Nios II

uC/OS-II porting with Nios II

µC/OS-II

The Real-Time Kernel

µC/OS-II
Main Features

- Portable (Most 8, 16, 32 and 64 bit CPUs)
- ROMable
- Scalable
- Preemptive
- Real-Time
  - Deterministic
  - High Performance
- Multitasking
- Robust
- Provides many services

µC/OS-II
ROMable and Scalable

- Designed for Embedded Systems
- Footprint depends on your needs:
  - Semaphores, Mutex, Event Flags, Mailboxes, Queues …
  - ROM (Code space) – NIOS-II:
    - 5 Kbytes (Min.)
    - 20 Kbytes (Max.)
  - RAM (Data space) – NIOS-II:
    - 1 Kbytes (Min.), plus task stacks
    - 5 Kbytes (Max.), plus task stacks

µC/OS-II

Services

- Semaphores
- Mutual Exclusion Semaphores
  - Reduces Priority Inversions
- Event Flags
- Message Mailboxes
- Message Queues
- Memory Management
- Time Management
- Task Management

µC/OS-II
Used in 100s of Commercial Products

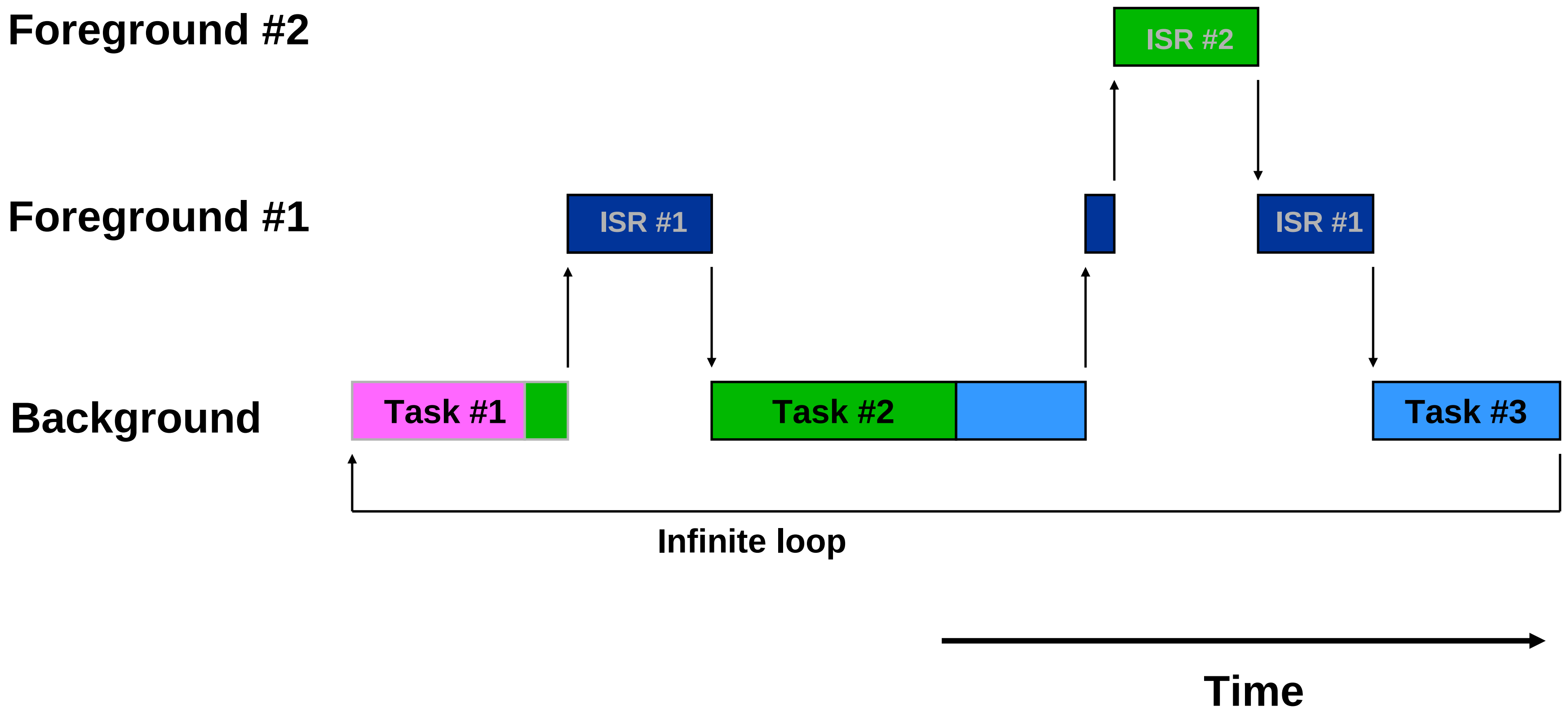
- Avionics
- Medical
- Cell phones
- Routers and switches
- High-end audio equipment
- Washing machines and dryers
- UPS (Uninterruptible Power Supplies)
- Industrial controllers
- GPS Navigation Systems
- Microwave Radios
- Instrumentation
- Point-of-sale terminals
- Many, many more

