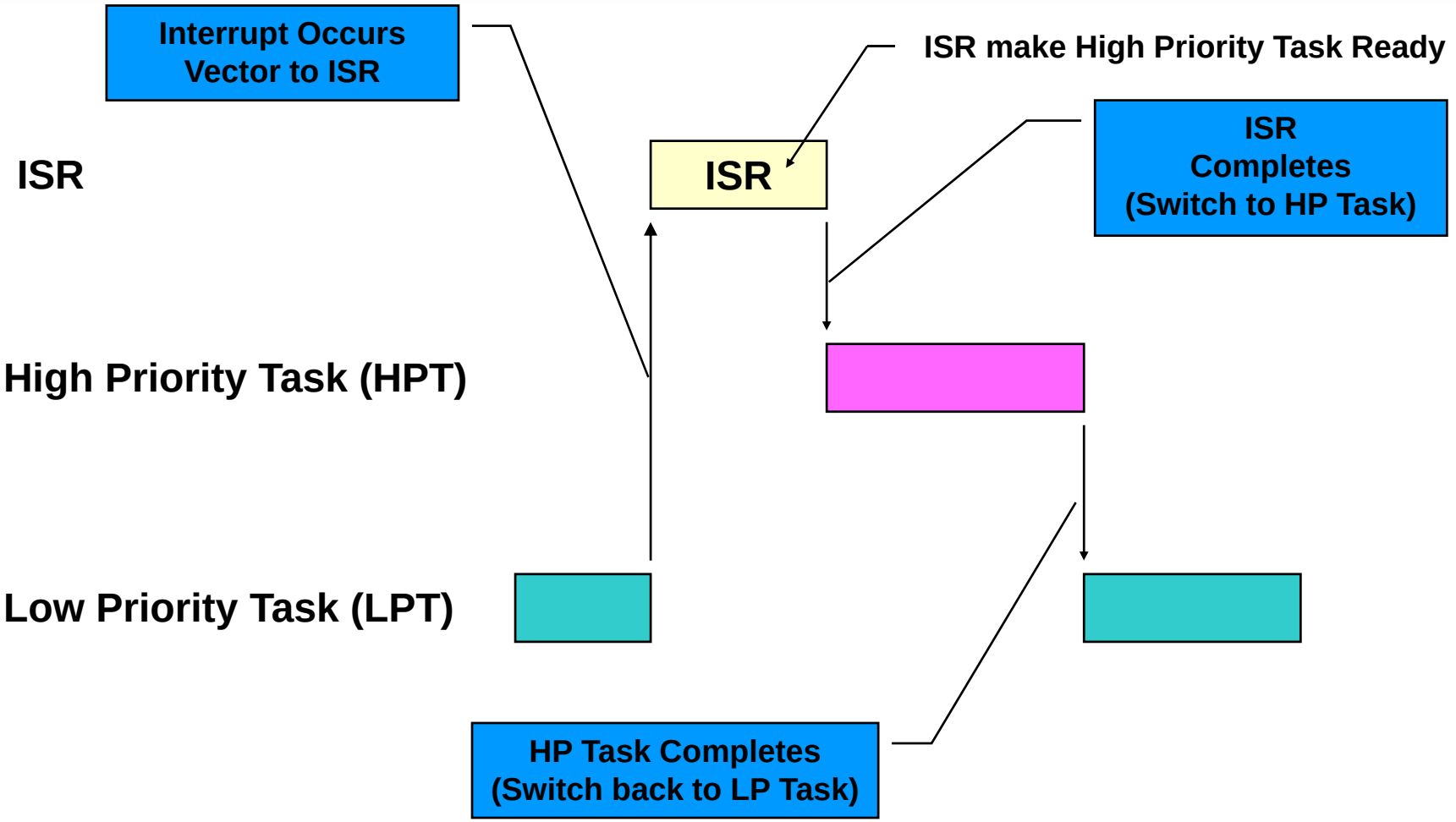
Designing with µC/OS-II
(Splitting an application into Tasks)



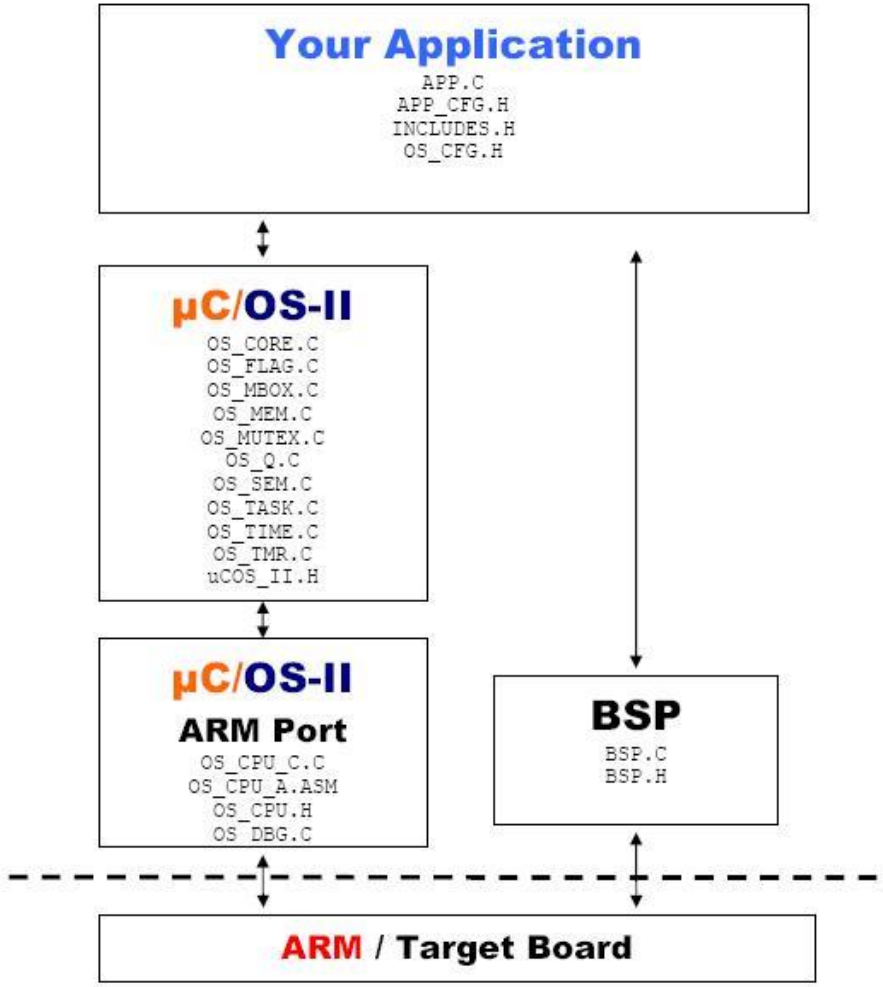
Each Task



µC/OS-II is a Preemptive Kernel



µC/OS-II and the Cortex-M3
Source Files



Part I

Foreground/Background Systems

µC/OS-II, The Real-Time Kernels

Task Management

What are Tasks?

- A task is a simple program that thinks it has the CPU all to itself.
- Each Task has:
  - Its own stack space
  - A priority based on its importance
- A task contains YOUR application code!

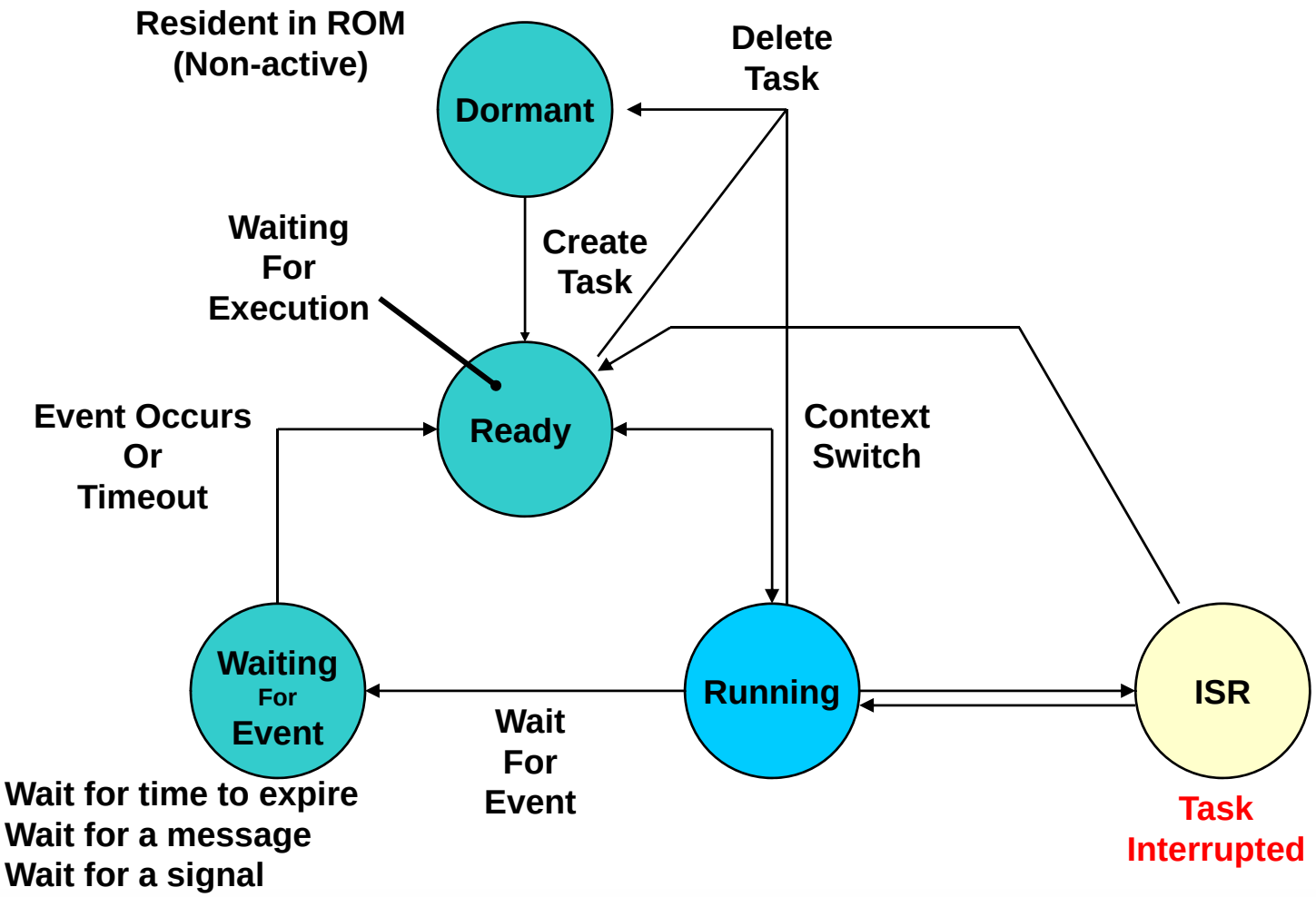
What are Tasks?

