Micrium OS Kernel Training

Class Objectives

Upon completing this class, you will …

- Understand how applications that incorporate a real-time kernel differ from foreground/background, or super-loop, applications
- Have experience with one of Micrium's real-time kernels
- Have a solid understanding of Micrium OS Kernel's API
- Know how to utilize many of the services that Micrium OS Kernel provides

Agenda

Introduction
Foreground/Background Systems
Kernel-Based Applications
Lab 1

`main()`
Tasks
Lab 2

Scheduling and Context Switches
Interrupts and Exceptions
Lab 3

Synchronization
Lab 4

Agenda (Cont.)

Mutual Exclusion
Lab 5

Inter-Task Communication & Dynamic Memory Pools
Lab 6

Software Timers
Lab 7
Conclusion

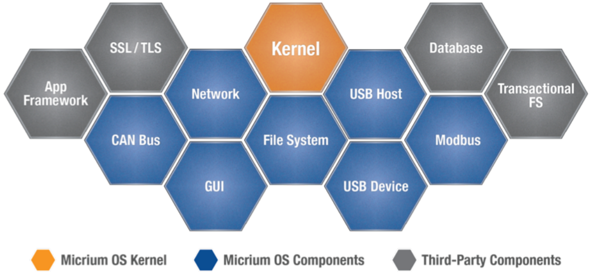
Introduction

An Overview of Micrium

- Incorporated in 1999. Acquired by Silicon Labs in 2016.
- Headquarters in South Florida, with an additional office in Montreal
- Provider of high-quality embedded software
- Known for:
  - Remarkably clean code
  - Thorough documentation
  - Top-notch technical support
  - A full lineup of products, including real-time kernels, protocol stacks, file system and debug tools.

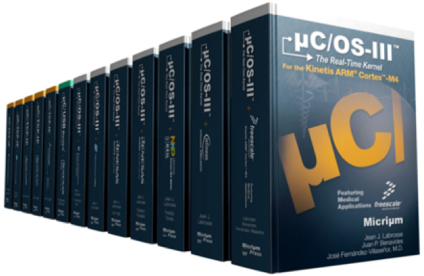
Who Are We?

- An Embedded Software Company
  - Proven - Shipping µC/OS For Over 25 Years
  - Most Widely Deployed RTOS (UBM 2015 Survey)
  - Strong in Safety Critical Applications
  - High Quality and Robust Code



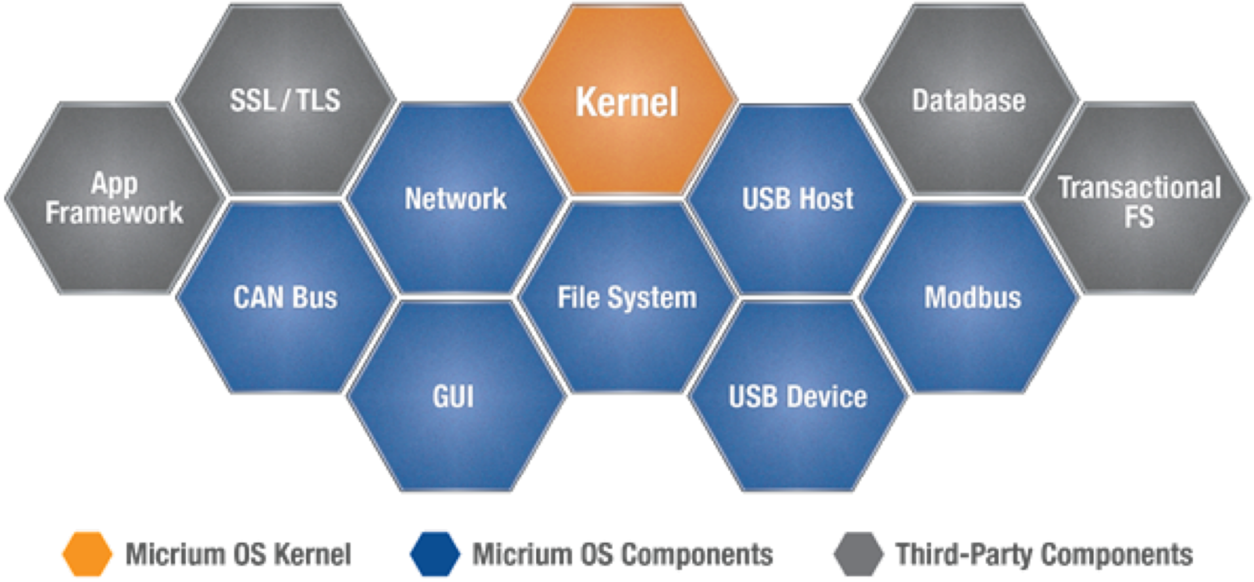
RTOS

Debug Tools



Education

Micrium OS


SSL / TLS
Kernel
Database
App Framework
Network
USB Host
Transactional FS
CAN Bus
File System
Modbus
GUI
USB Device
Micrium OS Kernel
Micrium OS Components
Third-Party Components

Micrium OS Kernel

- Based off the **extremely** successful µC/OS-III kernel
  - Updated error handling
- A **reliable** kernel with an efficient, preemptive scheduler
  - Supports round-robin scheduling at each task priority level
- Supports an **unlimited** number of tasks and other kernel objects
- Highly configurable
  - Gives the ability to enable and disable most parts of the kernel to save space
  - ROM size ranges from **6-24 kBytes**
  - RAM size is typically **3-4 kBytes**
- Delivered in source-code form, and its thorough documentation helps to ensure a smooth user experience
- Built-in performance measurement capabilities

Micrium OS: Communication Software

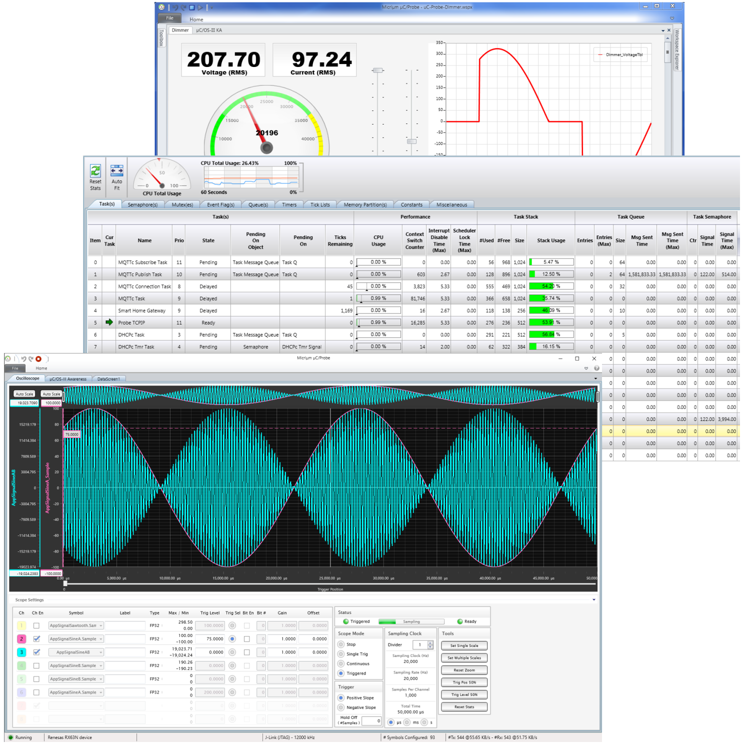
- **Network**
  - Features dual IPv4 and IPv6 support, an SSL/TLS socket option
  - Supports DHCP, DNS, HTTP, MQTT, SNTP, Telnet, SMTP, TFTP, FTP
- **USB Device**
  - Support for Audio, CDCACM, CDCEEM, HID, MSC, and Vendor classes
- **USB Host**
  - A USB host stack for embedded systems equipped with a USB host or OTG controller.
  - Includes support for MSC, HID, CDC ACM, USB2Ser and AOAP classes

Micrium OS: Storage and Display Software

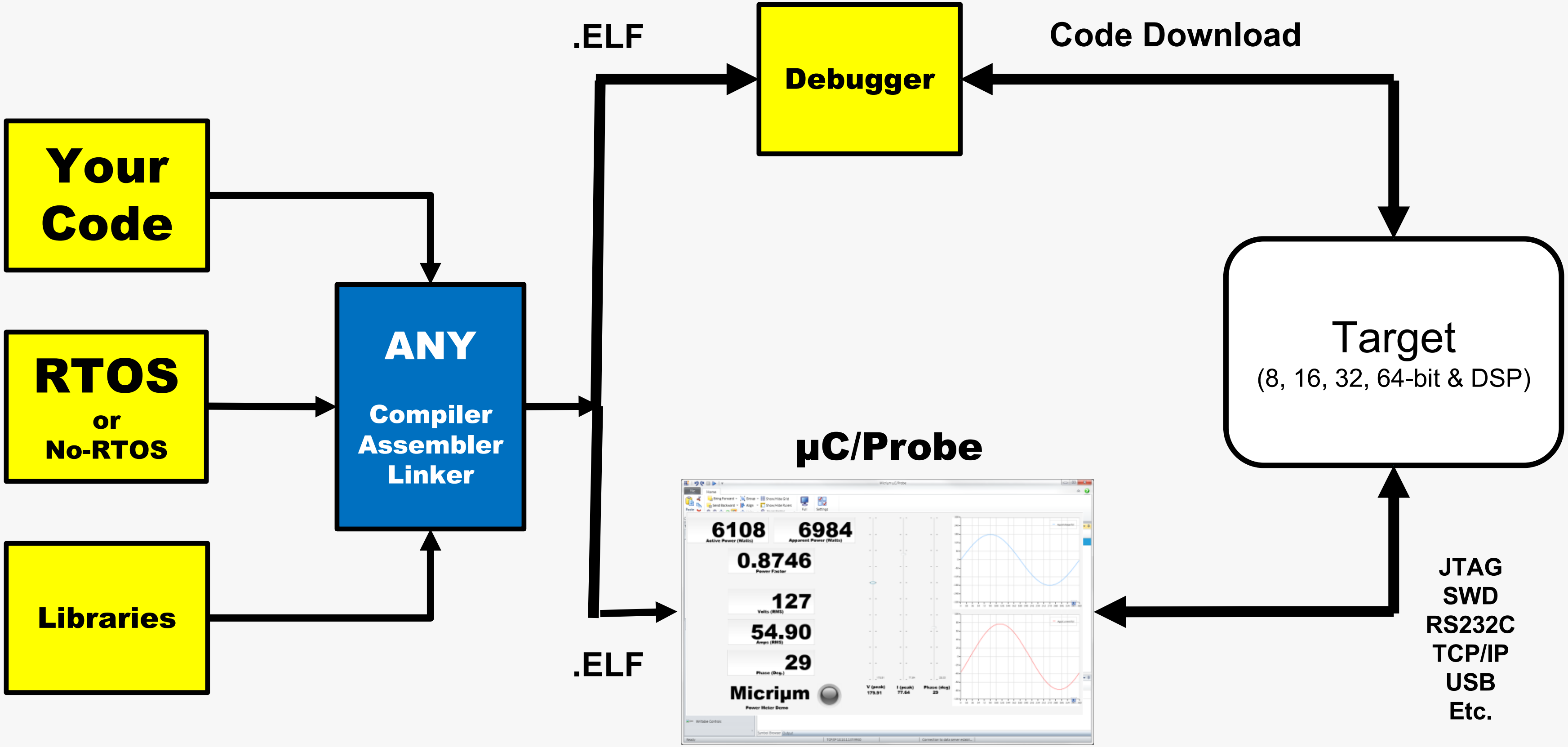
- **File System**
  - A FAT file system compatible with a wide range of storage devices. An optional journaling component provides fail-safe operation
- **Graphical User Interface**
  - A graphical user interface solution capable of satisfying a variety of display needs, from simple monochrome text to rich, full-color images and touch-screen functionality