SOPC WORLD 2004

µC/OS-II
The Real-Time Kernel

Foreground/Background Systems

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...
  }
}
```

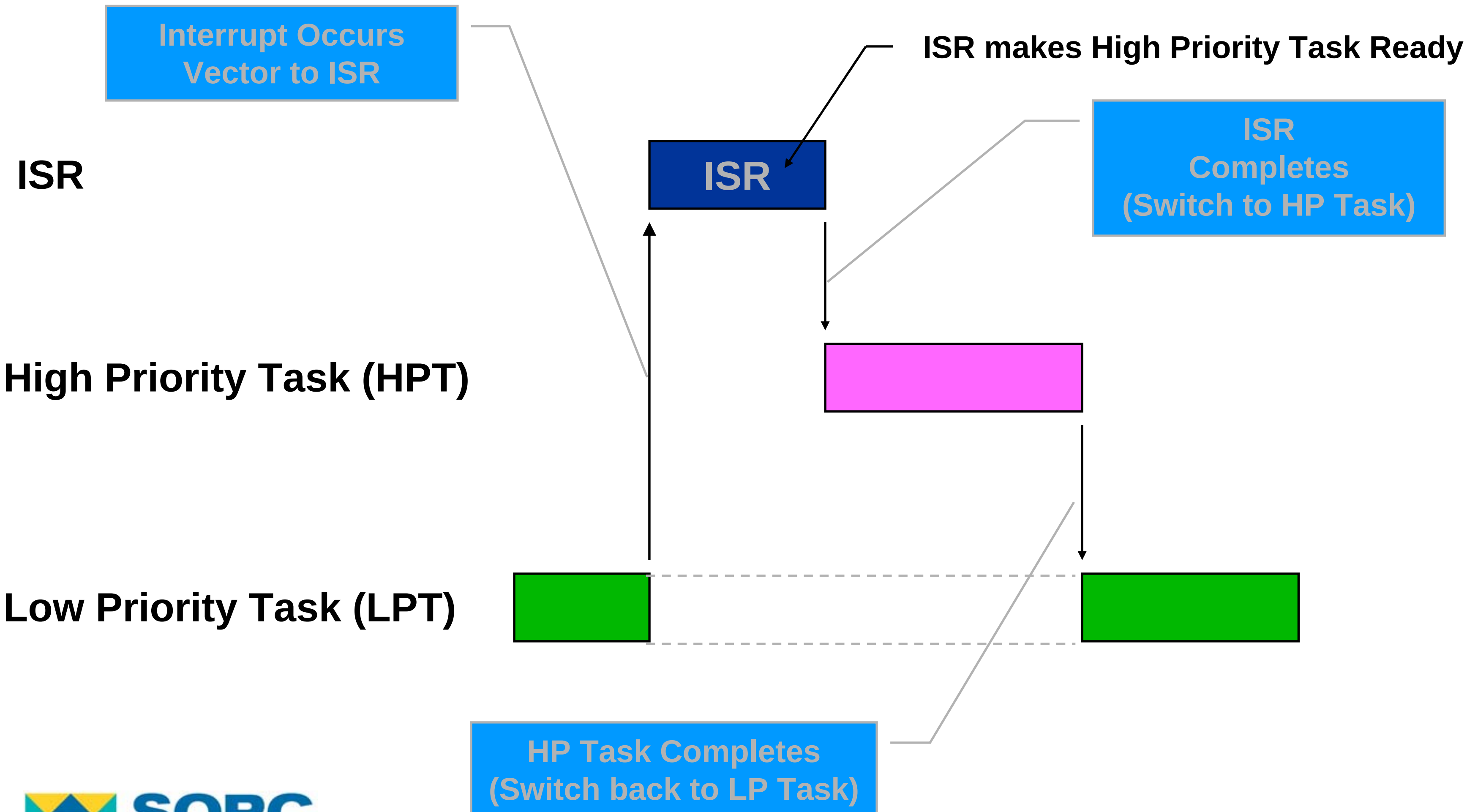
```
/* Foreground */
ISR (void)
{
    Handle asynchronous event;
}
```

Real-Time Kernels and µC/OS-II

What is a Real-Time Kernel?

- 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
  - Semaphore management
  - Intertask communication and synchronization
  - More

µC/OS-II is a Preemptive Kernel



What is a Task?

- 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

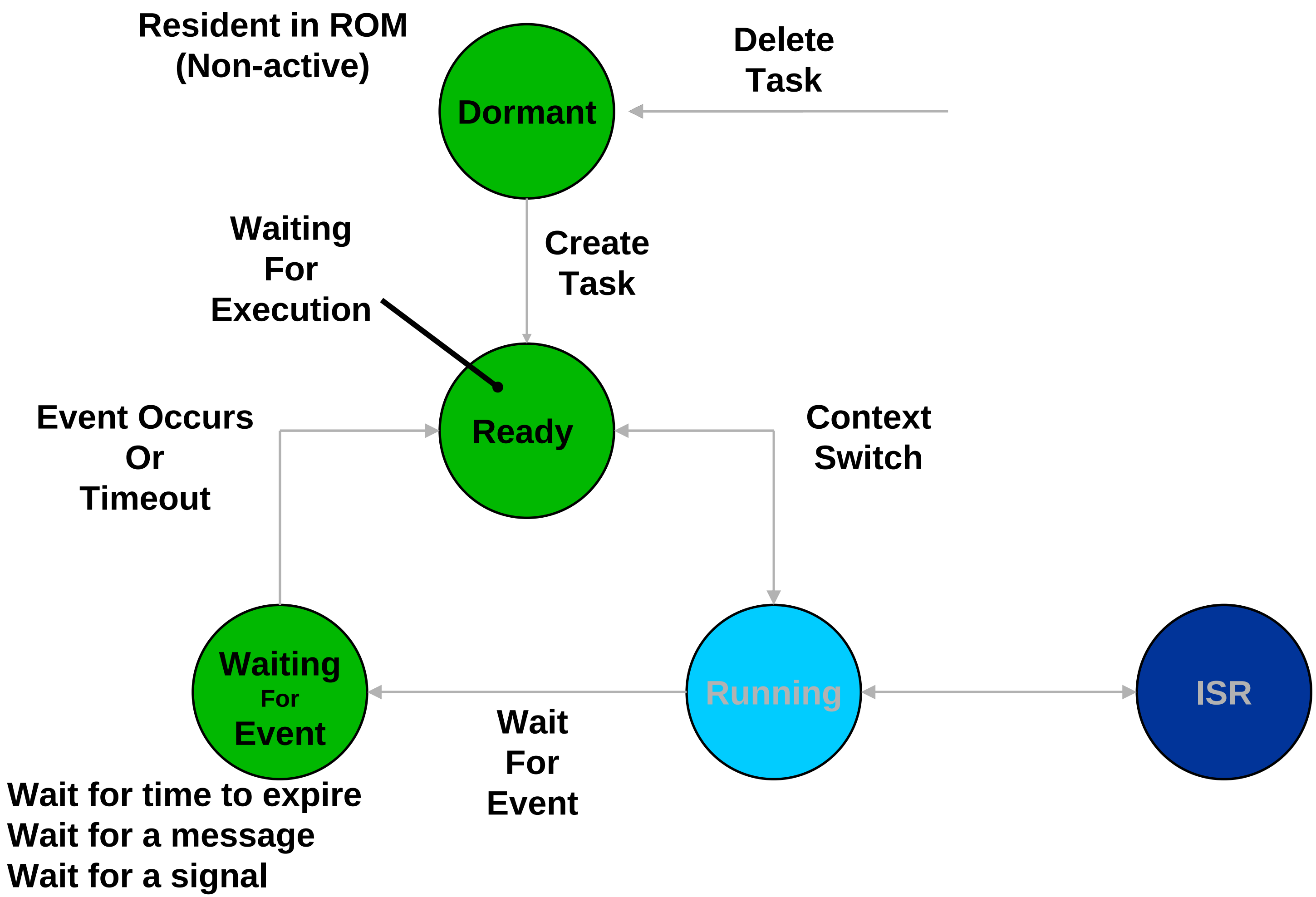
- A task contains YOUR application code!

What is a Task?

- 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)                      */
    }
}
```

Task States


Resident in ROM
(Non-active)
Dormant
Delete
Task
Waiting
For
Execution
Create
Task
Event Occurs
Or
Timeout
Ready
Context
Switch
Waiting
For
Event
Running
ISR
Wait
For
Event
Wait for time to expire
Wait for a message
Wait for a signal

‘Creating’ a Task

- µC/OS-II needs to have information about your task:
  - Its starting address
  - Its top-of-stack (TOS)
  - Its priority
  - Arguments passed to the task
  - Other

- You create a task by calling a service provided by µC/OS-II – OSTaskCreateExt()