- A task is an infinite loop:

```
void Task(void *p_arg)
{
    Do something with 'argument' p_arg;
    Task initialization;
    for (;;) {
        /* Processing (Your Code)                    */
        Wait for event;     /* Time to expire ...    */
                            /* Signal from ISR ...   */
                            /* Signal from task ...  */
        /* Processing (Your Code)                    */
    }
}
```

- A task can be in one of 5 states...

Task States



Tasks needs to be ‘Created’

- To make them ready for multitasking!
- The kernel needs to have information about your task:
  - Its starting address
  - Its top-of-stack (TOS)
  - Its priority
  - Arguments passed to the task

‘Creating’ a Task

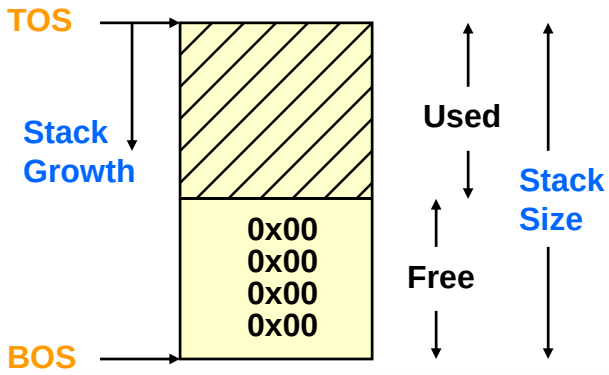
- You create a task by calling a service provided by the kernel:

```
INT8U OSTaskCreate(void (*p_task)(void *p_arg),
                   void  *p_arg,
                   void  *p_stk,
                   INT8U  prio);
```

- You can create a task:
  - before you start multitasking (at init-time) or,
  - during (at run-time).

Stack Checking

- Stacks can be checked at run-time to see if you allocated sufficient RAM
- Allows you to know the ‘worst case’ stack growth of your task(s)
- Stack is cleared when task is created
  - Optional



Deleting a Task

- Tasks can be deleted (return to the 'dormant' state) at run-time
  - Task can no longer be scheduled
- Code is NOT actually deleted
- Can be used to 'abort' (or 'kill') a task
- TCB freed and task stack could be reused.

µC/OS-II
Task APIs

```
INT8U       OSTaskChangePrio (INT8U          oldprio,
                              INT8U          newprio);

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

INT8U       OSTaskCreateExt  (void           (*task)(void *p_arg),
                              void           *p_arg,
                              OS_STK         *ptos,
                              INT8U           prio,
                              INT16U          id,
                              OS_STK         *pbos,
                              INT32U          stk_size,
                              void           *pext,
                              INT16U          opt);

INT8U       OSTaskDel        (INT8U          prio);

INT8U       OSTaskDelReq     (INT8U          prio);

INT8U       OSTaskNameGet    (INT8U          prio,
                              INT8U         *pname,
                              INT8U         *perr);

void        OSTaskNameSet    (INT8U          prio,
                              INT8U         *pname,
                              INT8U         *perr);

INT8U       OSTaskResume     (INT8U          prio);

INT8U       OSTaskSuspend    (INT8U          prio);

INT8U       OSTaskStkChk     (INT8U          prio,

INT8U       OSTaskQuery      (INT8U          prio,

INT32U      OSTaskRegGet     (INT8U          prio,
                              INT8U          id,
                              INT8U         *perr);

void        OSTaskRegSet     (INT8U          prio,
                              INT8U          id,
                              INT32U         value,
                              INT8U         *perr);
```