µC/Probe

- Windows-based Universal Dashboard
  - Gauges, Graphs, Indicators, LEDs, Sliders,
  - Buttons, Oscilloscope, Kernel Awareness, etc.
- Works with ANY CPU
  - 8-, 16, 32-, 64-bit and DSPs
- Interface to target:
  - J-Link, CMSIS-DAP,
  - Proxy through Debuggers,
  - RS232C, TCP/IP, USB
- Integrated with Simplicity Studio
  - Windows-only

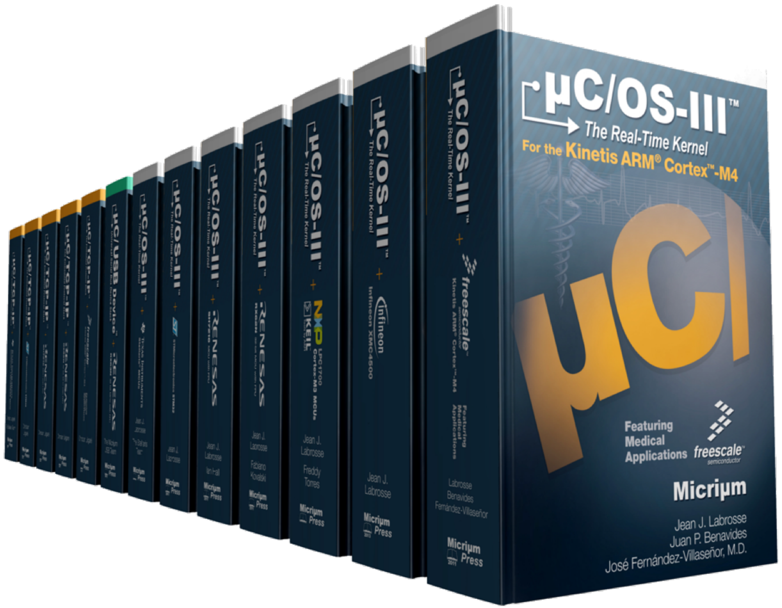


µC/Probe


.ELF
Debugger
Code Download
Your Code
ANY
Compiler
Assembler
Linker
RTOS
or
No-RTOS
Libraries
Target
(8, 16, 32, 64-bit & DSP)
µC/Probe
6108
6984
0.8746
127
54.90
29
Micrium
.ELF
JTAG
SWD
RS232C
TCP/IP
USB
Etc.

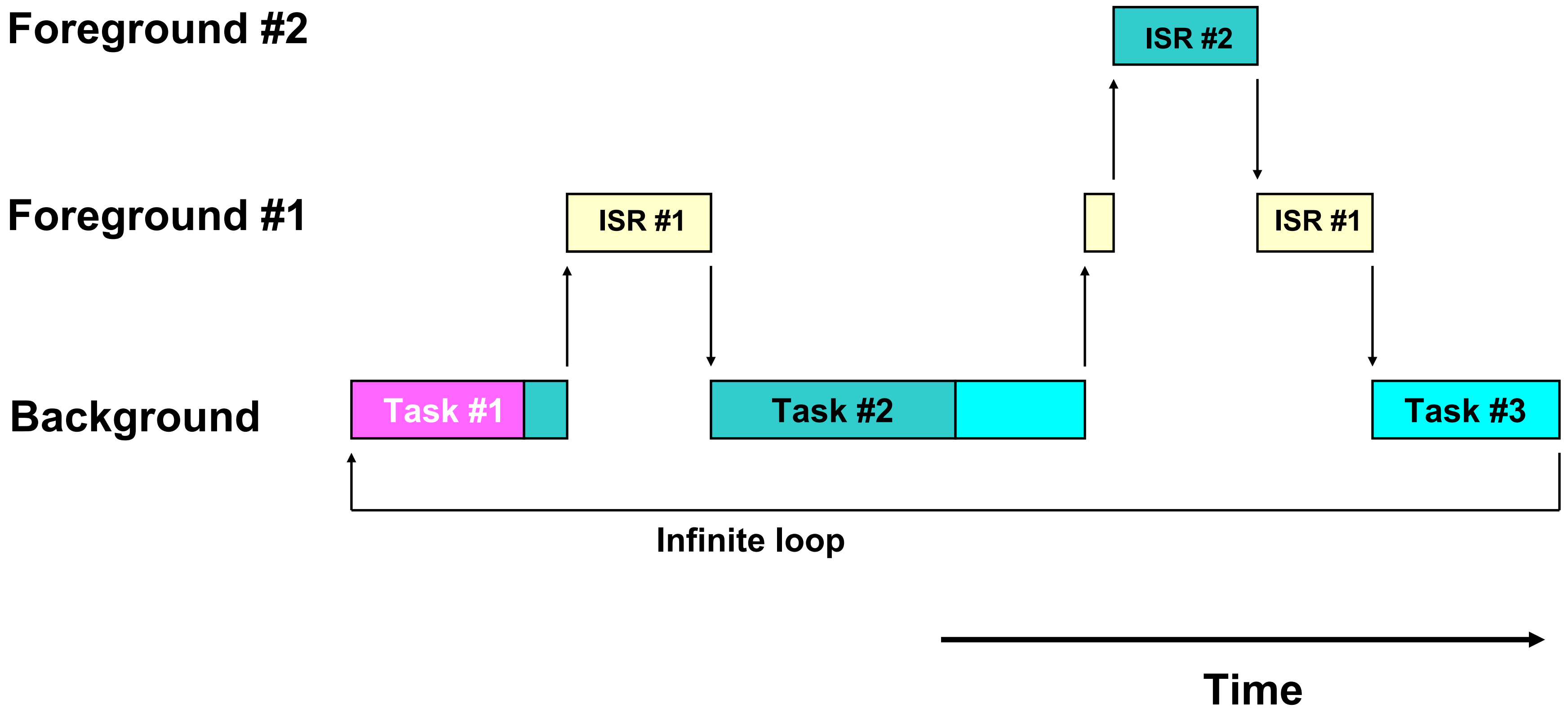
Micrium Press

- µC/OS-III Books (7 books)
- µC/TCP-IP Books (5 books)
- µC/USB-Device Book (1 book)
- 5 Books Translated to Mandarin
- All Books Available as **Free PDFs**
- Printed Versions Available on Amazon.com



Foreground / Background Systems

Foreground/Background Model



Pseudo-Code

Background

```
int  main (void)
{
    Perform initializations;
    while (1) {
        ADC_Read();
        SPI_Read();
        USB_Packet();
        LCD_Update();
        Audio_Decode();
        File_Write();
    }
}
```

Foreground

```
void  USB_ISR (void)
{
    Clear interrupt;
    Read packet;
}
```

Benefits

- No upfront cost
- Minimal training required
  - Developers don’t need to learn a kernel’s API
- No extra memory resources to accommodate a kernel
  - There is a small amount of overhead associated with a kernel

Drawbacks: Simplistic Scheduling

Foreground

- On demand scheduling
  - ISRs execute when urgent events occur
- Preferred for high-priority code but can become difficult to manage
  - Each ISR typically leaves at least a portion of a system's interrupts disabled

Background

- Fixed sequence of operations
  - Functions must wait their turn

```
while (1) {
    ADC_Read();
    SPI_Read();
    USB_Packet();
    LCD_Update();
}
```

Drawbacks: Maintenance and Upgrade Difficulties

- Performance requirements may dictate extensive use of ISRs
  - Porting code to a new interrupt controller, with a different prioritization scheme, can be challenging
  - Debugging ISRs presents many potential problems
- It can be difficult to make additions (of ISRs or background calls) without negatively impacting existing code
  - Upgrades or improvements may require moving code between foreground and background
  - Development teams must be closely coordinated

Drawbacks: Polling

```
while (1) {
    ADC_Read();
    SPI_Read();
    USB_Packet();
    LCD_Update();
    Audio_Decode();
    File_Write();
}
```

```
void ADC_Read(void) {
    while((ADC_ConvComplete()) == 0) {
        ;
    }
    Process analog value;
}
```

```
void File_Write(void) {
    while((File_DevRdy()) == 0) {
        ;
    }
    Write to device;
}
```

CPU cycles are wasted waiting for hardware events

Drawbacks: Counters

```
void  main (void)
{
    unsigned short i;

    i = 0;
    while (1) {
        ADC_Read();
        if ((i % 8192) == 0) {
            SPI_Read();
        }
        USB_Packet();
        LCD_Update();
        if ((i % 1024) == 0) {
            Audio_Decode();
        }
        File_Write();
        i++;
    }
}
```

Within the background there are multiple rates of execution

All rates are based on the execution time of the loop

Drawbacks: Repetitive Function Calls

```
while (1) {
    ADC_Read();
    LCD_Update();
    SPI_Read();
    USB_Packet();
    LCD_Update();
    Audio_Decode();
    File_Write();
    LCD_Update();
}
```

Duplicate calls to a single function must be made during just one iteration of the main loop

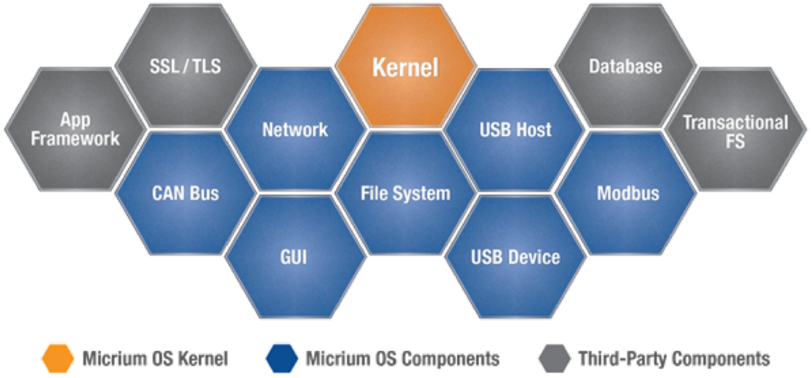
The Bottom Line: Is a Kernel Absolutely Necessary?