Creating a Task

Stack … Task Create … Task Code

```
#define   APP_TASK_ID                10
#define   APP_TASK_PRIO              10
#define   APP_TASK_STK_SIZE         256

static    OS_STK    AppTaskStk[APP_TASK_STK_SIZE];
```

Task Stack

Create a Task

```
OSTaskCreateExt(AppTask,                                    // Task address
                (void *)0,                                  // 'p_arg'
                &AppTaskStk[APP_TASK_START_STK_SIZE - 1],   // Top-Of-Stack
                APP_TASK_PRIO,                              // Task priority
                APP_TASK_ID,                                // Task ID (not used)
                &AppTaskStk[0],                             // Bottom-Of-Stack
                APP_TASK_STK_SIZE,                          // Stack size
                (void *)0,                                  // 'p_ext'
                0x0000);                                    // Options
OSTaskNameSet(APP_TASK_PRIO, "App Task", &err);
```

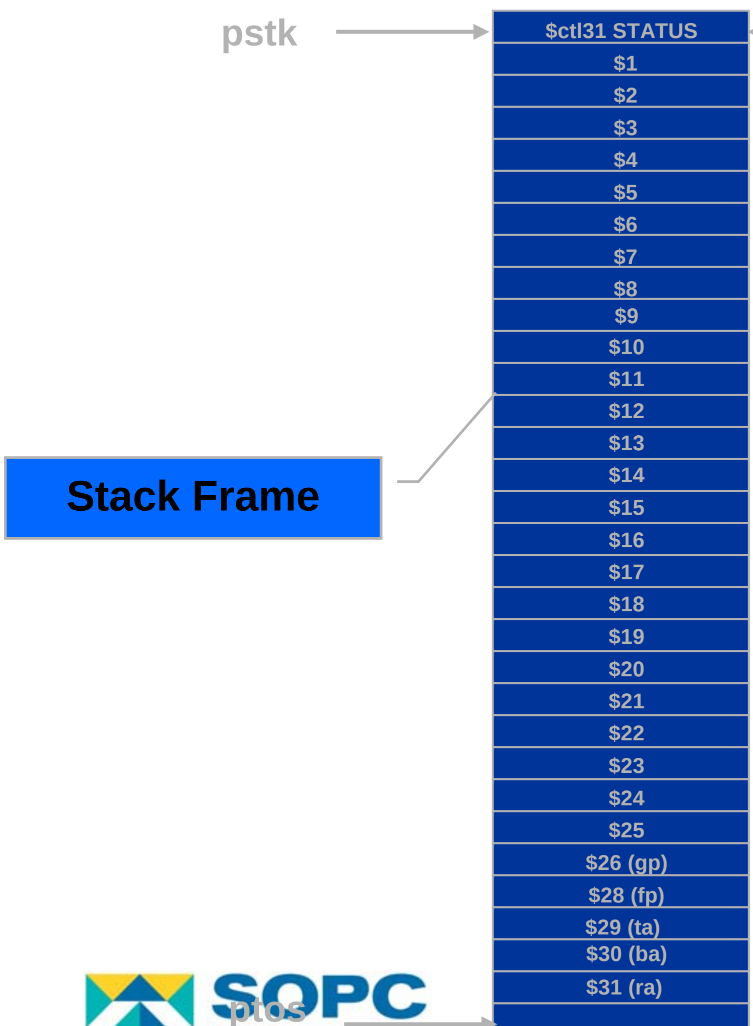
Assigning a Name to a Task

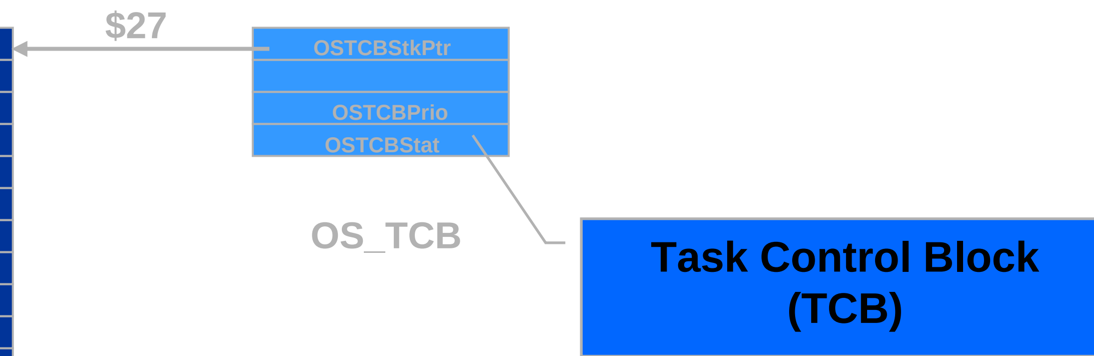
```
static  void  AppTask (void *p_arg)
{
    while (1) {
        OSTimeDly(5);
    }
}
```

Task (Infinite Loop)

Suspend for 5 ticks

Creating a Task for NIOS-II






```
OSTaskCreateExt(void (*task)(void *parg),
                void   *parg,
                void   *ptos,
                INT8U   prio
                INT16U id,
                void   *pbos,
                INT32U stk_size,
                void   *p_ext,
                INT16U opt);
```

HIGH Memory

Task Control Blocks (TCBs)