Part II

Task Scheduling

Context Switching

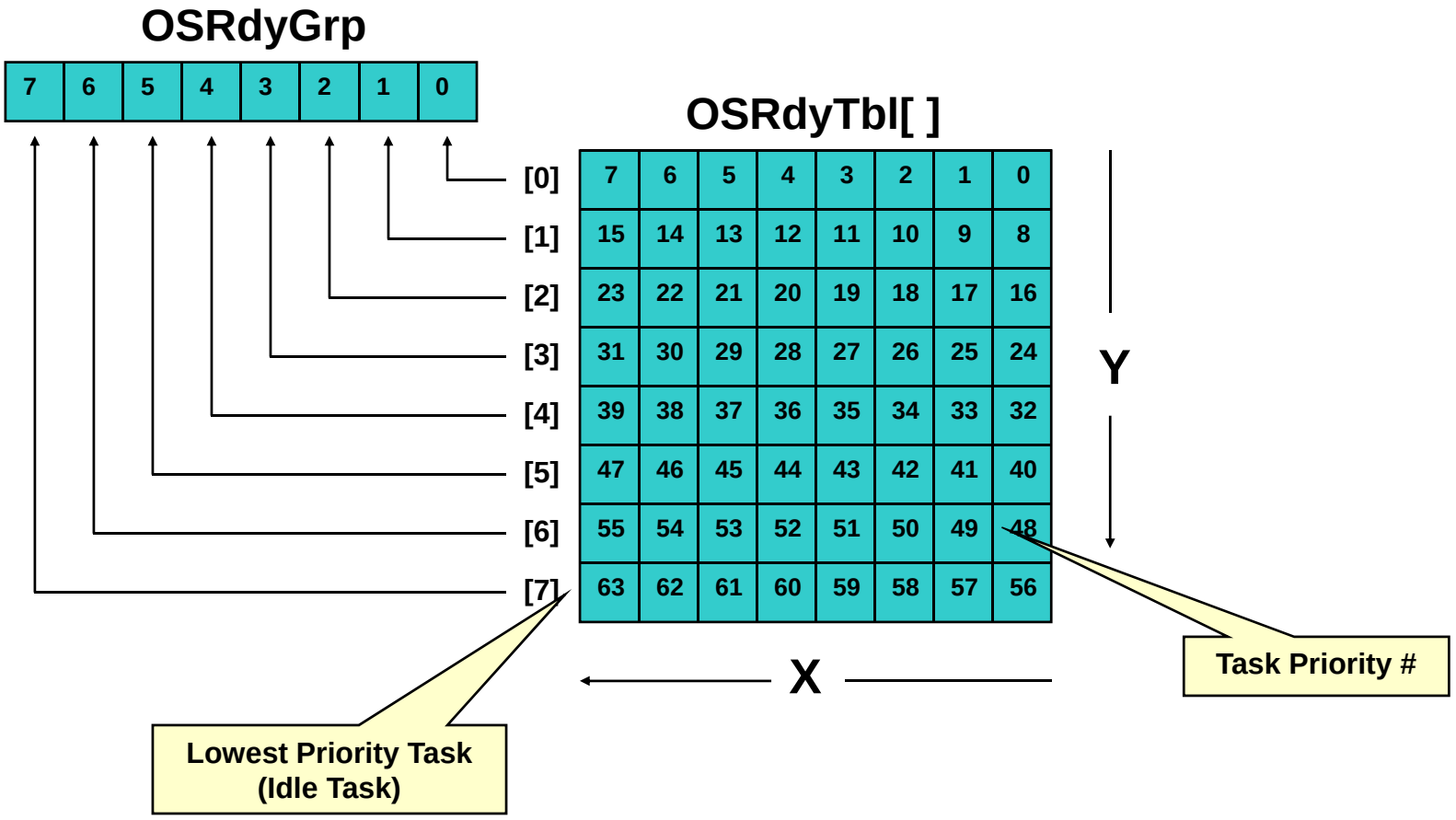
Servicing Interrupts

Time delays and Timeouts

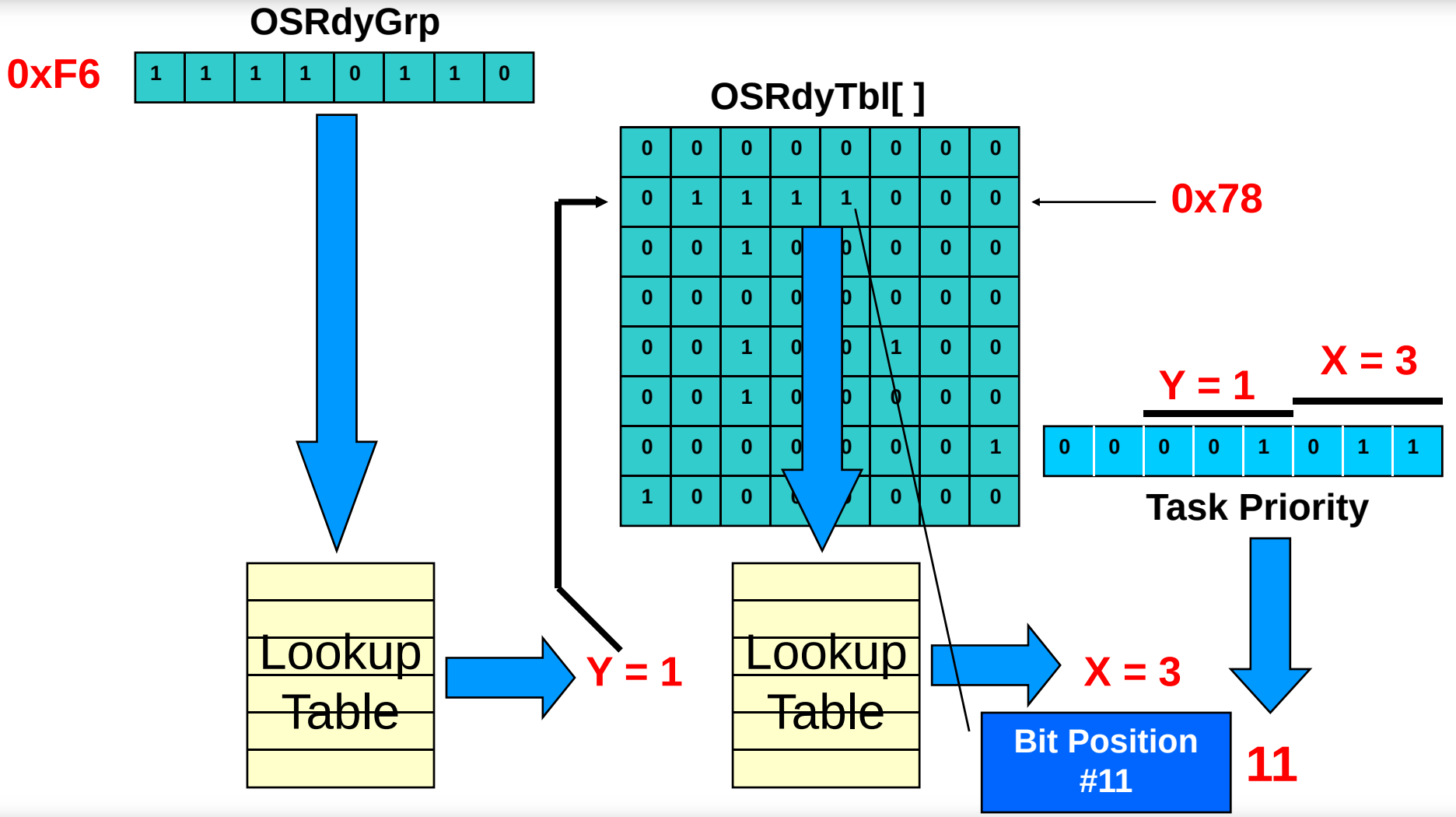
What is Scheduling?

- Deciding whether there is a more important task to run.
- Occurs:
  - When a task decides to wait for time to expire
  - When a task sends a message or a signal to another task
  - When an ISR sends a message or a signal to a task
    - Occurs at the end of all nested ISRs
- Outcome:
  - Context Switch if a more important task has been made ready-to-run or returns to the caller or the interrupted task

The µC/OS-II Ready List



Finding the Highest Priority Task Ready



Priority Resolution Table

```
/*********************************************************
*                PRIORITY RESOLUTION TABLE
*
* Note(s): 1) Index into table is bit pattern to resolve
*             highest priority.
*          2) Indexed value corresponds to highest priority
*             bit position (i.e. 0..7)
*********************************************************/
INT8U const OSUnMapTbl[] = {
   0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x00-0x0F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x10-0x1F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x20-0x2F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x30-0x3F
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x40-0x4F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x50-0x5F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x60-0x6F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x70-0x7F
   7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x80-0x8F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x90-0x9F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xA0-0xAF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xB0-0xBF
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xC0-0xCF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xD0-0xDF
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xE0-0xEF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0    // 0xF0-0xFF
};
```

(Step #2)
X = @ [0x78]
(i.e. 0x78 = OSRdyTbl[1])

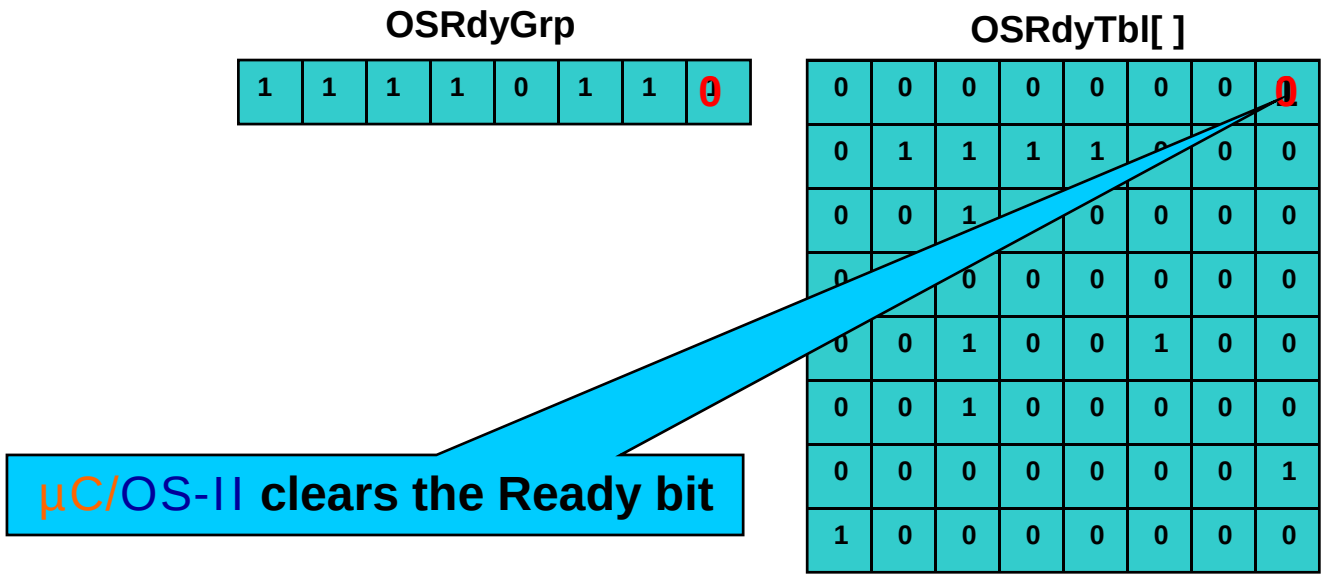
(Step #1)
Y = @ [0xF6]
(i.e. 0xF6 = OSRdyGrp)

Scheduling
(Delaying a Task)

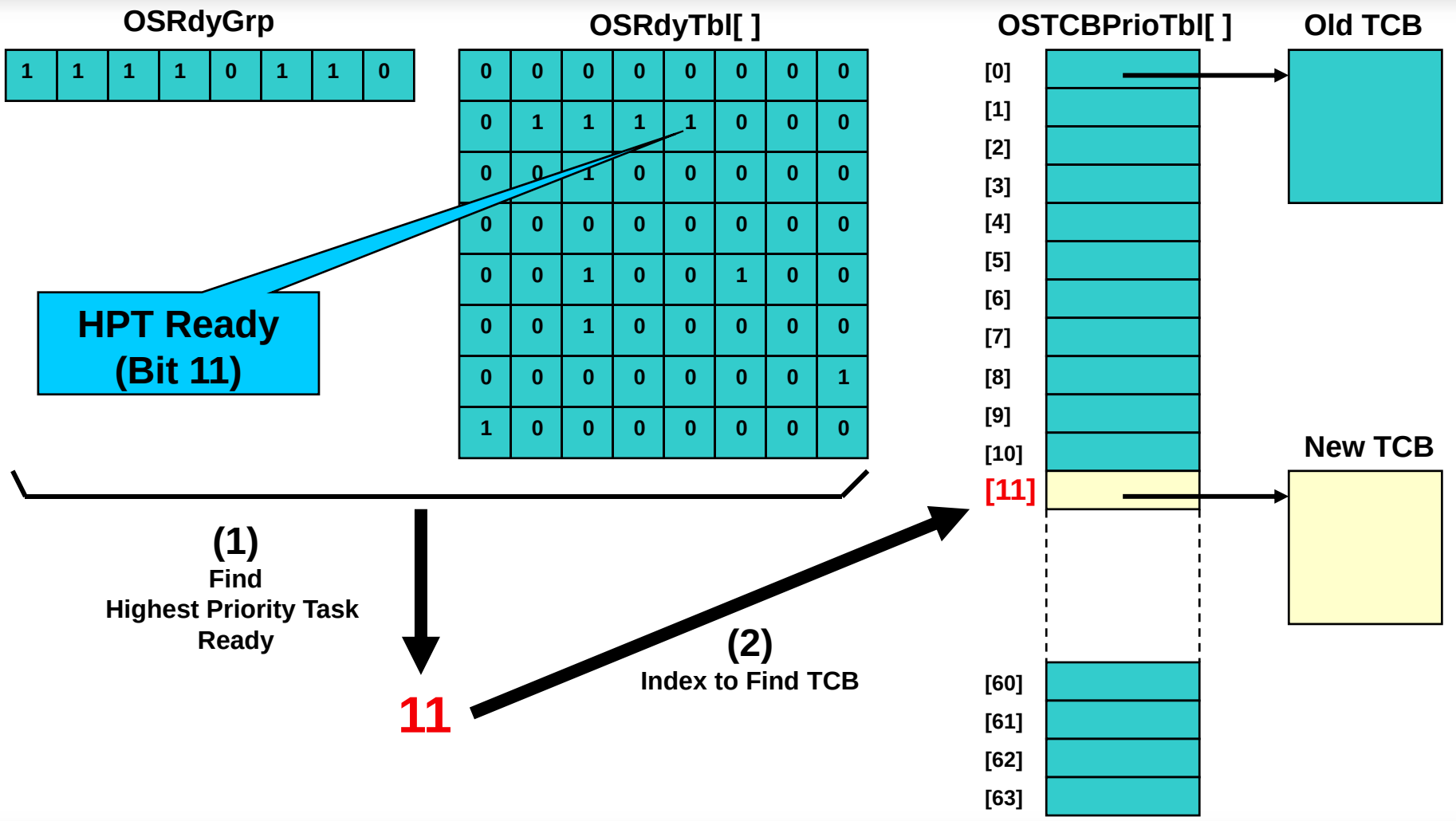
```
void TaskAtPrio0 (void *p_arg)
{
    while (TRUE) {
        .
        OSTimeDlyHMSM(0, 0, 1, 0);
        .
        .
    }
}
```