- Any application could be written without a kernel
- In the absence of a kernel, scheduling decisions are made when the code is written
  - The aforementioned issues must be considered anew for each version of an application
- A kernel can be seen as a portable and intelligent scheduling framework that simplifies the scheduling decisions application developers must make

Kernel-Based Applications

Operating System vs Kernel

- The terms operating system and kernel are often used interchangeably
- A kernel is actually a subset of an operating system
  - It can be viewed as the glue that holds the other components together
- Micrium OS Kernel is a real-time preemptive kernel!



Real-Time

- If a task must be completed within a given time, it is said to be a real-time task
  - In other words a real-time task is one that has a deadline
- There are three different categories of real-time tasks
  - Hard
    - If a task misses a deadline, it is considered a catastrophic or irrecoverable error
    - Avionics, medical, control systems
  - Firm
    - Infrequency misses of a task's deadline are tolerable but value of task's completion is useless
    - Manufacturing systems
  - Soft real-time tasks
    - Frequent misses of a task's deadline are ok but usefulness of task's completion degrade after the deadline
    - Weather monitoring station, video streaming

Determinism

- Fast software is not necessarily real-time software
- *Determinism* is a desirable quality in real-time software
- Software that is **deterministic** has a **bounded** response time to events

Real-Time Kernel Benefits

- Developers who use a real-time kernel don’t have to implement a scheduler and related services
- Typically, applications that incorporate a kernel are much easier to expand than foreground/background systems
  - Adding low-priority tasks generally don’t impact higher priority tasks.
  - Kernels help teams of multiple developers.
- The best kernels have undergone thorough testing
  - Formal testing is performed by the software’s developers, while its users may engage in informal testing
  - Unlike ad-hoc scheduling code, kernels are highly unlikely to introduce bugs into an application

When is a Real-Time Kernel Needed?

- When you want to build a framework for your application
- When you have some time-sensitive tasks
- When you use one or more 32-bit CPUs
- When you have multiple programmers
- When you need complex OS services
  - Protocol stacks, GUI, File System, etc.
- When you have excessive polling loops

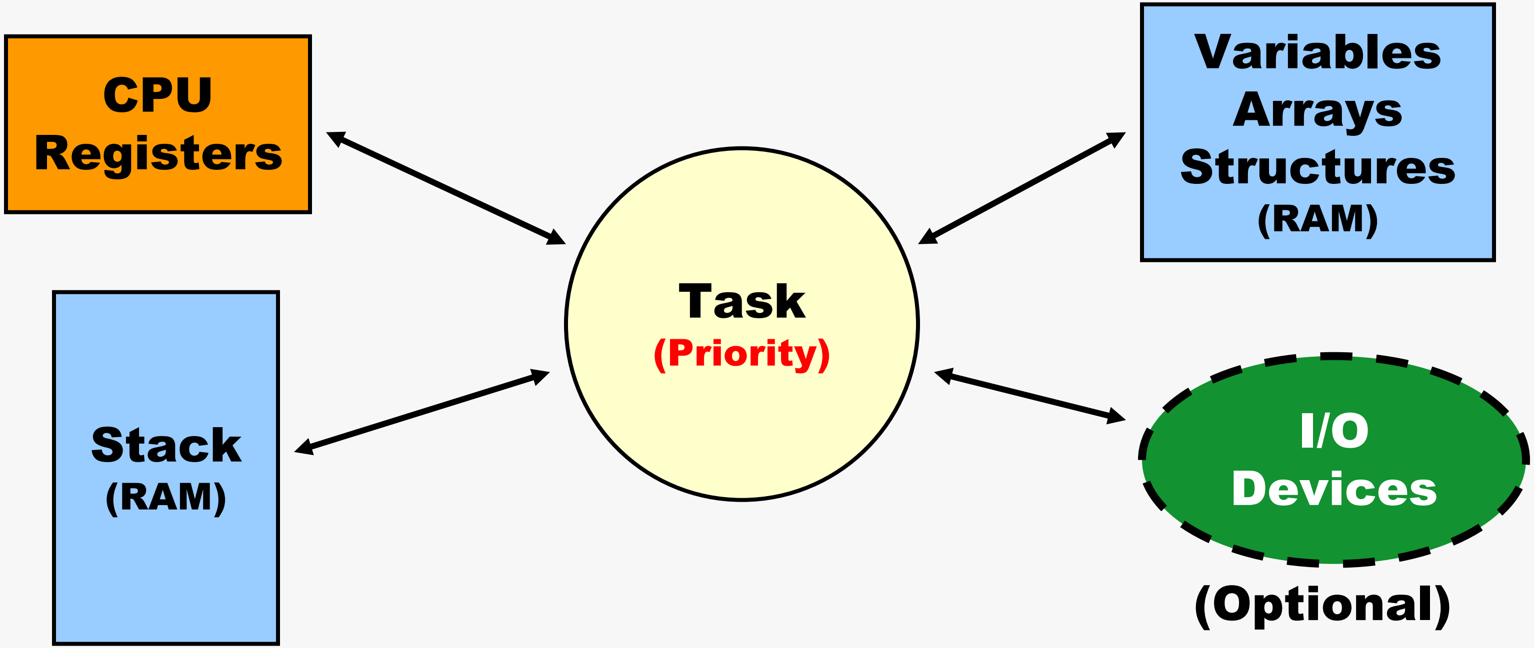
Kernel Hardware Requirements

- Context Switch Support (PendSV)
- Typically a timer or some other source of periodic interrupts
  - Not mandatory in Micrium OS Kernel
- A small amount of ROM and RAM
  - The memory footprint of Micrium OS Kernel is configuration-dependent
    - ROM size ranges from **6-24 kBytes**
    - RAM size is typically **3-4 kBytes**
  - RAM usage can be monitored with µC/Probe

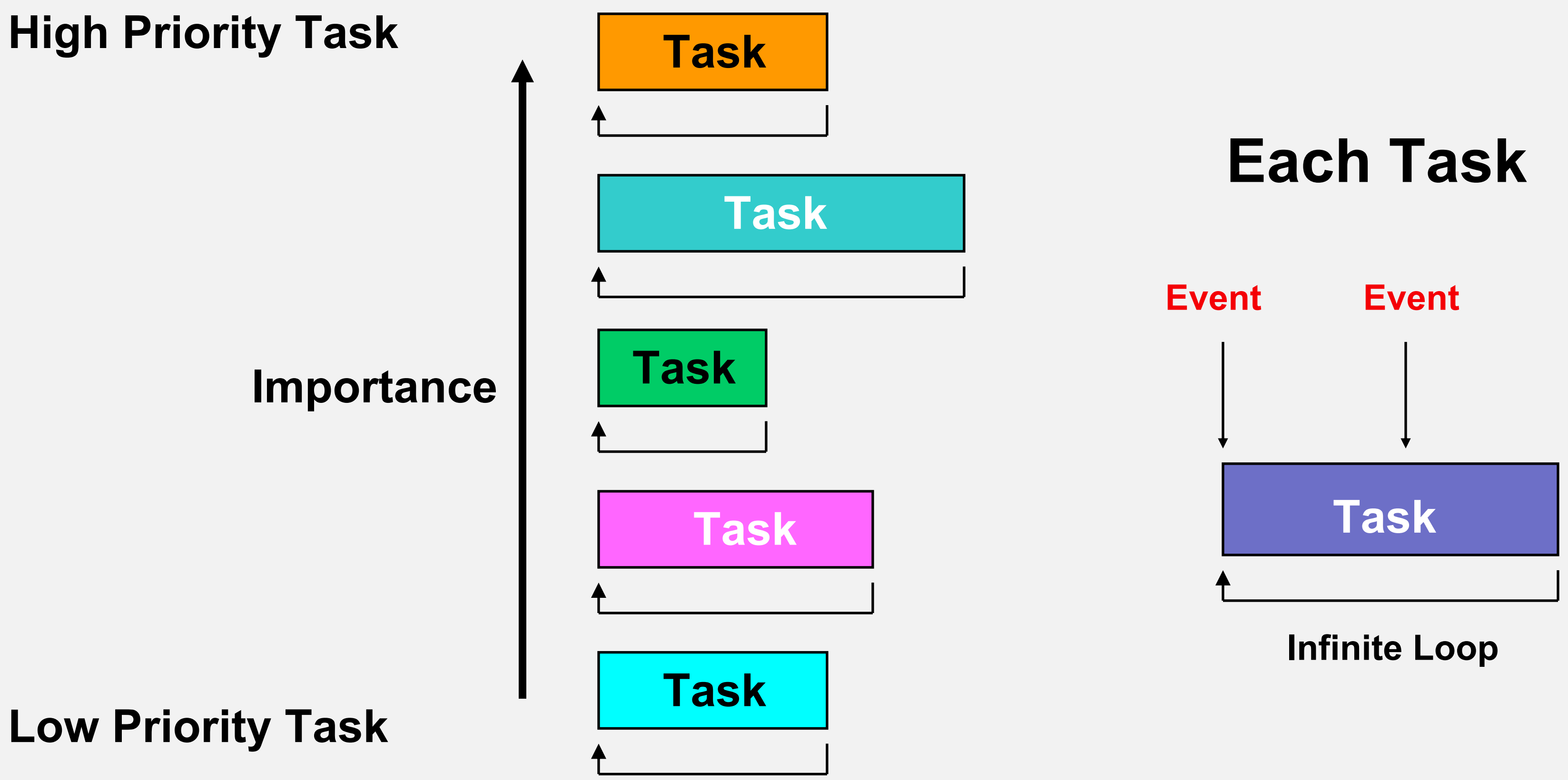
Scheduling

- A kernel’s primary function is to schedule the various tasks comprising an application
- Micrium OS Kernel, like the Micrium kernels that came before it, is a pre-emptive kernel
  - Scheduler always attempts to run the highest priority task that is in the ready state
- Round-robin scheduling is also an option in Micrium OS Kernel
  - Each task is run for a designated period of time

What is a Task?