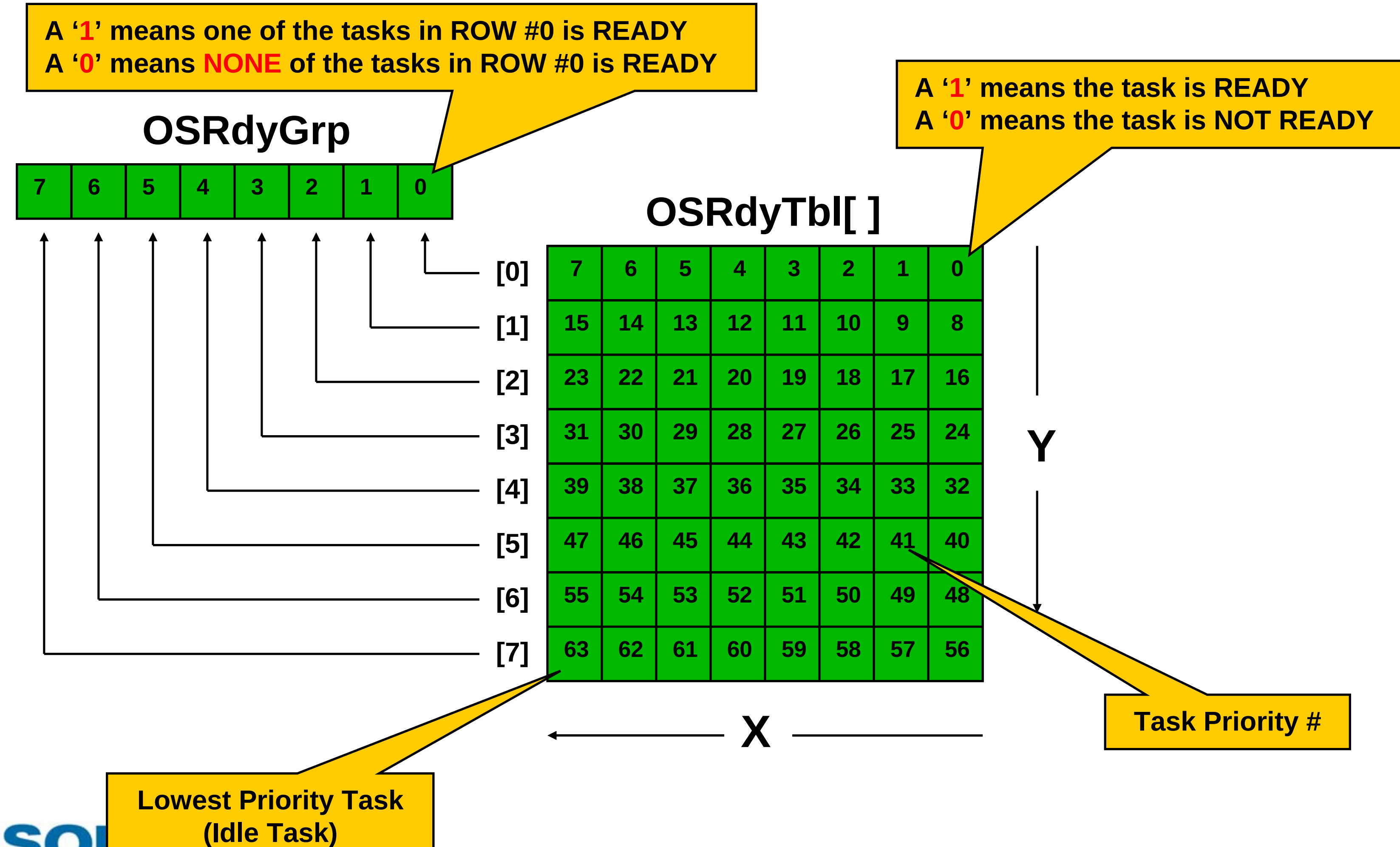
- A TCB is a data structure that is used by the kernel for task management.
- Each task is assigned a TCB when it is 'created'.
- A TCB contains:
  - The task's priority
  - The task's state (Ready, Waiting ...)
  - A pointer to the task's Top-Of-Stack (TOS)
  - Other task related data
- TCBs reside in RAM

Scheduling and Context Switching

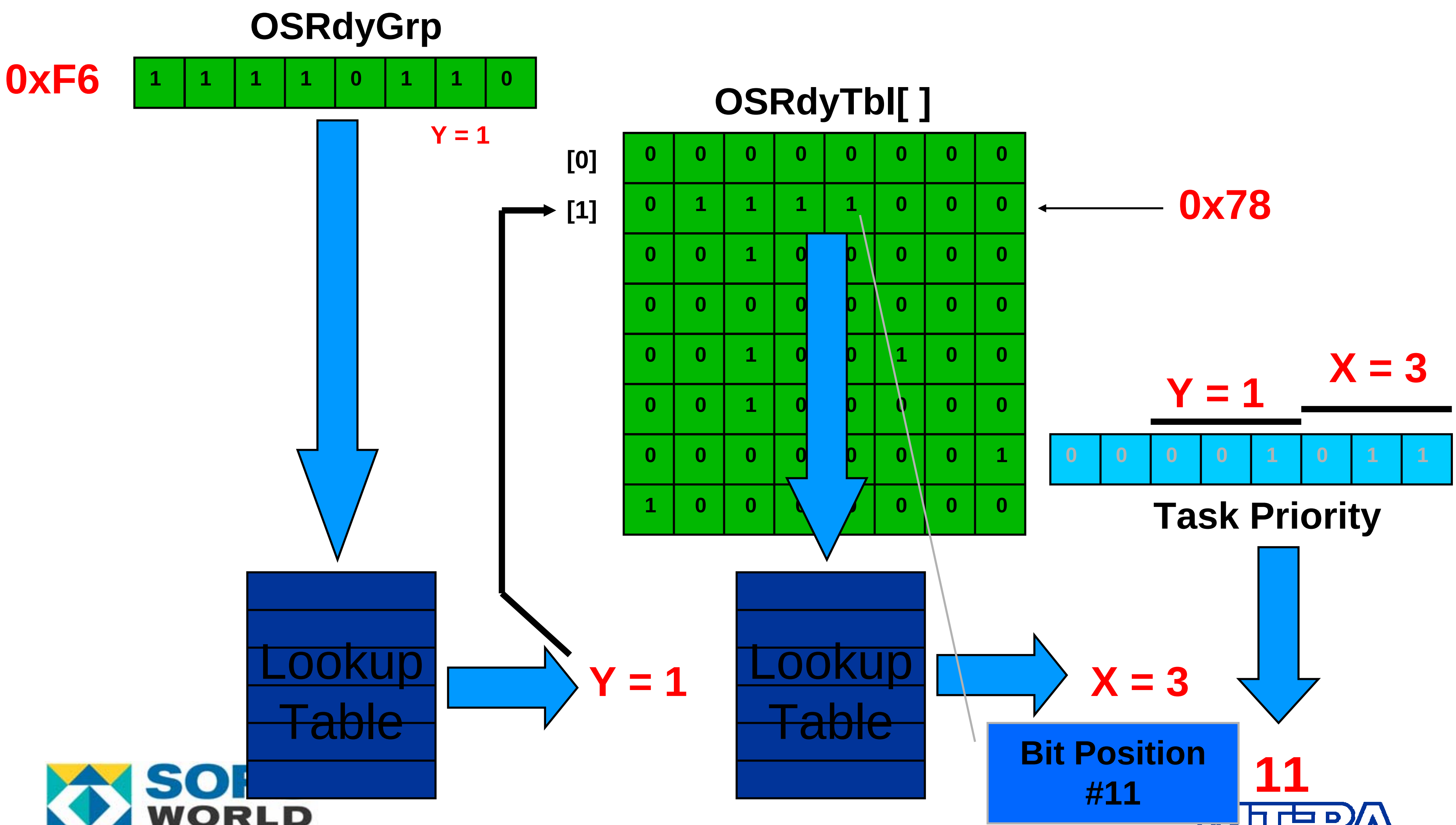
What is Scheduling?