Task at Priority 0 runs

Task needs to suspend for 1 second



Scheduling



Part II

Task Scheduling

Context Switching

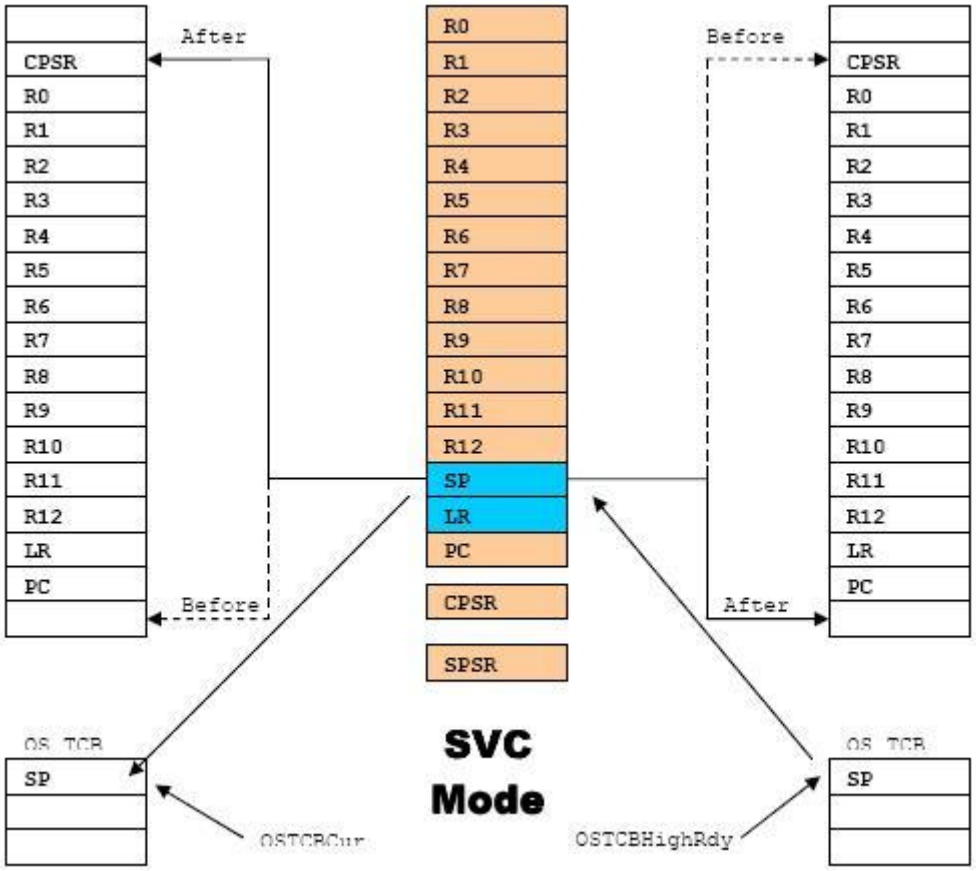
Servicing Interrupts

Time delays and Timeouts

Context Switch
(or Task Switch)

- Once the µC/OS-II finds a NEW 'High-Priority-Task', µC/OS-II performs a Context Switch.
- The context is the 'volatile' state of a CPU
  - Generally the CPU registers

Context Switch
(or Task Switch)



Part II

Task Scheduling

Context Switching

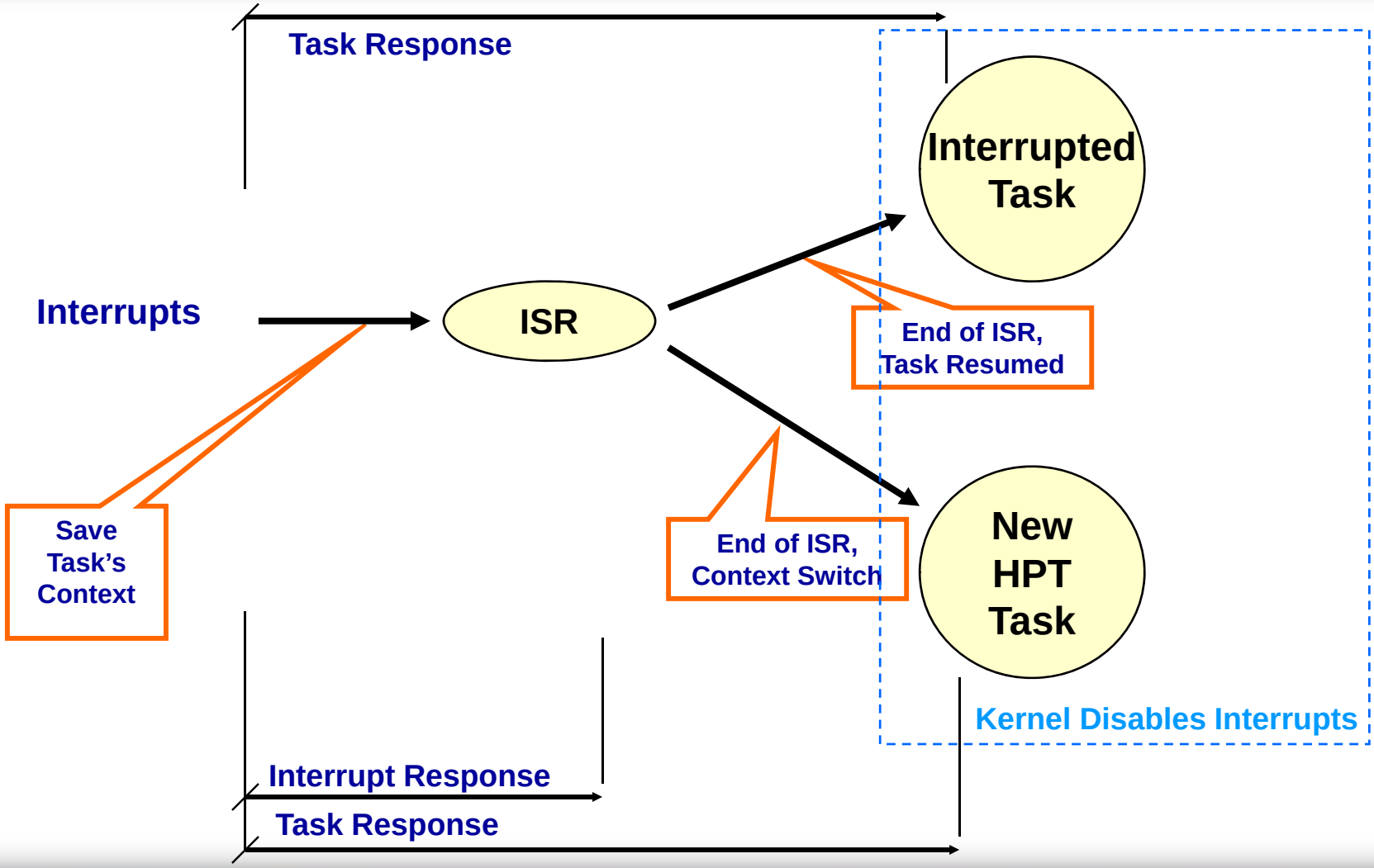
Servicing Interrupts

Time delays and Timeouts

Interrupts

- Interrupts are always more important than tasks!
- Interrupts are always recognized
  - Except when they are disabled by µC/OS-II or your application
  - Your application can disable interrupts for as much time as µC/OS-II does without affecting latency
- You should keep ISRs (Interrupt Service Routines) as short as possible.
  - Acknowledge the device
  - Signal a task to process the device

Servicing Interrupts



The Clock Tick ISR

- µC/OS-II requires a periodic interrupt source
  - Through a hardware timer
    - Between 10 and 1000 ticks/sec. (Hz)
  - Could be the power line frequency
    - 50 or 60 Hz
  - Called a ‘Clock Tick’ or ‘System Tick’
  - Higher the rate, the more the overhead!
- The tick ISR calls a service provided by µC/OS-II to signal a ‘tick’

Why keep track of Clock Ticks?