Tasks in a Kernel-Based Application



A Kernel-Based Application

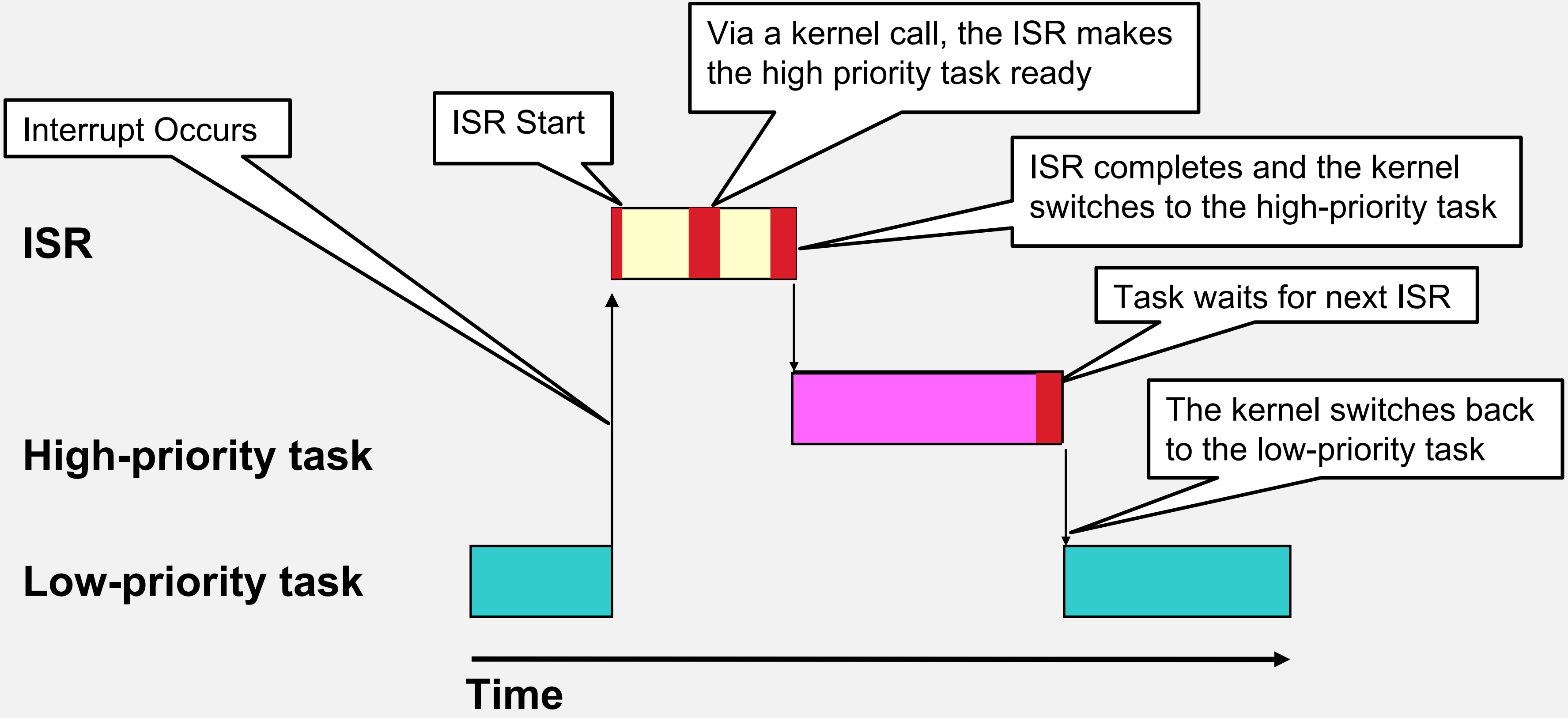
Tasks

```
void  AppTaskADC (void *p_arg)
{
    while (1) {
        ADC_Read();
        Sleep for 1 ms;
    }
}
void  AppTaskUSB (void *p_arg)
{
    while (1) {
        Wait for signal from ISR;
        USB_Packet();
    }
}
```

ISRs

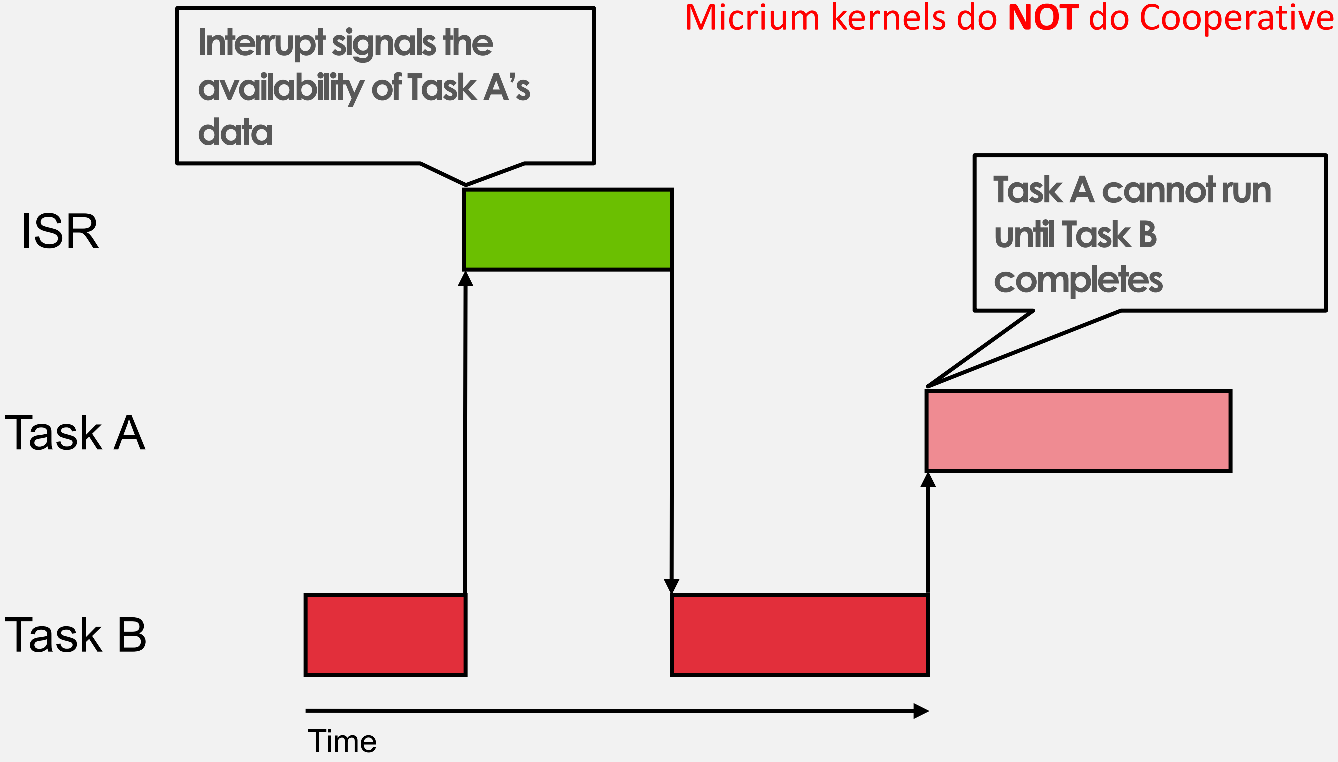
```
void  AppISRUSB (void)
{
    Clear interrupt;
    Signal USB Task;
}
```

Preemptive Scheduling

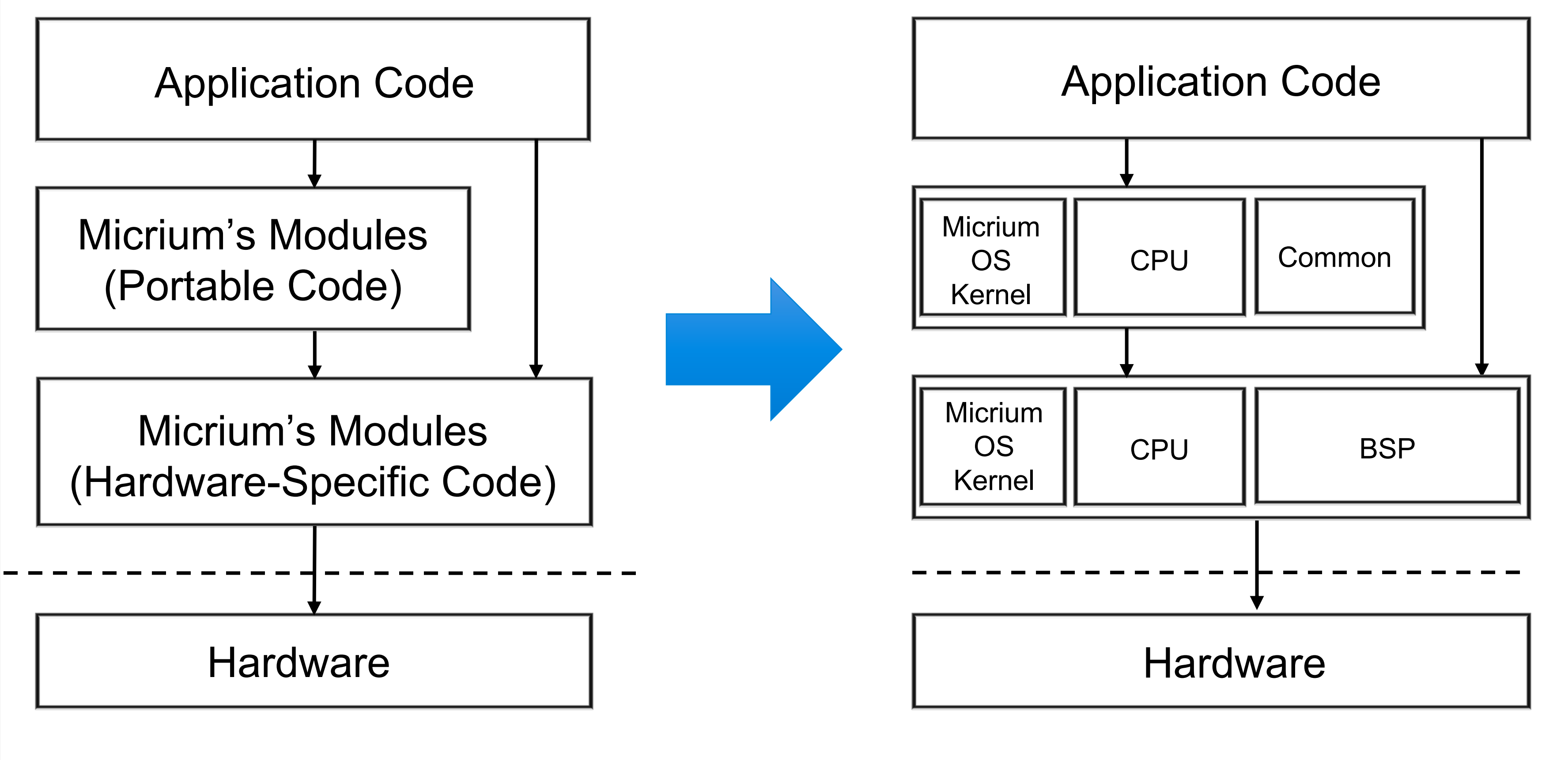


Cooperative Scheduling

Micrium kernels do **NOT** do Cooperative Scheduling!