- 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)

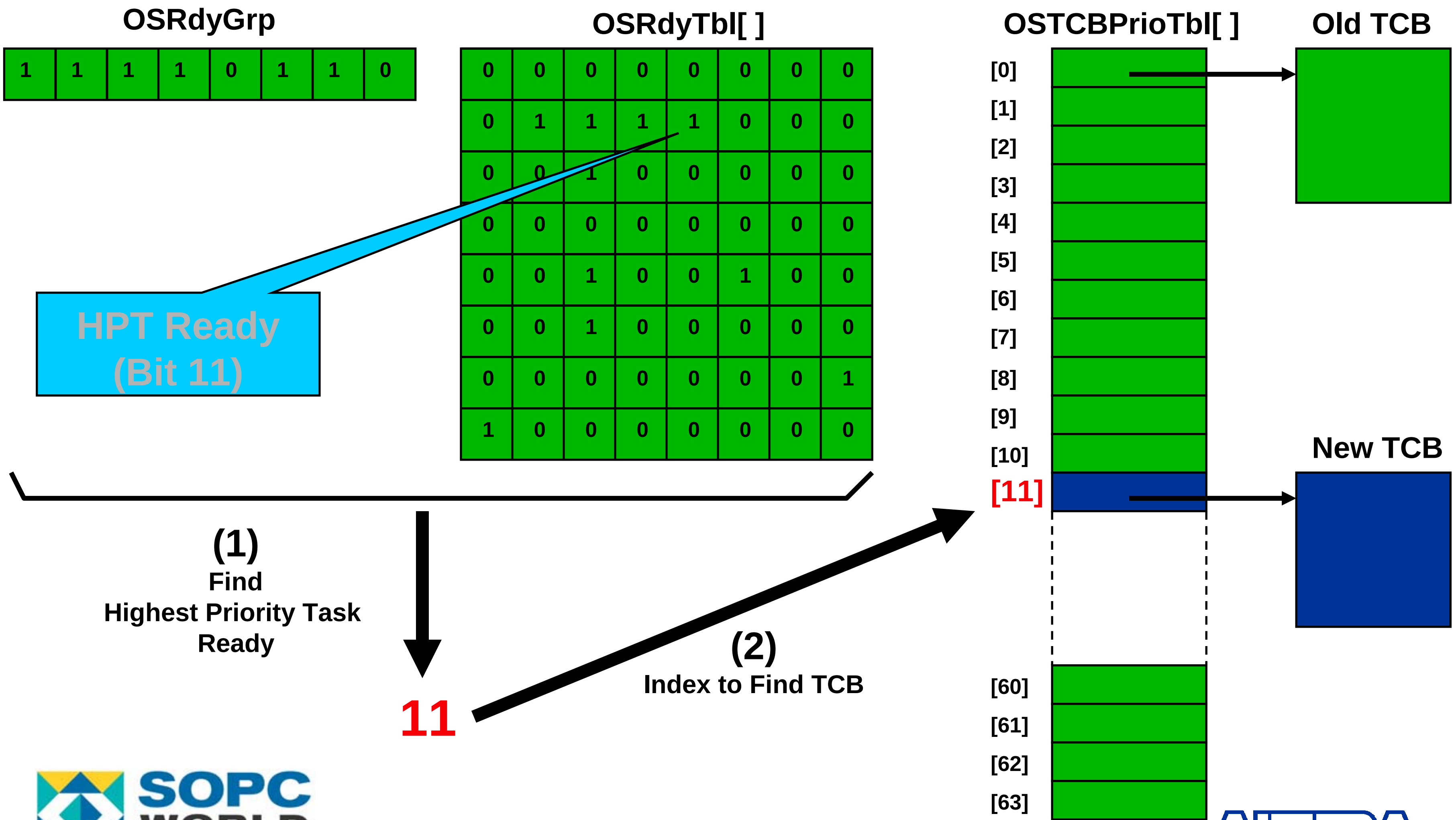
Priority Resolution

```
Y               = OSUnMapTbl[OSRdyGrp];
X               = OSUnMapTbl[OSRdyTbl[Y]];
HighestPriority = (Y * 8) + X;
```

```
Y (i.e. 1)      = OSUnMapTbl[0xF6];
X (i.e. 3)      = OSUnMapTbl[0x78];
HighestPriority = (1 * 8) + 3;
```

```
HighestPriority = 11
```

Scheduling


OSRdyGrp
1 1 1 1 0 1 1 0
OSRdyTbl[ ]
0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
HPT Ready
(Bit 11)
(1)
Find
Highest Priority Task
Ready
11
(2)
Index to Find TCB
OSTCBPrioTbl[ ]
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[60]
[61]
[62]
[63]
Old TCB
New TCB

Context Switch
(or Task Switch)

- Once the kernel finds a NEW 'High-Priority-Task', the kernel performs a Context Switch.

- The context is the 'volatile' state of a CPU
  - The NIOS-II CPU registers

- A context switch consist of:
  - Saving the current CPU registers onto the CURRENT task's stack
  - Restoring the CPU registers from the NEW task's stack

Interrupts

Interrupts

- Interrupts are always more important than tasks!

- Interrupts are always recognized
  – Except when they are disabled by µC/OS-II or the application

- You should keep ISRs (Interrupt Service Routines) as short as possible.