- To allow tasks to suspend execution for a certain amount of time
  - In integral number of 'ticks'
    - `OSTimeDly(ticks)`
  - In Hours, Minutes, Seconds and Milliseconds
    - `OSTimeDlyHMSM(hr, min, sec, ms)`
- To provide timeouts for other services (more on this later)
  - Avoids waiting forever for events to occur
  - Eliminates deadlocks

Part II

Task Scheduling

Context Switching

Servicing Interrupts

Time delays and Timeouts

µC/OS-II Time Delays

- µC/OS-II allows for a task to be delayed:
  - OSTimeDly(ticks)
  - OSTimeDlyHMSM(hr, min, sec, ms)
  - Always forces a context switch
  - Suspended task uses little or no CPU time
- If the tick rate is 100 Hz (10 mS), a keyboard scan every 100 mS requires 10 ticks:

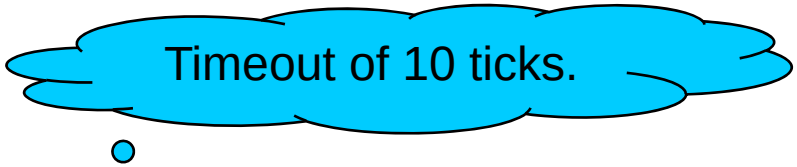
```
void Keyboard_Scan_Task (void *p_arg)
{
    for (;;) {
        OSTimeDly(10);  /* Every 100 mS */
        Scan keyboard;
    }
}
```

µC/OS-II Timeouts

- Pending on events allow for timeouts
  - To prevent waiting forever for events
- To avoid deadlocks
- Example:
  - Read ‘slow’ ADC
  - Timeout indicates that conversion didn’t occur within the expected time.

```
void ADCTask (void *p_arg)
{
    void   *p_msg;
    OS_ERR  err;

    for (;;) {
        Start ADC;
        p_msg = OSMboxPend(.., .., 10, &err);
        if (err == OS_NO_ERR) {
            Read ADC and Scale;
        } else {
            /* Problem with ADC converter! */
        }
    }
}
```



µC/OS-II
Time and Timer APIs

Time Delays

```
void        OSTimeDly          (INT16U      ticks);
INT8U       OSTimeDlyHMSM      (INT8U       hours,
                                INT8U       minutes,
                                INT8U       seconds,
                                INT16U      milli);
INT8U       OSTimeDlyResume    (INT8U       prio);
INT32U      OSTimeGet          (void);
void        OSTimeSet          (INT32U      ticks);
```

Timers

```
OS_TMR      *OSTmrCreate       (INT32U           dly,
                                INT32U           period,
                                INT8U            opt,
                                OS_TMR_CALLBACK  callback,
                                void            *callback_arg,
                                INT8U           *pname,
                                INT8U           *perr);
BOOLEAN     OSTmrDel           (OS_TMR          *ptmr,
                                INT8U           *perr);
INT8U       OSTmrNameGet       (OS_TMR          *ptmr,
                                INT8U           *pdest,
                                INT8U           *perr);
INT32U      OSTmrRemainGet     (OS_TMR          *ptmr,
                                INT8U           *perr);
INT8U       OSTmrStateGet      (OS_TMR          *ptmr,
                                INT8U           *perr);
BOOLEAN     OSTmrStart         (OS_TMR          *ptmr,
                                INT8U           *perr);
BOOLEAN     OSTmrStop          (OS_TMR          *ptmr,
                                INT8U            opt,
                                void            *callback_arg,
                                INT8U           *perr);
```

Part III

Resource Sharing and Mutual Exclusion

Task Synchronization

Task Communication

Resource Sharing

- YOU MUST ensure that access to common resources is protected!
  - µC/OS-II only gives you mechanisms
- You protect access to common resources by:
  - Disabling/Enabling interrupts
  - Lock/Unlock
  - MUTEX (Mutual Exclusion Semaphores)

Resource Sharing
(Disable and Enable Interrupts)

- When access to resource is done quickly
  - Must be less than µC/OS-II's interrupt disable time!
  - Be careful with Floating-point!
- Disable/Enable interrupts is the fastest way!

```
rpm = 60.0 / time;
CPU_CRITICAL_ENTER();
Global RPM = rpm;
CPU_CRITICAL_EXIT();
```

Resource Sharing
(Lock/Unlock the Scheduler)

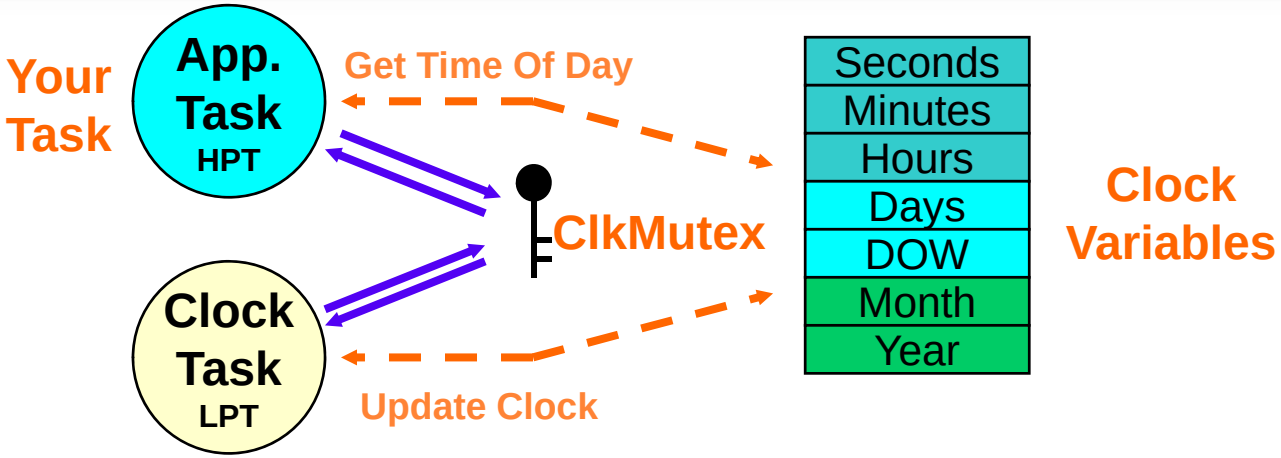
- 'Lock' prevents the scheduler from changing tasks
  - Interrupts are still enabled
  - Can be used to access non-reentrant functions
  - Can be used to reduce priority inversion
  - Same effect as making the current task the Highest Priority Task
  - Don't Lock for too long
    - Defeats the purpose of having µC/OS-II.
- 'Unlock' invokes the scheduler to see if a High-Priority Task has been made ready while locked

```
OSSchedLock();
Code with scheduler disabled;
OSSchedUnlock();
```

Mutual Exclusion
Mutexes

- Used when time to access a resource is longer than µC/OS-II's interrupt disable time!
- Mutexes are binary semaphores and are used to access a shared resource
- Mutexes reduce unbounded 'priority inversions'

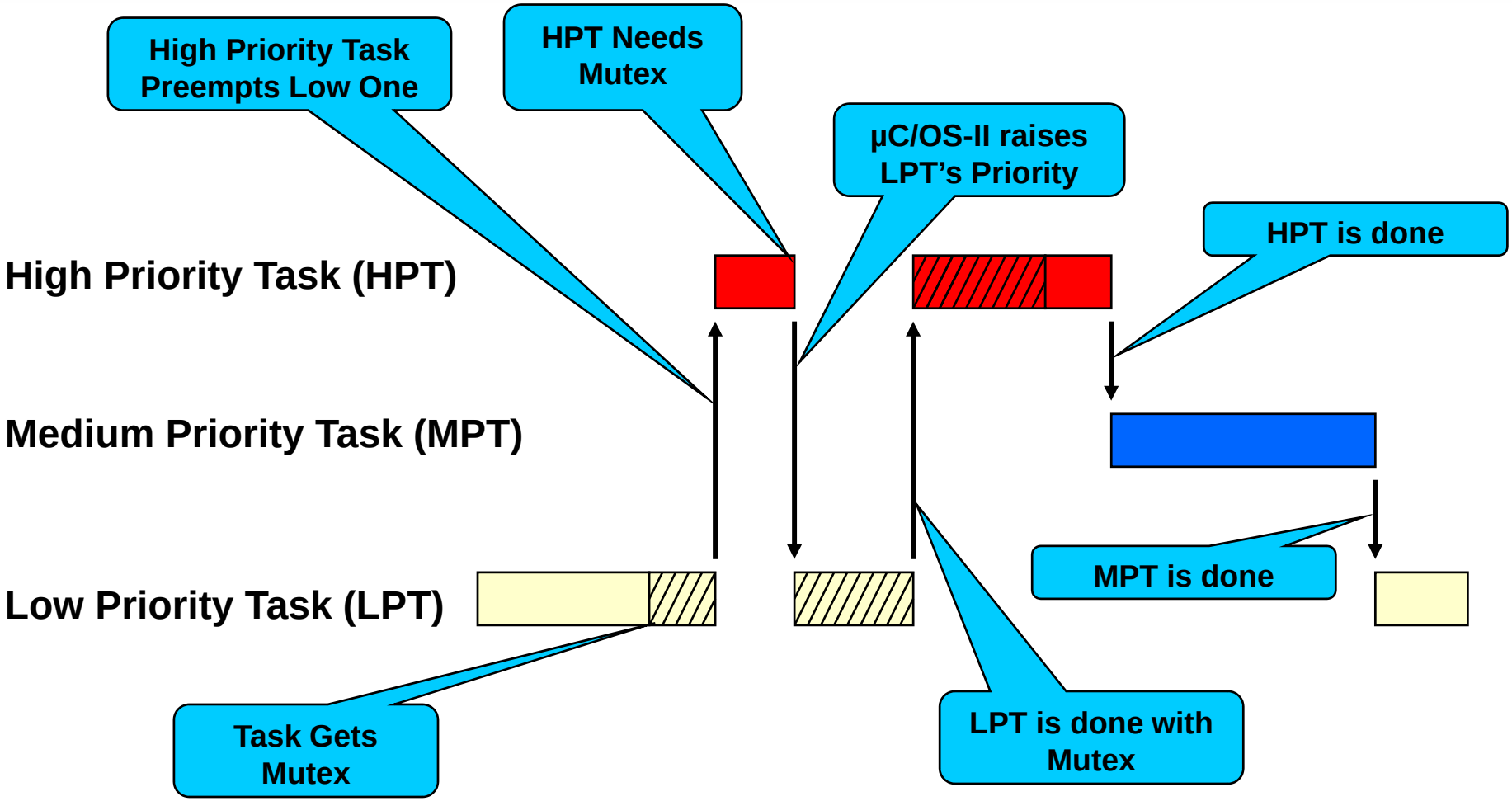
Using a Mutex
(Time-of-Day Clock)




```
ClockTask(void)
{
    while (TRUE) {
        OSTimeDlyHMSM(0, 0, 1, 0);
        OSMutexPend(&ClkMutex, 0);
        Update clock;
        OSMutexPost(&ClkMutex);
    }
}
```

```
AppTask(void)
{
    while (TRUE) {
        OSMutexPend(&ClkMutex, 0);
        Get Time Of Day;
        OSMutexPost(&ClkMutex);
        :
    }
}
```