A Typical Micrium OS Kernel Based Application



Source Code Comments

- The Micrium OS Kernel source code is thoroughly commented
  - **Code** on the **left**
  - **Comments** on the **right**
- A block of descriptive comments precedes every function
- The comment blocks are a convenient means of learning how each function works

1. vim

```
/*
************************************************************************************************************************
*                                                    INITIALIZATION
*
* Description: This function is used to initialize the internals of uC/OS-III and MUST be called prior to
*              creating any uC/OS-III object and, prior to calling OSStart().
*
* Arguments  : p_err         is a pointer to a variable that will contain an error code returned by this function.
*
*                                OS_ERR_NONE    Initialization was successful
*                                Other          Other OS_ERR_xxx depending on the sub-functions called by OSInit().
* Returns    : none
************************************************************************************************************************
*/

void  OSInit (OS_ERR  *p_err)
{
#if (OS_CFG_ISR_STK_SIZE > 0u)
    CPU_STK      *p_stk;
    CPU_STK_SIZE  size;
#endif



#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

    OSInitHook();                                               /* Call port specific initialization code               */

    OSIntNestingCtr       =           0u;                       /* Clear the interrupt nesting counter                  */

    OSRunning             =  OS_STATE_OS_STOPPED;               /* Indicate that multitasking has not started           */

    OSSchedLockNestingCtr =           0u;                       /* Clear the scheduling lock counter                    */

    OSTCBCurPtr           = (OS_TCB *)0;                        /* Initialize OS_TCB pointers to a known state          */
    OSTCBHighRdyPtr       = (OS_TCB *)0;

    OSPrioCur             =           0u;                       /* Initialize priority variables to a known state       */
    OSPrioHighRdy         =           0u;
```

Section Summary - Kernels

- A kernel is a **subset** of an RTOS
- Kernel is **code** and provides a set of **APIs**
  - Micrium OS Kernel has ~**70** APIs, **20** or so are commonly used
- The primary service of a kernel is **task management**
  - Micrium OS Kernel supports and unlimited number of tasks
  - Each task needs a priority, a stack, variables and optional I/Os
- Micrium OS Kernel is a Real-Time Preemptive Kernel
  - Will always run the '**Highest-Priority Task**' ready
  - Supports Round Robin scheduling at every priority level

Lab #1

`main()`

The Role of main()

- The first function executed following startup
- Typically, `main()` initializes Micrium OS Kernel via a standard sequence of three API calls
  - `OSInit()`
  - `OSTaskCreate()`
  - `OSStart()`
- The implementations of `main()` in Micrium's example projects are fairly consistent across hardware platforms

Typical main()

```
void  main (void)
{
    RTOS_ERR  err;


    OSInit(&err);                                              /* Init Micrium OS Kernel.          */

    OSTaskCreate((OS_TCB     *)&AppTaskStart_TCB,              /* Create the start task            */
                 (CPU_CHAR   *)"Start Task",
                 (OS_TASK_PTR )AppTaskStart,
                 (void       *)0,
                 (OS_PRIO     )APP_TASK_START_PRIO,
                 (CPU_STK    *)&AppTaskStart_Stk[0],
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY  )10,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (RTOS_ERR   *)&err);

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

OSInit()

- Must be invoked before any Micrium OS Kernel services are used
- Responsible for initializing kernel data structures
- System tasks are created by `OSInit()`
- The extent of initializations performed by `OSInit()` depends on configuration constants
  - Constants can be found in **os_cfg.h**

Configuring Micrium OS Kernel

- Micrium OS Kernel is highly configurable
  - Memory footprint is configuration-dependent
  - All configuration files are located in the cfg directory
- There are a few configuration header files that a developer needs to modify
  - **os_cfg.h**
    - Allows application developers to scale the kernel's services
  - **rtos_description.h**
    - Defines what modules are enabled in a Micrium OS application
    - Must enable **`RTOS_MODULE_KERNEL_AVAIL`**
  - **common_cfg.h**
    - Modify the size of the internal Micrium heap
      - May be used by the system tasks, kernel objects and other Micrium OS modules

Micrium OS Kernel’s SystemTasks

- Number of system tasks created by `OSInit()` ranges from 0 to 4
  - Tick Task
  - Idle Task
  - Timer Task
  - Statistics Task
- System task allotment for a given application is determined by configuration parameters
- Stack sizes, and in some cases priorities, must be configured for system tasks

The Idle Task

- Automatically assigned the lowest possible priority
  - Priority is assigned via **`OS_CFG_PRIO_MAX – 1`**
  - No other task may be given this priority
- Runs when all other tasks are unable to do so
- Repeatedly invokes a hook function, `OSIdleTaskHook()`
  - Can be used to put the CPU to sleep when idling
- If disabled, it is replaced by a loop in the kernel's scheduler

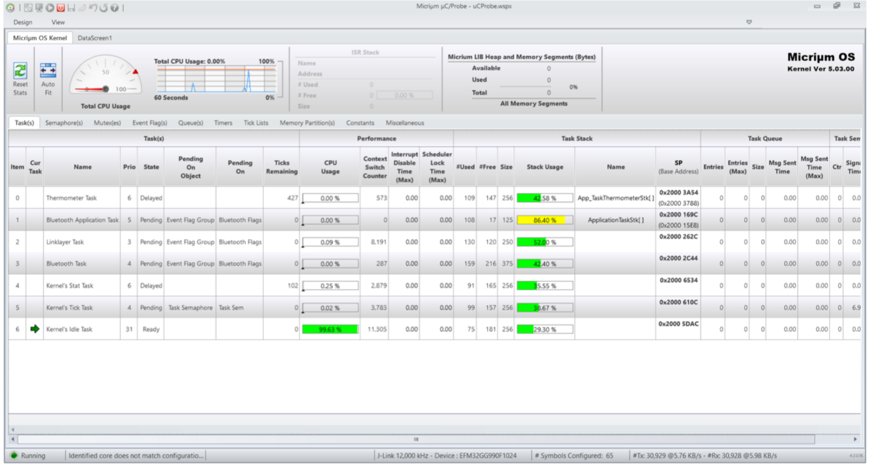
```
void OS_IdleTask (void)
{
    while (1) {
        OSIdleTaskCtr++;
        OSIdleTaskHook();
    }
}
```

The Tick Task

- Needed to implement time-based services
  - `OSTimeDly()`, `OSTimeDlyHMSM()`, all timeouts on `Pend()` calls
- Is synchronized with a periodic interrupt (the tick interrupt)
- Has a configurable priority
- Can be disabled for applications that don’t require the use of time delays and timeouts

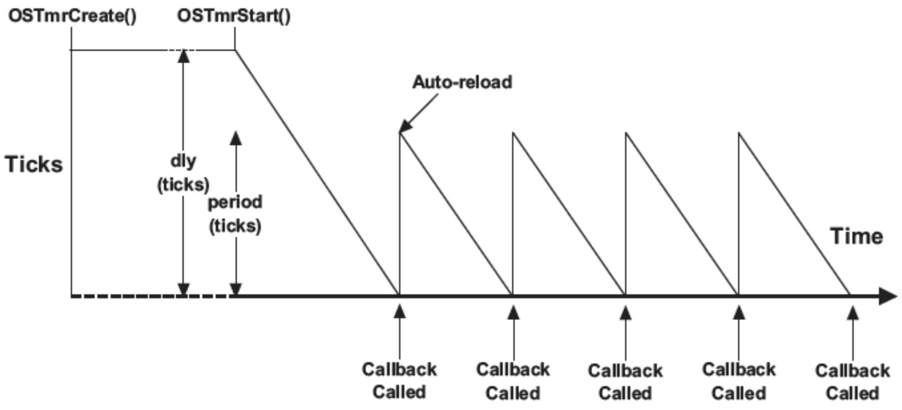
The Statistics Task

- Keeps track of run-time statistics
  - Current and peak CPU usage
  - Current and peak per-task CPU usage
  - Per-task stack usage
  - And more!
- Has a configurable priority
- Can be removed if is not needed
  - Some developers opt to enable statistics during debugging only
- Requires additional initializations in application code



The Timer Task

- Manages all software timers
  - Supports an unlimited number of timers
  - Timers can be one-shot or cyclic
  - Each timer can call a ‘callback function’ when it expires
- Can be removed from applications that don’t use software timers
- Synchronized with the same interrupt as the tick task



Creating the First Task

- After `OSInit()` has completed, application tasks can be created via calls to `OSTaskCreate()`
- Micrium recommends creating just one task in main()
  - Creating multiple tasks in `main()` causes problems for the statistics task
  - The first application task can create any other needed tasks
- Any task in a Micrium OS Kernel application can create other tasks
- Task creation is prohibited in ISRs

OSStart()

- The last call in `main()` must be `OSStart()`
  - This is what starts multitasking
- `OSStart()` is a very short function
  - Determines the highest priority task
  - Calls `OSStartHighRdy()`
    - Assembly function to perform first context switch
- `OSStart()` does not return
  - Following the call to this function, execution proceeds in tasks and ISRs

2. vim

```c
void  OSStart (RTOS_ERR  *p_err)
{
    OS_OBJ_QTY  kernel_task_cnt;


    OS_ASSERT_DBG_ERR_PTR_VALIDATE(p_err, ;);

    OS_ASSERT_DBG_ERR_SET((OSInitialized == DEF_TRUE), *p_err, RTOS_ERR_NOT_INIT, ;);


    kernel_task_cnt = 0u;                                       /* Calculate the number of kernel tasks                */
#if (OS_CFG_STAT_TASK_EN == DEF_ENABLED)
    kernel_task_cnt++;
#endif
#if (OS_CFG_TASK_TICK_EN  == DEF_ENABLED)
    kernel_task_cnt++;
#endif
#if (OS_CFG_TMR_EN == DEF_ENABLED)
    kernel_task_cnt++;
#endif
#if (OS_CFG_TASK_IDLE_EN == DEF_ENABLED)
    kernel_task_cnt++;
#endif

                                                                /* Make sure at least one application task is created  */
    OS_ASSERT_DBG_ERR_SET((OSTaskQty > kernel_task_cnt), *p_err, RTOS_ERR_INVALID_CFG, ;);

                                                                /* Make sure kernel is not already running             */
    OS_ASSERT_DBG_ERR_SET((OSRunning == OS_STATE_OS_STOPPED), *p_err, RTOS_ERR_INVALID_STATE, ;);