Interrupts

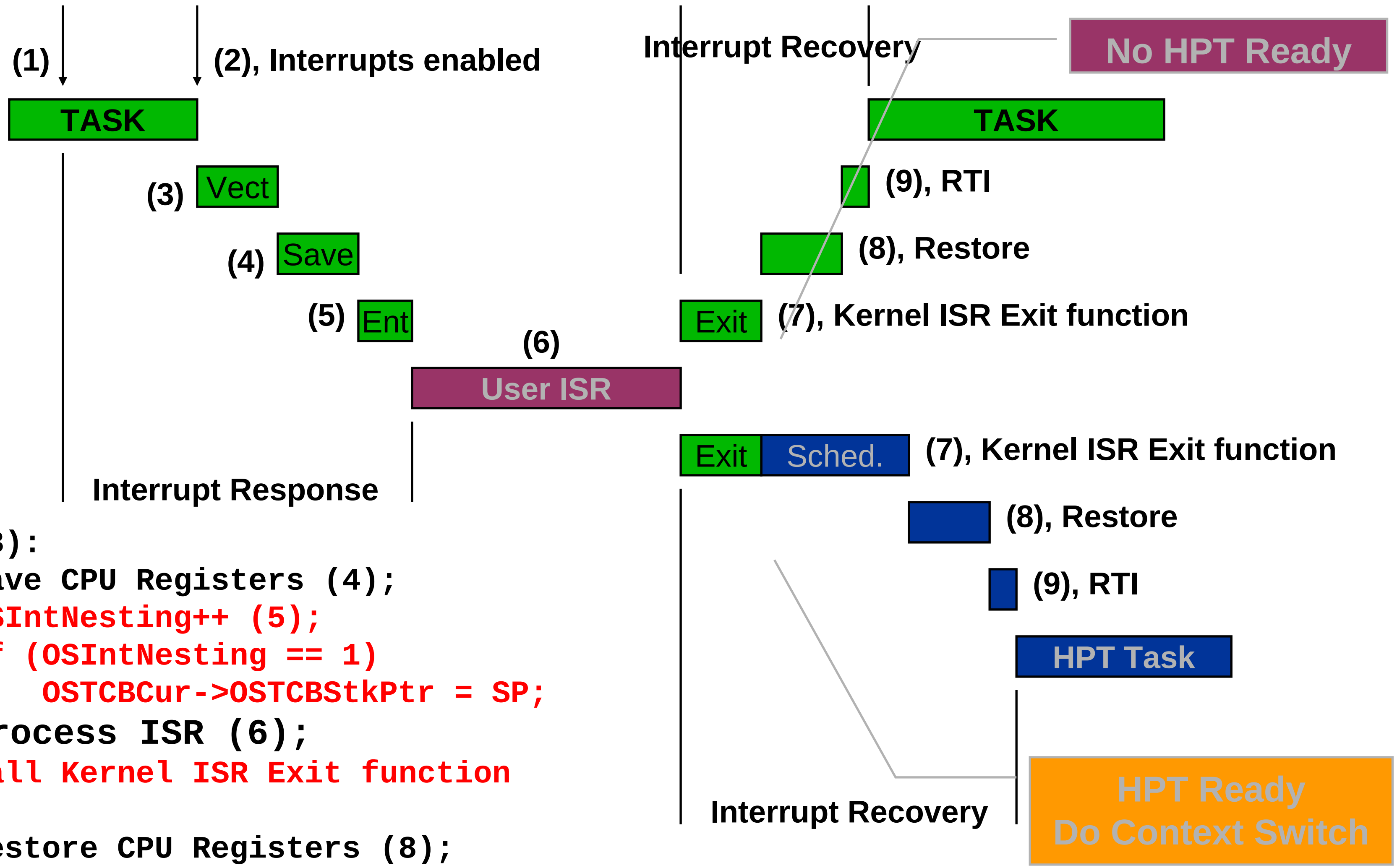
ISR from vector

```
YourISR:
    Save CPU Registers;
    Notify kernel of ISR entry;
    Determine SOURCE of interrupt;
       Process ISR(s) (Your code!);
           /* Take care of device         */
           /* Buffer data                 */
           /* Clear interrupt             */
           /* Signal task to process data */
    Notify kernel about end of ISR;
    Restore CPU Registers;
    Return from Interrupt;
```

If a more important task is Ready, the Kernel will do a Context Switch

There are no HP Task Ready, Return to Interrupted Task!

Servicing Interrupts




```
ISR (3):
    Save CPU Registers (4);
    OSIntNesting++ (5);
    if (OSIntNesting == 1)
        OSTCBCur->OSTCBStkPtr = SP;
    Process ISR (6);
    Call Kernel ISR Exit function
(7);
    Restore CPU Registers (8);
    Return from Interrupt (9);
```

The Clock Tick ISR

- µC/OS-II requires a periodic interrupt source
  - Through a hardware timer
    - Between 10 and 100 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 the µC/OS-II called **`OSTimeTick()`**

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

Resource Sharing

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
    - Some CPUs don't allow you to do this in 'user' code
  - Lock/Unlock
  - Semaphores
  - MUTEX (Mutual Exclusion Semaphores)

Resource Sharing
(Disable and Enable Interrupts)

- When access to resource is done quickly
  - Be careful with Floating-point!

- Disable/Enable interrupts is the fastest way!

```
rpm = 60.0 / time;
OS_ENTER_CRITICAL();
Global RPM = rpm;
OS_EXIT_CRITICAL();
```

Resource Sharing
(Lock/Unlock the Scheduler)

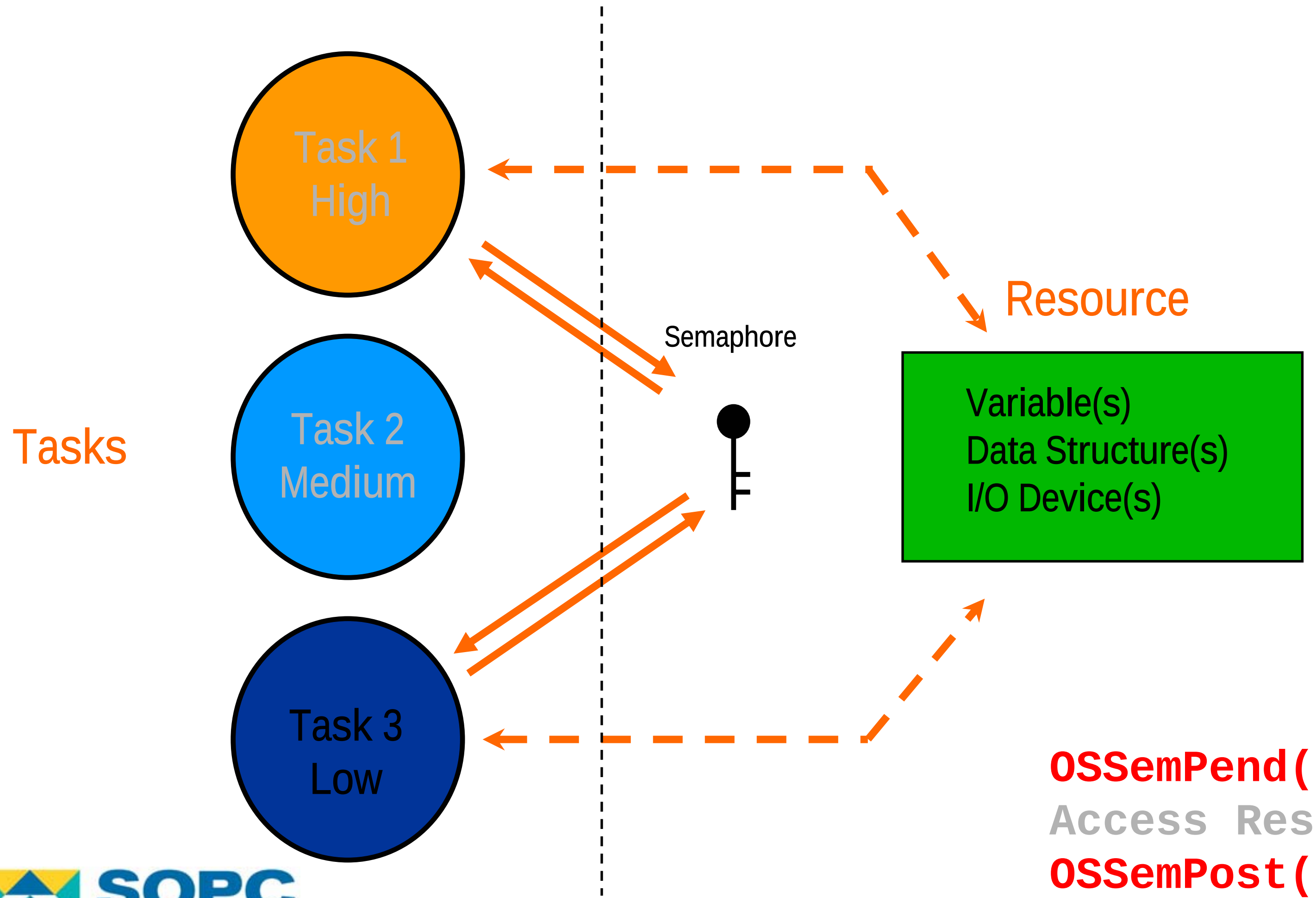
- ‘Lock’ prevents the scheduler from changing tasks
  - Interrupts are still enabled
  - Can be used to access non-reentrant functions
  - Can be used to reduce priority inversion
  - Same effect as making the current task the Highest Priority Task
- ‘Unlock’ invokes the scheduler to see if a High-Priority Task has been made ready while locked