µC/OS-II's Mutexes
Priority Ceiling



µC/OS-II
Resource Sharing APIs

Mutual Exclusion Semaphores

```
BOOLEAN        OSMutexAccept     (OS_EVENT        *pevent,
                                  INT8U            *perr);
OS_EVENT       *OSMutexCreate    (INT8U            prio,
                                  INT8U            *perr);
OS_EVENT       *OSMutexDel       (OS_EVENT        *pevent,
                                  INT8U            opt,
                                  INT8U            *perr);
void           OSMutexPend       (OS_EVENT        *pevent,
                                  INT16U           timeout,
                                  INT8U            *perr);
INT8U          OSMutexPost       (OS_EVENT        *pevent);
INT8U          OSMutexQuery      (OS_EVENT        *pevent,
                                  OS_MUTEX_DATA    *p_mutex_data);
```

Scheduler Lock/Unlock

```
void           OSSchedLock       (void);
void           OSSchedUnlock     (void);
```

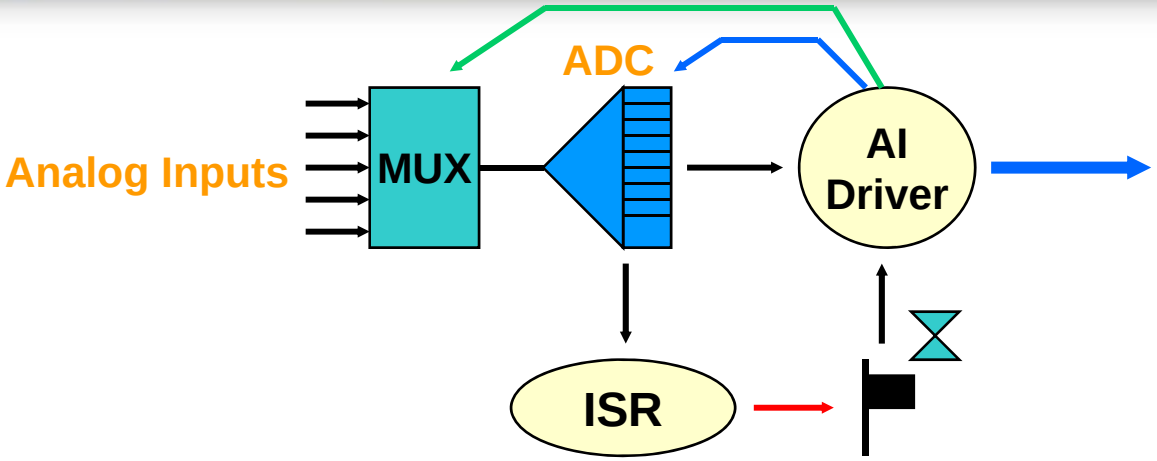
Part III

Resource Sharing and Mutual Exclusion

Task Synchronization

Task Communication

Semaphores to signal tasks
(Analog-Digital Conversion)



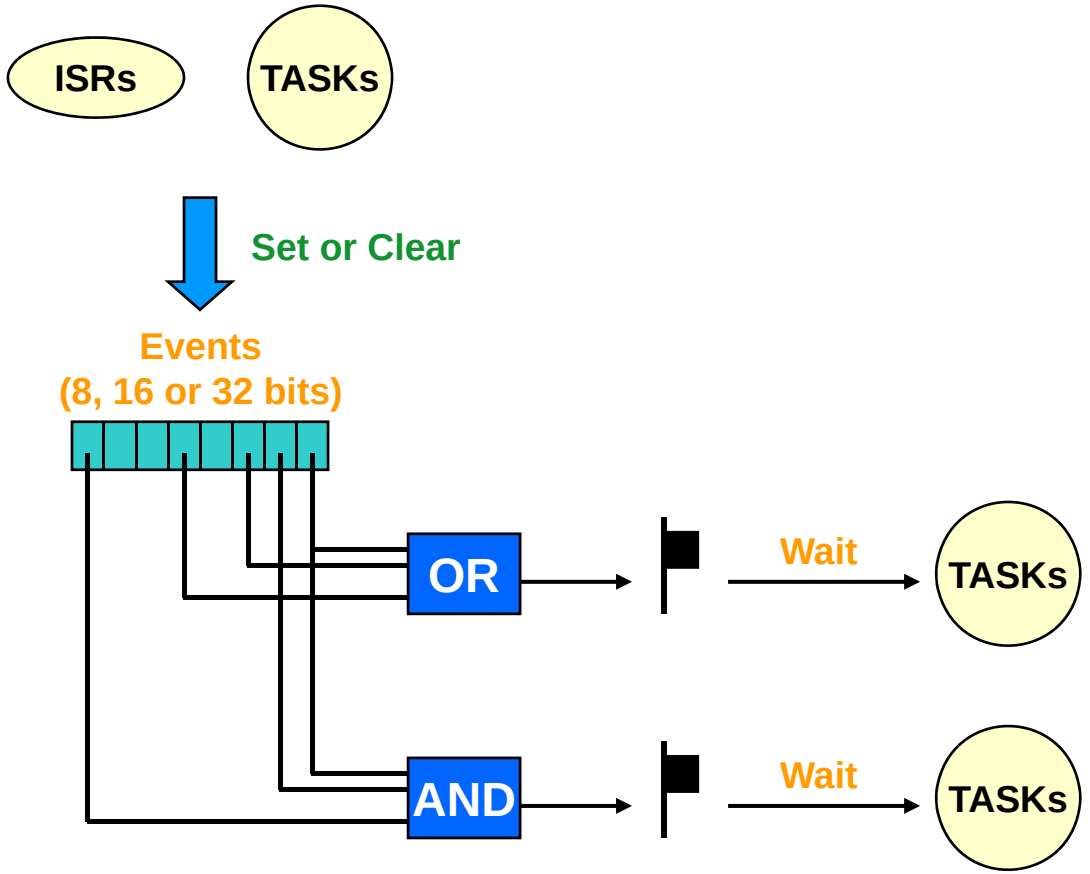

```
Read_Analog_Input_Channel_Cnts(channel#, *adc_counts)
{
    Select the desired analog input channel
    Wait for MUX output to stabilize
    Start the ADC Conversion
    Wait for signal from ADC ISR (with timeout)
    if (timed out)
        Return error code to caller
    else
        Read ADC counts
        Return ADC counts to caller
}
```

```
ADC_ISR(void)
{
    Signal Event
    Clear EOC interrupt
}
```

Event Flags

- Synchronization of tasks with the occurrence of multiple events
- Events are grouped
  - 8, 16 or 32 bits per group (compile-time configurable)
- Types of synchronization:
  - Disjunctive (OR): *Any* event occurred
  - Conjunctive (AND): *All* events occurred
- Task(s) or ISR(s) can either *Set* or *Clear* event flags
- Only tasks can *Wait* for events

Event Flags



µC/OS-II

Task Synchronization APIs

Event Flags

```
OS_FLAGS       OSFlagAccept            (OS_FLAG_GRP     *pgrp,
                                        OS_FLAGS         flags,
                                        INT8U            wait_type,
                                        INT8U            *perr);
OS_FLAG_GRP    *OSFlagCreate           (OS_FLAGS        flags,
                                       INT8U            *perr);
OS_FLAG_GRP    *OSFlagDel              (OS_FLAG_GRP     *pgrp,
                                        INT8U            opt,
                                        INT8U            *perr);
INT8U          OSFlagNameGet           (OS_FLAG_GRP     *pgrp,
                                        INT8U            *pname,
                                        INT8U            *perr);
void           OSFlagNameSet           (OS_FLAG_GRP     *pgrp,
                                        INT8U            *pname,
                                        INT8U            *perr);
OS_FLAGS       OSFlagPend              (OS_FLAG_GRP     *pgrp,
                                        OS_FLAGS         flags,
                                        INT8U            wait_type,
                                        INT16U           timeout,
                                        INT8U            *perr);
OS_FLAGS       OSFlagPendGetFlagsRdy   (void);
OS_FLAGS       OSFlagPost              (OS_FLAG_GRP     *pgrp,
                                        OS_FLAGS         flags,
                                        INT8U            opt,
                                        INT8U            *perr);
OS_FLAGS       OSFlagQuery             (OS_FLAG_GRP     *pgrp,
                                        INT8U            *perr);
```