    OSPrioHighRdy             = OS_PrioGetHighest();            /* Find the highest priority                           */
    OSPrioCur                 = OSPrioHighRdy;
    OSTCBHighRdyPtr           = OSRdyList[OSPrioHighRdy].HeadPtr;
    OSTCBCurPtr               = OSTCBHighRdyPtr;
#ifdef OS_SAFETY_CRITICAL_IEC61508
    OSSafetyCriticalStartFlag = DEF_TRUE;                       /* Prevent creation of additional kernel objects       */
#endif
    OSRunning                 = OS_STATE_OS_RUNNING;
    OSStartHighRdy();                                           /* Execute target specific code to start task          */
    RTOS_CRITICAL_FAIL_EXEC(RTOS_ERR_OS, ;);                    /* OSStart() is not supposed to return                 */
}
```

Error Handling

Error Handling in Micrium OS

- The last parameter in any Micrium OS API is reserved for the error value
  - This is true for not just the Kernel, but all modules
- The error parameter is defined as `RTOS_ERR`

```
typedef  struct  rtos_err {
        RTOS_ERR_CODE     Code;                 /**< Err code enum val.             */
#if (RTOS_ERR_CFG_EXT_EN == DEF_ENABLED)
#if (RTOS_ERR_CFG_STR_EN == DEF_ENABLED)
        CPU_CHAR const   *CodeText;             /**< Err code in string fmt.        */
        CPU_CHAR const   *DescText;             /**< Err desc string. */
#endif
        CPU_CHAR         *FileName;             /**< File name where error occurred. */
        CPU_INT32U        LineNbr;              /**< Line nbr  where error occurred. */
#ifdef PP_C_STD_VERSION_C99_PRESENT             /*    Only present if C99 enabled.   */
        const  CPU_CHAR  *FnctName;             /**< Fnct name where err occurred.   */
#endif
#endif
} RTOS_ERR;
```

Error Handling in Micrium OS

- Micrium OS provides a number of macros to assist developers when working with `RTOS_ERR`
- After making a Micrium OS API call, you should always check the code
  - `RTOS_ERR_CODE_GET(err) == RTOS_ERR_NONE`
  - `err.Code == RTOS_ERR_NONE` is not recommended
- Use the string and description GET() macros for logging
- Use `RTOS_ERR_COPY()` to copy an `RTOS_ERR` state
- `RTOS_ERR_SET()` may set the following
  - Error Code
  - Error Code Text
  - Error Description
  - File Name
  - Line Number
  - Function Name

Micrium OS RTOS_ERR Macros:

```
RTOS_ERR_CODE_GET(err_val)

RTOS_ERR_STR_GET(err_code)

RTOS_ERR_DESC_STR_GET(err_code)

RTOS_ERR_SET(err_var, err_code)

RTOS_ERR_COPY(err_dst, err_src)
```

Asserts

- The `APP_RTOS_ASSERT_CRITICAL()` and `APP_RTOS_ASSERT_DBG()` macros check if a given expression is evaluated with a positive result. If the result is not positive, the macro will do one of the following operations:
  - If `RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL(ret_val)` and/or `RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL(ret_val)` are defined, the macro will call these.
  - If they are not defined, `CPU_SW_EXCEPTION(ret_val)` will be called.
- The debug asserts typically check for conditions that are caused by invalid parameters or invalid configurations. They are used to notify the developer that something is not correct with the way the code is being used. Those can and should be disabled once development is completed.
- The critical asserts typically check for conditions from which it is practically impossible to recover at run-time. Therefore, if such a condition is detected, the program's execution should be suspended before any more damage occurs.

Asserts

```c
#define  VALUE_FIRST    1u
#define  VALUE_SECOND   2u
#define  VALUE_THIRD    3u

void  ExAssertFail (CPU_INT08U  value)
{
    CPU_INT08U  flag;

    switch (value) {
        case VALUE_FIRST:
        case VALUE_SECOND:
             flag = DEF_YES;
             break;

        case VALUE_THIRD:
             flag = DEF_NO;
             break;

        default:                                 /* Dflt case reached means an invalid arg was passed.   */
                                                 /* Call APP_RTOS_ASSERT_DBG_FAIL() to indicate err.     */
            APP_RTOS_ASSERT_DBG_FAIL(;);         /* Indicate ; as return value if function returns void. */
    }
    /* ... */
}
```

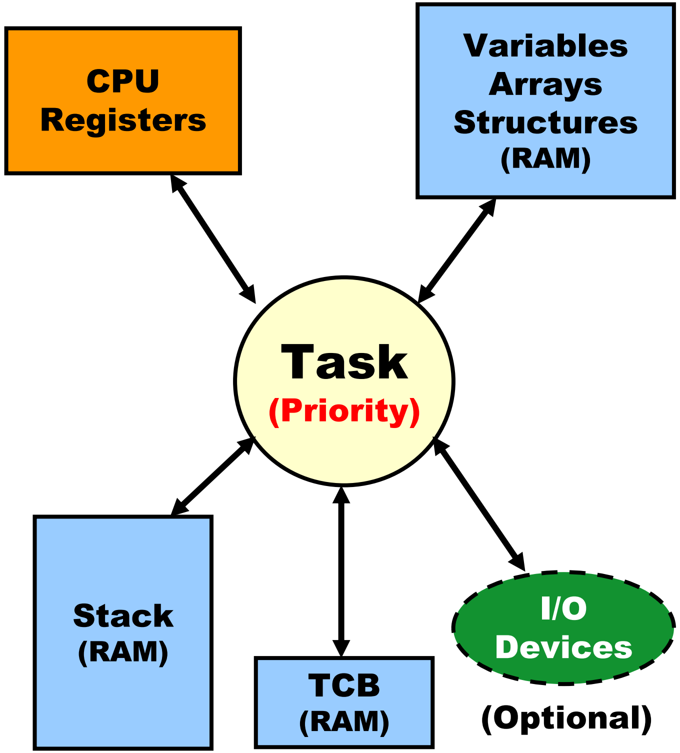
Asserts

- In addition to the application assert macros, Micrium OS has its own internal assert macros
  - `RTOS_ASSERT_CRITICAL(expr, err_code, ret_val)`
  - `RTOS_ASSERT_DBG(expr, err_code, ret_val)`
  - These macros should only be used by Micrium OS code, not your application
- Both `APP_RTOS_ASSERT and RTOS_ASSERT` rely on the definitions in **rtos_cfg.h** to define the behavior of those macros.
  - Default definition for `APP_RTOS_ASSERT_DBG()` and `RTOS_ASSERT_DBG()`
    - `while(1) {;}`
  - Default definition for `APP_RTOS_ASSERT_CRITICAL()` and `RTOS_ASSERT_CRITICAL()`
    - `CPU_SW_EXCEPTION(ret_val)`

Tasks

What makes up a Task?

- There are few required components
  - Task Control Block (TCB)
  - Stack space
  - Priority
- Normally, a task involves an infinite loop
  - Tasks must not return
- Initialization might precede the loop

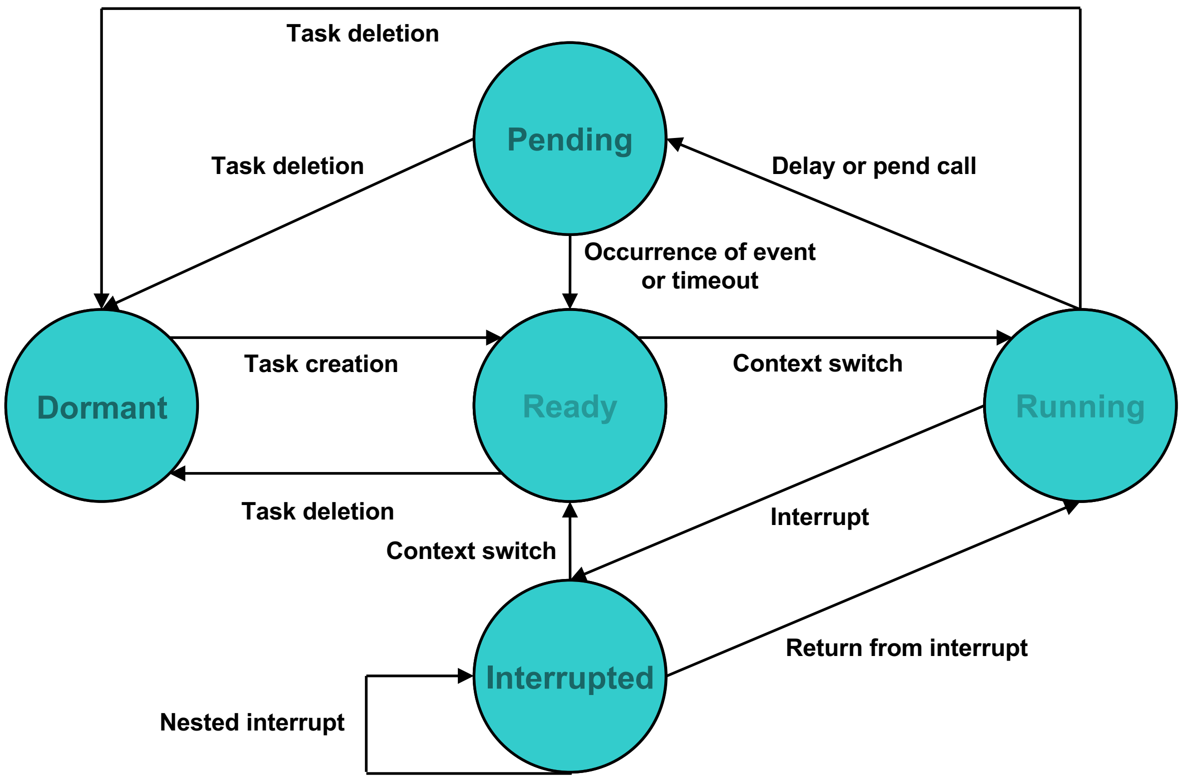


A Task Template