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

Mutual Exclusion
(Semaphores)

- Used when time to access a resource is longer than the kernel interrupt disable time!
- Binary semaphores are used to access a single resource
- Counting semaphores are used to access multiple resources

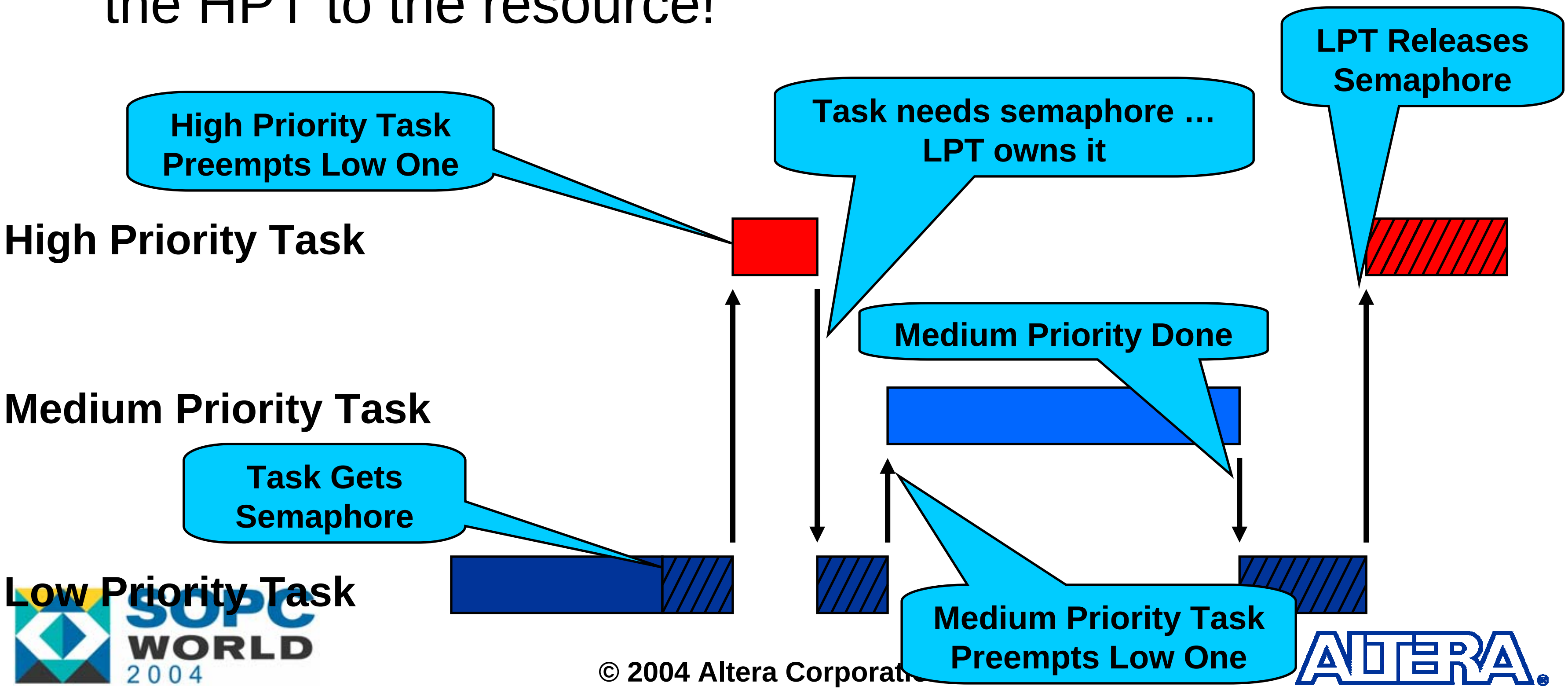
Mutual Exclusion
(Semaphores)




```
OSSemPend(..);
Access Resource;
OSSemPost(..);
```

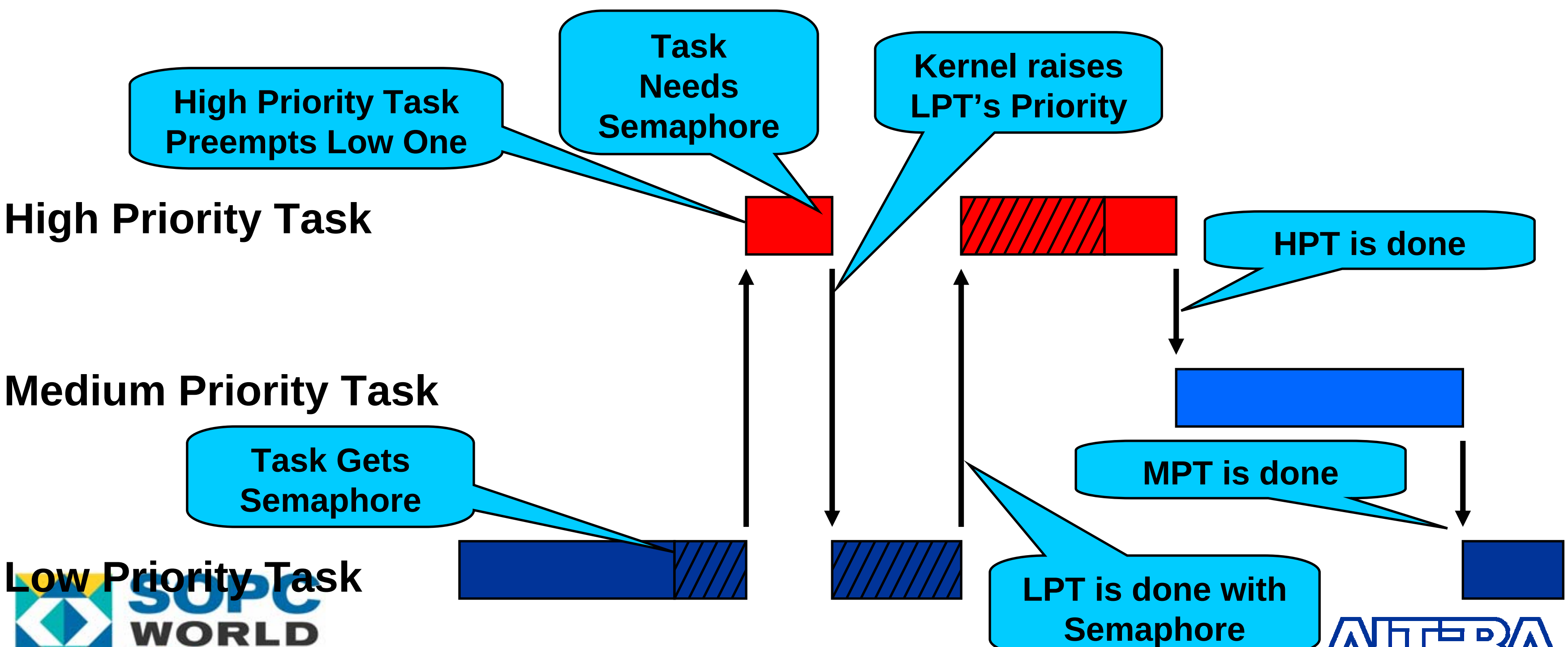
Semaphores
(Priority Inversion)