Counting Semaphores

```
INT16U         OSSemAccept             (OS_EVENT        *pevent);
OS_EVENT       *OSSemCreate            (INT16U           cnt);
OS_EVENT       *OSSemDel               (OS_EVENT        *pevent,
                                        INT8U            opt,
                                        INT8U            *perr);
void           OSSemPend               (OS_EVENT        *pevent,
                                        INT16U           timeout,
                                        INT8U            *perr);
INT8U          OSSemPendAbort          (OS_EVENT        *pevent,
                                        INT8U            opt,
                                        INT8U            *perr);
INT8U          OSSemPost               (OS_EVENT        *pevent);
INT8U          OSSemQuery              (OS_EVENT        *pevent,
void           OSSemSet                (OS_EVENT        *pevent,
                                        INT16U           cnt,
                                        INT8U            *perr);
```

Part III

Resource Sharing and Mutual Exclusion

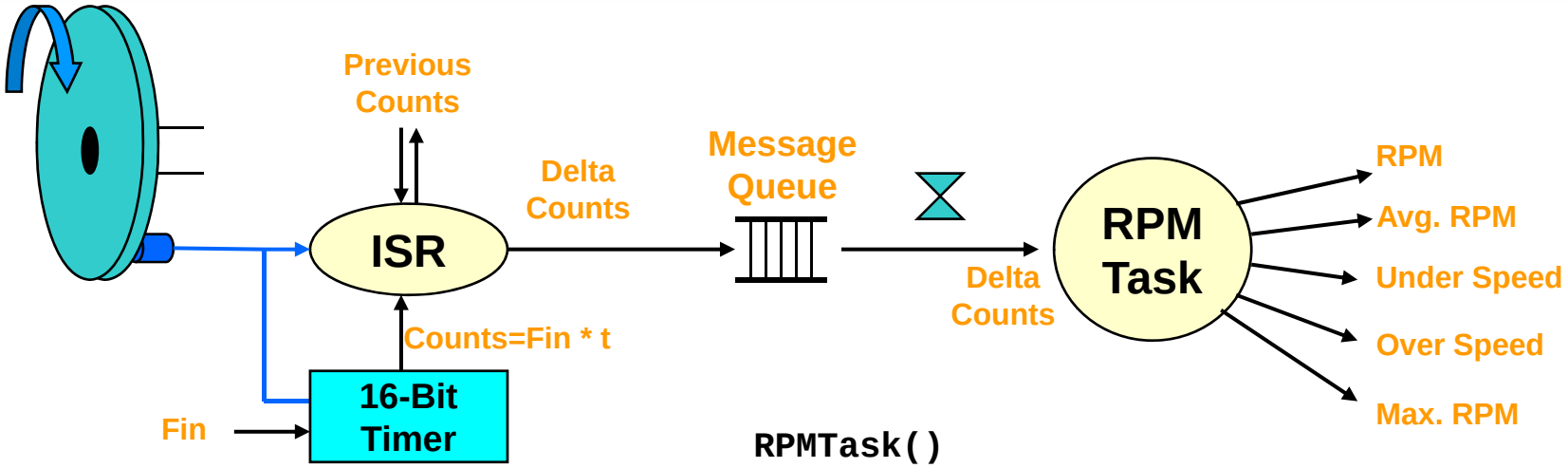
Task Synchronization

Task Communication

Message Queues

- Message passing
  - Message is a pointer
  - Pointer can point to a variable or a data structure
- FIFO *(First-In-First-Out)* type queue
  - Size of each queue can be specified to the kernel
- LIFO *(Last-In-First-Out)* also possible
- Tasks or ISR can ‘send’ messages
- Only tasks can ‘receive’ a message
  - Highest-priority task waiting on queue will get the message
- Receiving task can timeout if no message is received within a certain amount of time

Message Mailbox
(RPM Measurement)




```
RPM_ISR()
{
  Read Timer;
  DeltaCounts = Counts
              – PreviousCounts;
  PreviousCounts = Counts
  Post DeltaCounts;
}
```

```
RPMTask()
{
  while (1)
    Wait for message from ISR (with timeout);
    if (timed out)
       RPM = 0;
    else
       RPM = 60 * Fin / counts;
    Compute average RPM;
    Check for overspeed/underspeed;
    Keep track of peak RPM;
    etc.
}
```

µC/OS-II

Task Communication APIs

Message Mailboxes

```
void        *OSMboxAccept      (OS_EVENT       *pevent);
OS_EVENT    *OSMboxCreate      (void           *pmsg);
OS_EVENT    *OSMboxDel         (OS_EVENT       *pevent,
                                INT8U           opt,
                                INT8U           *perr);
void        *OSMboxPend        (OS_EVENT       *pevent,
                                INT16U          timeout,
                                INT8U           *perr);
INT8U        OSMboxPendAbort   (OS_EVENT       *pevent,
                                INT8U           opt,
                                INT8U           *perr);
INT8U        OSMboxPost        (OS_EVENT       *pevent,
                                void            *pmsg);
INT8U        OSMboxPostOpt     (OS_EVENT       *pevent,
                                void            *pmsg,
                                INT8U           opt);
INT8U        OSMboxQuery       (OS_EVENT       *pevent,
                                OS_MBOX_DATA    *p_mbox_data);
```

Message Queues

```
void        *OSQAccept         (OS_EVENT       *pevent,
                                INT8U           *perr);
OS_EVENT    *OSQCreate         (void           **start,
                                INT16U          size);
OS_EVENT    *OSQDel            (OS_EVENT       *pevent,
                                INT8U           opt,
                                INT8U           *perr);
INT8U        OSQFlush          (OS_EVENT       *pevent);
void        *OSQPend           (OS_EVENT       *pevent,
                                INT16U          timeout,
                                INT8U           *perr);
INT8U        OSQPendAbort      (OS_EVENT       *pevent,
                                INT8U           opt,
                                INT8U           *perr);
INT8U        OSQPost           (OS_EVENT       *pevent,
                                void            *pmsg);
INT8U        OSQPostFront      (OS_EVENT       *pevent,
                                void            *pmsg);
INT8U        OSQPostOpt        (OS_EVENT       *pevent,
                                void            *pmsg,
                                INT8U           opt);
INT8U        OSQQuery          (OS_EVENT       *pevent,
                                OS_Q_DATA       *p_q_data);
```

Part IV

Configuration and Initialization

Debugging with Kernels

µC/OS-II Configuration

- Allows you to specify which services are available
  - Done through #defines in application specific file: **OS_CFG.H**
- Memory footprint depends on configuration
  - On ARM, 6K to 24K of code space
  - RAM depends on kernel objects used

Initialization

```
void main (void)
{
    /* User initialization                    */

    OSInit();    /* Kernel Initialization */

    /* Install interrupt vectors              */

    /* Create at least 1 task (Start Task) */
    /* Additional User code                   */

    OSStart();   /* Start multitasking    */
}
```

Initialization

- µC/OS-II creates 1 to 3 internal tasks:
  - OS_TaskIdle()
    - Runs when no other task runs
    - Always the lowest priority task
    - Cannot be deleted
  - OS_TaskStat()
    - Computes run-time statistics
      - CPU Usage
      - Check stack usage of other tasks
  - OS_TmrTask()
    - If you enabled the ‘timer’ services
- Initializes other data structures