```
void  App_TaskExample (void *p_arg)
{
    Task initialization;
    for (;;) {
        Work toward task's goals;
        Wait for event;
    }
}
```

Tasks that do not occasionally give up the CPU will starve out lower-priority tasks

Task States


Task deletion
Pending
Task deletion
Delay or pend call
Occurrence of event
or timeout
Task creation
Context switch
Dormant
Ready
Running
Task deletion
Interrupt
Context switch
Interrupted
Return from interrupt
Nested interrupt

Creating a Task

- Information relating to each task must be passed to the kernel
- This information includes the following:
  - The starting address of the task
  - A reference to the task’s TCB
  - A reference to the task’s stack
  - The task’s priority
  - Optionally, an argument to pass to the task
- Application code provides this information to the kernel via a call to a task creation function

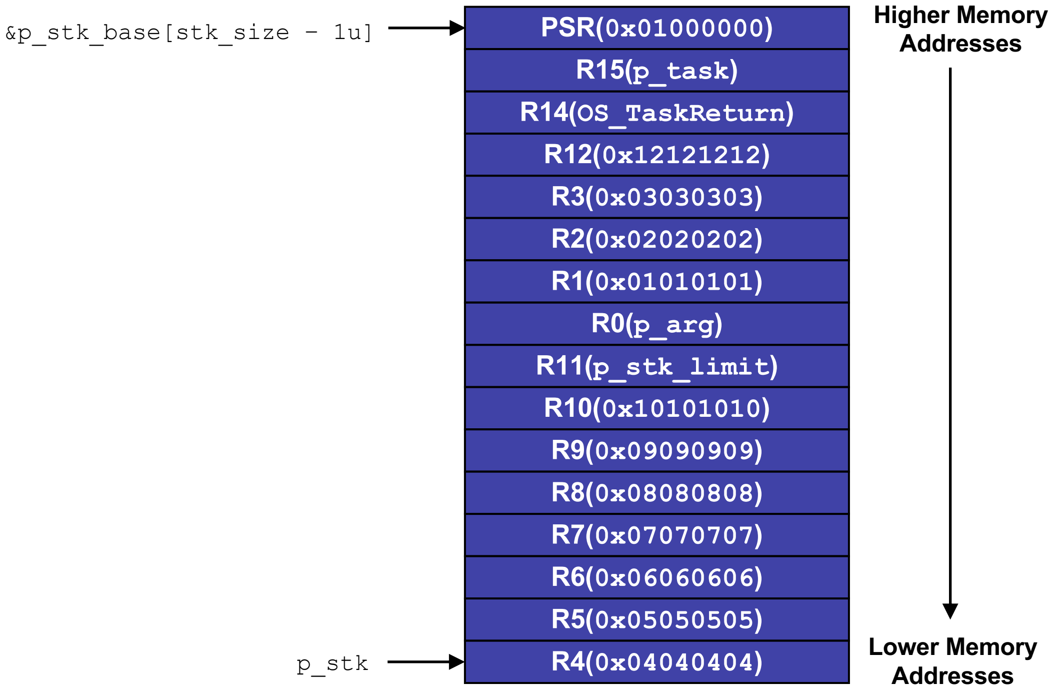
OSTaskCreate()

```
void  OSTaskCreate (OS_TCB          *p_tcb,
                    CPU_CHAR        *p_name,
                    OS_TASK_PTR      p_task,
                    void            *p_arg,
                    OS_PRIO          prio,
                    CPU_STK         *p_stk_base,
                    CPU_STK_SIZE     stk_limit,
                    CPU_STK_SIZE     stk_size,
                    OS_MSG_QTY       q_size,
                    OS_TICK          time_quanta,
                    void            *p_ext,
                    OS_OPT           opt,
                    RTOS_ERR        *p_err)
```

The Role of Stacks

- Compilers use a stack to implement subroutine (function) calls
- In Micrium OS Kernel, each task has a stack
  - Stacks must be declared in application code
  - CPU_STK AppTaskStartStk[*stack_size*]
- The kernel uses the task stacks to save context
  - Context is the state of the CPU at a given time, as defined by register values

A Newly Initialized Task Stack



Sizing a Task's Stack

- Stack requirements vary from task to task
  - A simple task may use 200 bytes of stack space, while a complex task can require 2 kBytes or more of space
- Each task's stack is used by both the compiler and the kernel
  - Function calls, context switches, and local variables all consume task stack space
- Application developers are responsible for sizing their stacks
- Tools are available but do not always provide a complete picture

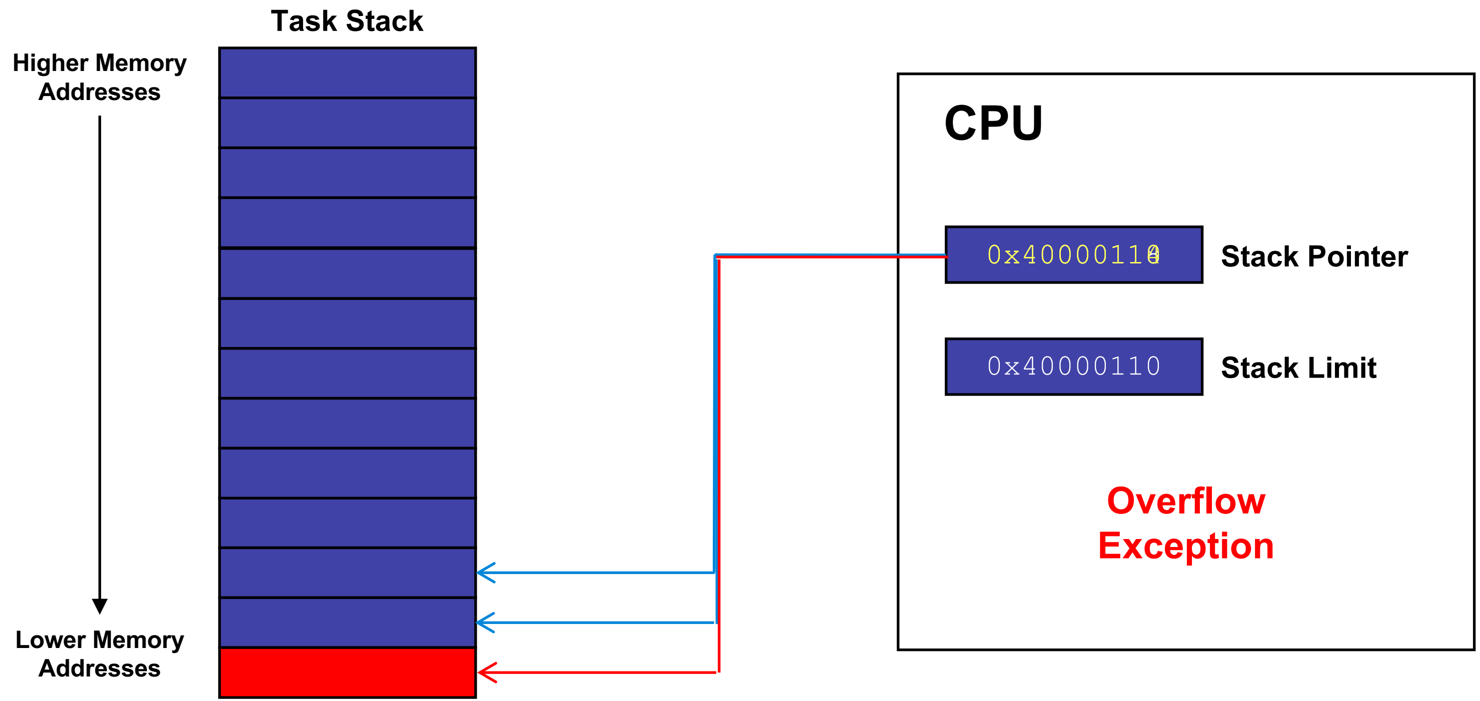
Stack Overflows

- Stack overflows are a problem that can occur when working with multitasking systems
  - Can be extremely difficult to debug
- There are several ways to prevent or detect stack overflows.
  - MMU/MPU
  - Stack Limit Register
  - Software-based Stack Limits
  - Software-based Stack Checking
  - Red Zone Stack Checking

MMU/MPU

- Memory Management Unit (MMU)
  - Typically seen in higher end processors
  - All memory references pass through the MMU
    - Tasks are each assigned their own virtual memory space
    - Prevents one task from affecting another task's memory space
  - Can also help prevent memory fragmentation
- Memory Protection Unit (MPU)
  - Has a subset of an MMU's features
  - Has the ability to define memory regions for specific tasks
  - Monitors memory access and throws faults when it detects an access violation
- Micrium OS Kernel does not currently support either of these features
  - MPU is on the roadmap

Stack Limit Register



Software-Based Stack Limit

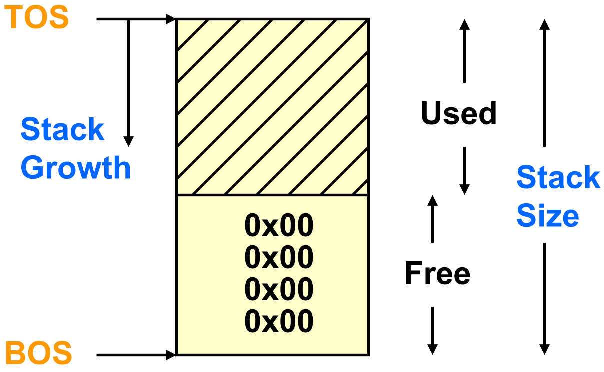
- Kernel compares stack pointer with stack limit each time a task is given control of the CPU
- Overflows are not detected immediately
  - The space between the stack limit and the end of the stack must be relatively large
- Few assumptions can be made about the extent of damage once an overflow is detected
- Micrium OS Kernel’s hook functions allow developers to implement software-based stack limits

Software-Based Stack Checking

- Micrium OS Kernel provides a stack-checking function, `OSTaskStkChk()`
- Stack checking is enabled through two `OSTaskCreate()` options
  - `OS_OPT_TASK_STK_CHK` and `OS_OPT_TASK_STK_CLR`
  - Goes through and checks the number of elements that are unused (set to 0)
- The statistics task can be configured to periodically check each task's stack usage

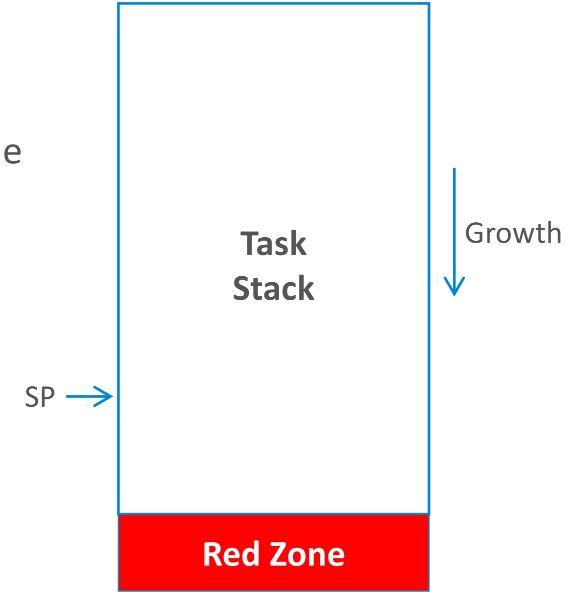
Software-Based Stack Checking Implementation