- Delay to a task's execution caused by interference from lower priority tasks
- All tasks of medium priority would delay access of the HPT to the resource!



Semaphores
(Priority Inheritance)

- Low Priority task assumes priority of High Priority task while accessing semaphore.
- µC/OS-II has automatic *priority ceiling* protocols.

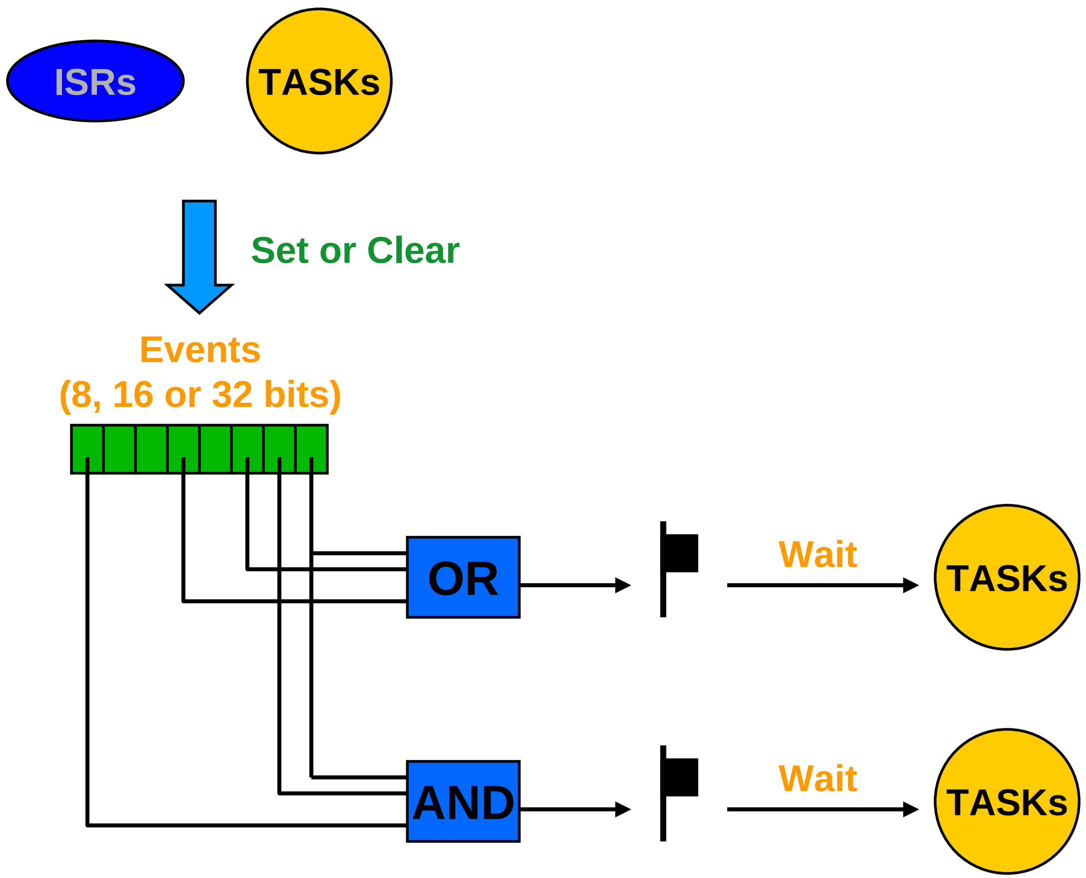


Intertask Communication

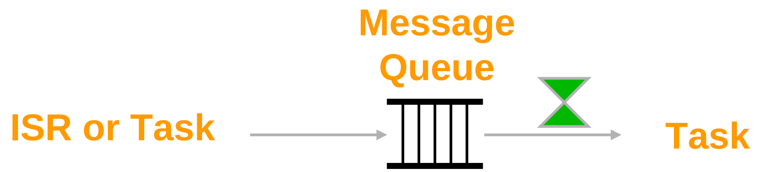
Event Flags

- Synchronization of tasks with the occurrence of multiple events
- Events are grouped
  - 8, 16 or 32 bits per group
- 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


ISRs
TASKs
Set or Clear
Events
(8, 16 or 32 bits)
OR
AND
Wait
TASKs
Wait
TASKs

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

Miscellaneous Services

Stack Checking

- Stacks can be checked at run-time to see if you allocated sufficient RAM
  - Assumes you created your task with OSTaskCreateExt()
- Allows you to know the 'worst case' stack growth of your task(s)
- Assumes stack is cleared when task is created
  - Could check for other patterns than 0x00

SOPC WORLD 2004

ALTERA

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.

```
INT8U   OSTaskDel(INT8U prio);
INT8U   OSTaskDelReq(INT8U prio);
```

Changing a Task's Priority

- Kernel can allow tasks to change their priority (or the priority of others) at run-time

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

Memory Management

- µC/OS-II provides fixed-sized memory block management
  - Prevents fragmentation
- Multiple ‘partitions’ can be created with each having a different block size
- You MUST ensure that you return blocks to the proper partition.
- Partitions can be ‘extended’ from a larger block.

Initialization

- µC/OS-II provides an initialization function
- You must create at least one task before starting multitasking

```
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

- You should initialize the ‘ticker’ in the first task to run.
  - Setup hardware timer,
  - Enable timer interrupt

```
void AppTaskStart (void)
{
    /* Task Initialization                          */
    /* Setup hardware timer for CLOCK tick */
    /* Enable GLOBAL interrupts                     */
    /* Create OTHER tasks as needed                 */

    while (1) {
        /* Task body (YOUR code)                    */
    }
}
```

POP-Quiz

- 다음 중 μC/OS-II에 대한 설명 중 잘못된 것은 무엇입니까?

  A) Task가 Semaphore를 획득하는 방법은 가장 우선순위가 높은 Task인 경우이다.

  B) μC/OS-II는 자동으로 Stack검사를 하지는 않는다.

  C) Non-preemptive Real-time Kernel이다.

  D) 최대 64개의 Task를 지원한다.