Part IV

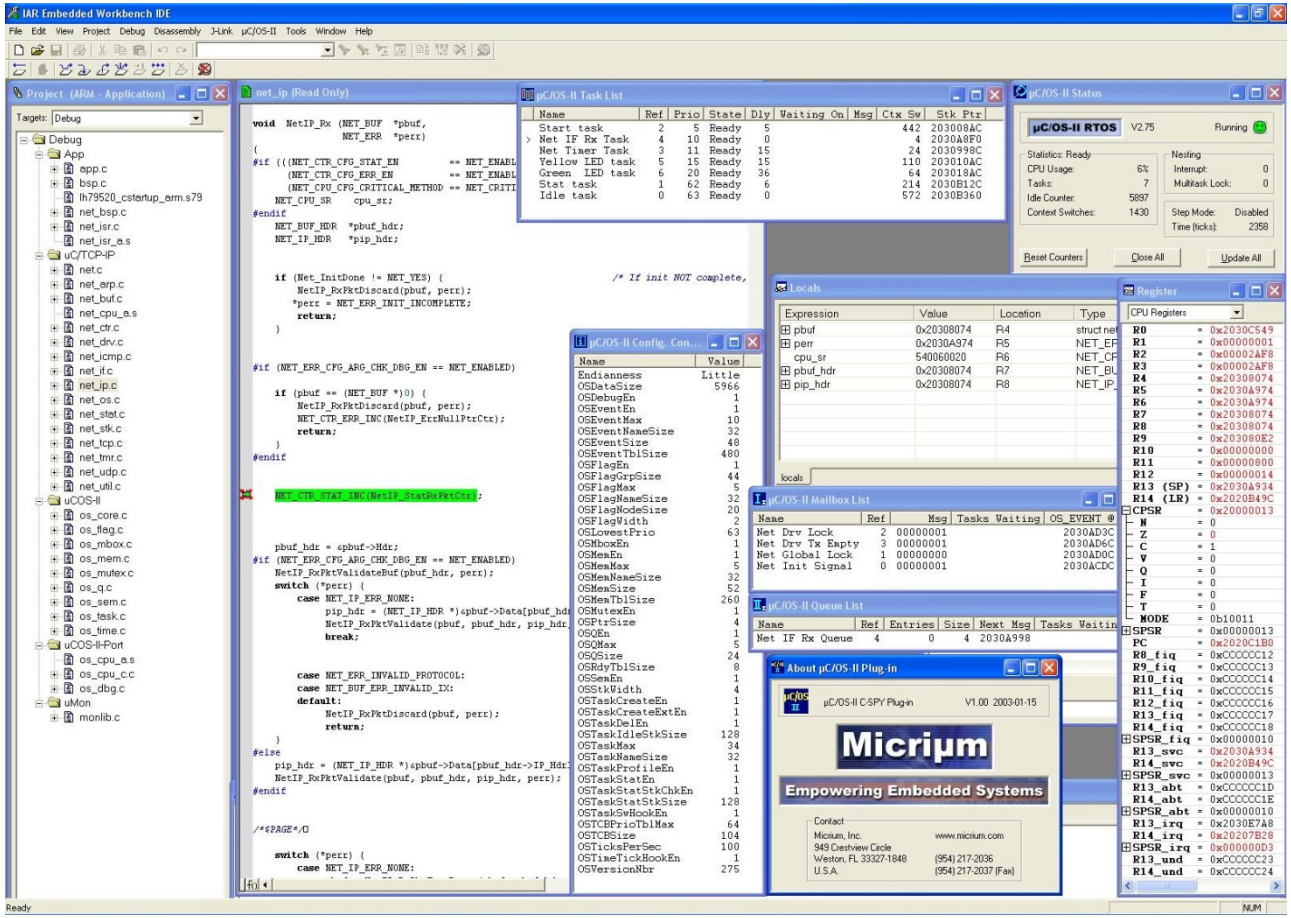
Configuration and Initialization

Debugging with Kernels

Debugging

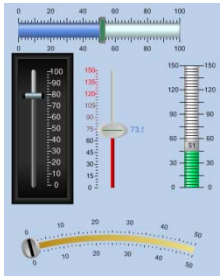
IAR's µC/OS-II Kernel Awareness

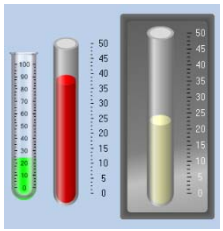
- Free with IAR's EWARM
- Static Tool
- Shows value of kernel structures:
  - Task list
  - Kernel objects

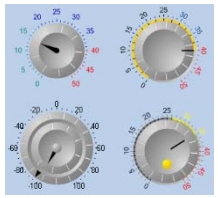


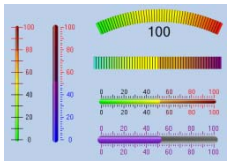
Debugging

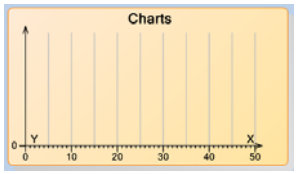
µC/Probe, Run-Time Data Monitor

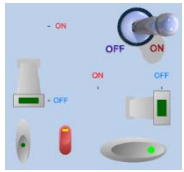


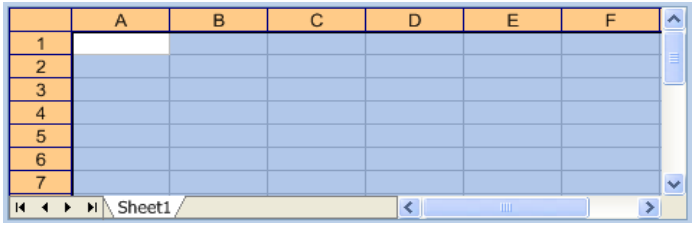


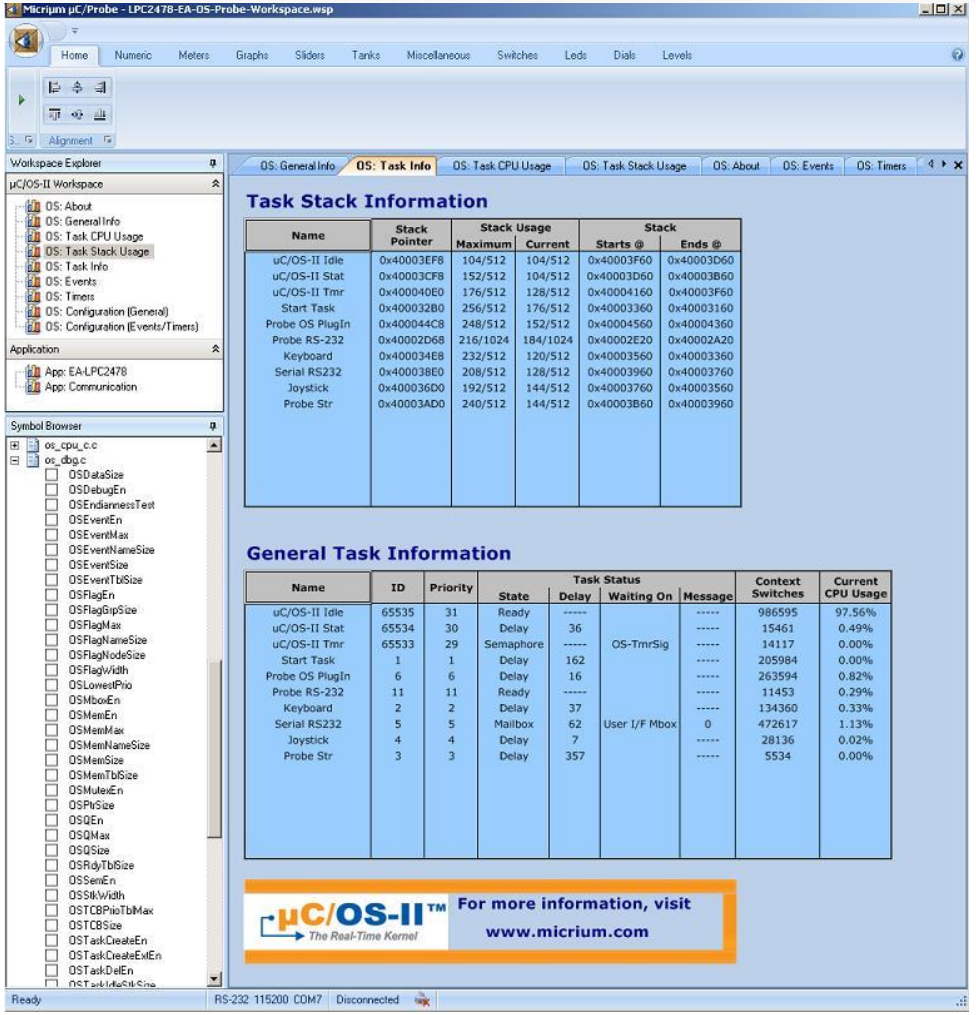












Debugging
Other Techniques

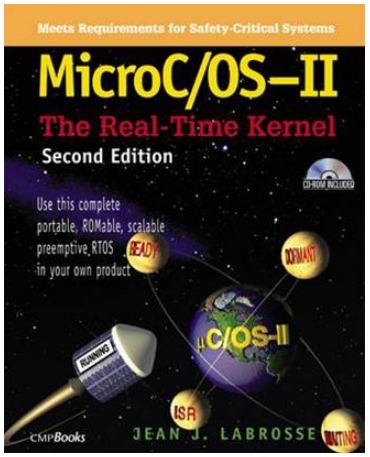
- Use a DAC (Digital to Analog Converter)
  - Output a value based on which task or ISR is running
    - Shows execution profile of each task/ISR running (oscilloscope)
    - Can be used to measure task execution time
- Use output ports for time measurements
- Use TRACE tools
  - Some processors allow you to capture execution traces
  - Some debugger captures and display run-time history

References

"A Practitioner's Handbook for Real-Time Analysis: Guide to RMA for Real-Time Systems"

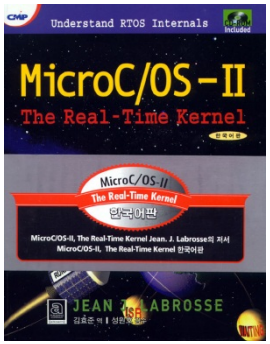
Mark H. Klein, Kluwer Academic Publishers

ISBN 0-7923-9361-9



Chinese

Korean



"µC/OS-II, The Real-Time Kernel, 2nd Edition"

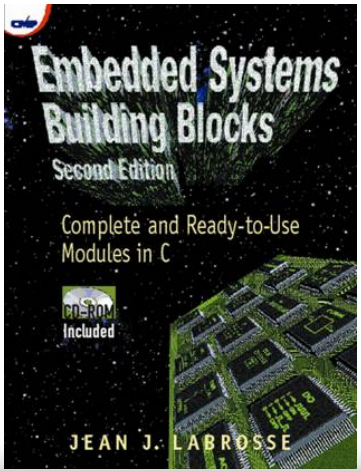
Jean J. Labrosse, CMP Books

ISBN 1-57820-103-9

"Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C"

Jean J. Labrosse, CMP Books

ISBN 0-97930-604-1



Chinese

Korean