```
CPU_INT32U OSTaskStkChk (prio)
{
   #elements = 0;
   p_bos      = Point at bottom of task stack;
   p_tos      = Point at top    of task stack;
   while (*p_bos == 0x00 && p_bos != p_tos)
      #elements++;
      p_bos++;
   return (#elements);
}
```



Red-Zone Stack Checking

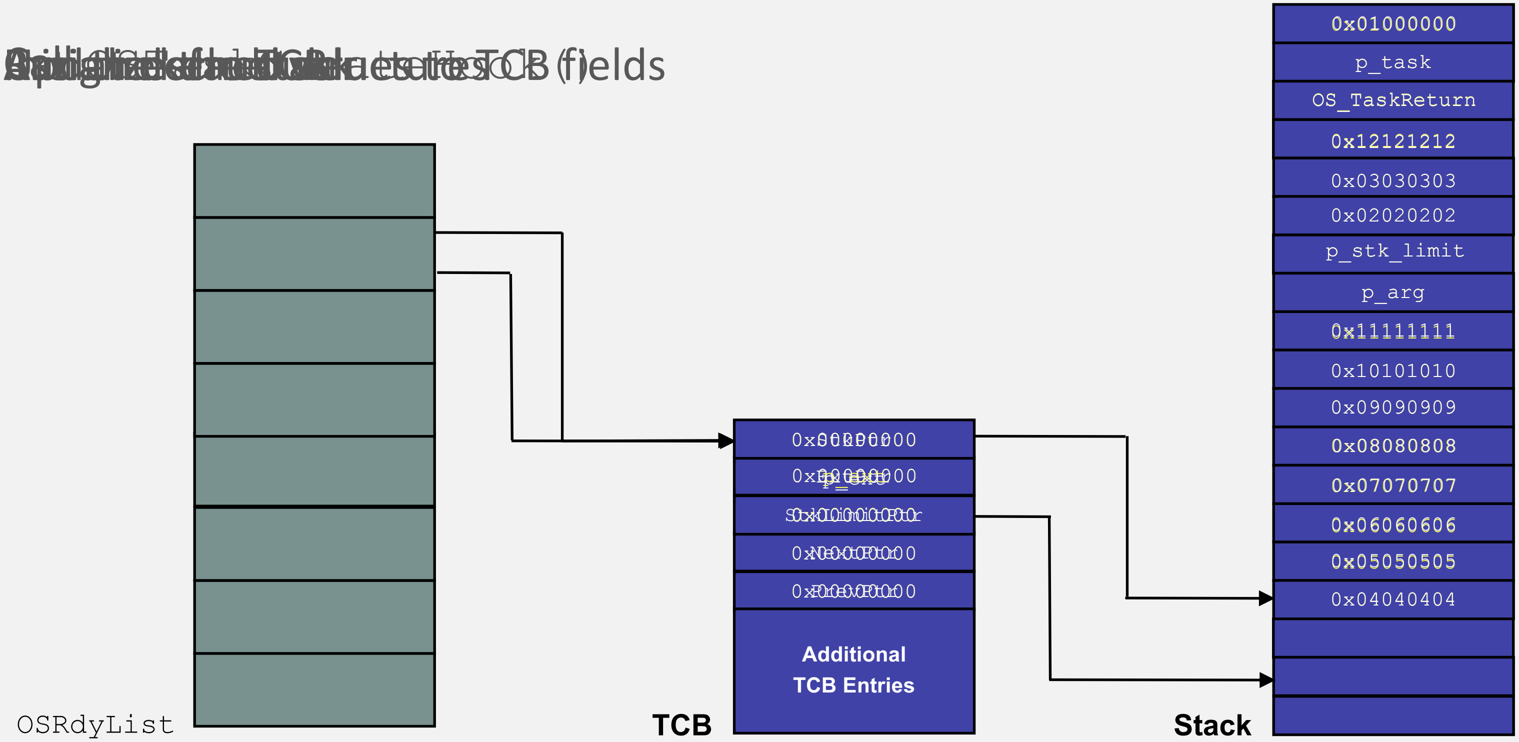
- Entries at end of stack filled with known value (0xABCD2345)
  - Number of entries specified through configuration constant, `OS_CFG_TASK_STK_REDZONE_DEPTH`
- If the red zone feature is enabled through `OS_CFG_TASK_STK_REDZONE_EN`, the kernel checks the red zone area upon each context switch



Task Control Blocks

- Within the kernel, a task control block (TCB) is used to keep track of each task
- Application code must declare TCBs for all of the tasks that it creates
  - OS_TCB App_TaskExampleTCB;
- The fields of a TCB should never be directly manipulated by application code
- A Micrium OS Kernel TCB has anywhere from 26 to 50 fields, depending on the kernel's configuration
- A TCB can include the following:
  - A pointer to the associated task's stack
  - The task's state
  - The task's priority
  - Data relating to event flags, message queues, and other kernel objects

OSTaskCreate() Implementation
Cortex-M



Deleting a Task

```
void  OSTaskDel (OS_TCB    *p_tcb,
                 RTOS_ERR *p_err);
```

- A deleted task is returned to the dormant state
- Even after the deletion is complete the task's code still exists in ROM
- The task's stack and TCB can be reused after deletion

Changing the Priority of a Task

```
void  OSTaskChangePrio (OS_TCB     *p_tcb,
                        OS_PRIO     prio_new,
                        RTOS_ERR   *p_err);
```

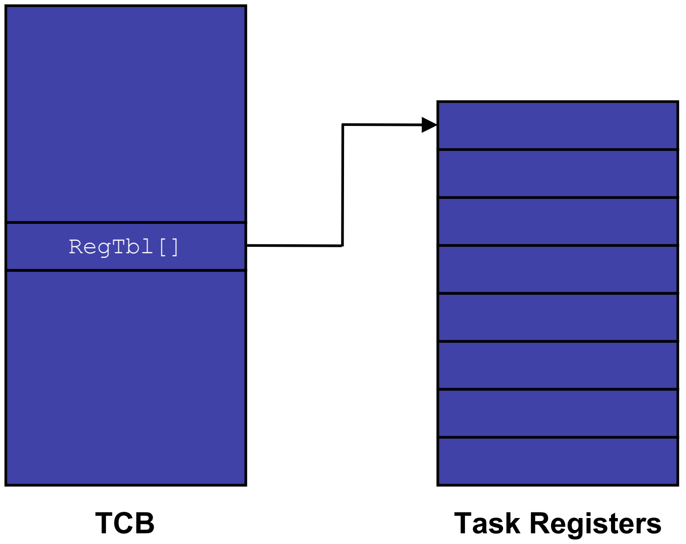
- A task can change its own priority or the priority of another task
- Any priority that can be assigned to a new task can also be passed to `OSTaskChangePrio()`

Valid Priorities

- The total number of priorities in a Micrium OS Kernel application is established by the configuration constant `OS_CFG_PRIO_MAX` (see **os_cfg.h**)
- The lowest priority (`OS_CFG_PRIO_MAX – 1`) is automatically assigned to the idle task
  - Assuming the idle task is enabled
- The highest priority (0) is given to your most important task

Task Registers

- A set of task registers is optionally included in each TCB
- The registers are simply array entries
  - The size of the array of registers is determined by the configuration constant `OS_CFG_TASK_REG_TBL_SIZE`
- Task-specific data can be stored in the registers
  - Allows you to create a “global” on a per-task basis

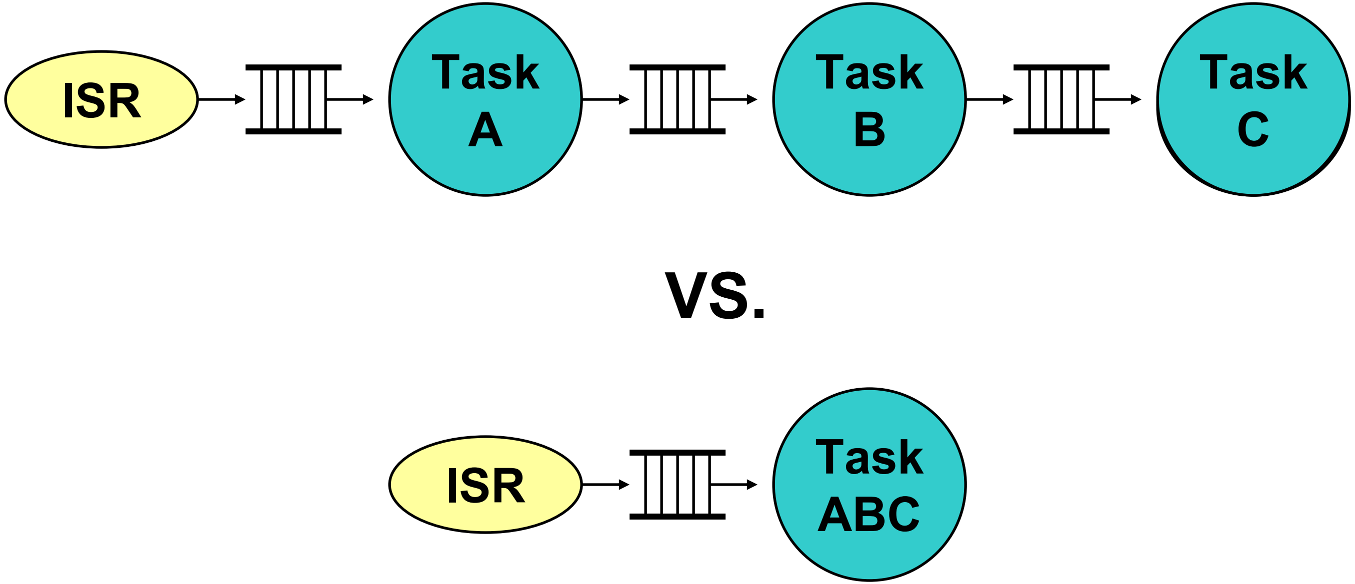


Partitioning an Application

- The job of dividing an application into tasks is rarely trivial
- A poorly partitioned application may fail to meet performance requirements
- To determine what portions of an application warrant separate tasks, developers should look for activities that can execute in parallel
  - For example, an LCD driver can update a display while file system code waits for data
- Problems can be created both by excessively large tasks and tasks that are too small
  - Large, complex tasks can behave like foreground/background systems
- Excessively small tasks increase the amount of time spent context switching
  - Generally, developers should make sure that the execution time of each task is significantly larger than twice the context-switch time
    - Task-Execution Time should be **much greater than** 2 x Context-Switch Time

Partitioning an Application

Excessive inter-task communication often reflects a poor design



Assigning Task Priorities

- Application developers must assign a priority to each of their tasks
- If priorities are assigned arbitrarily, the benefits of using a real-time kernel may not be realized
- When multiple tasks have important deadlines, assigning priorities can be particularly difficult
- Its important that developers remember to consider the priorities of any system tasks when assigning priorities to their application tasks

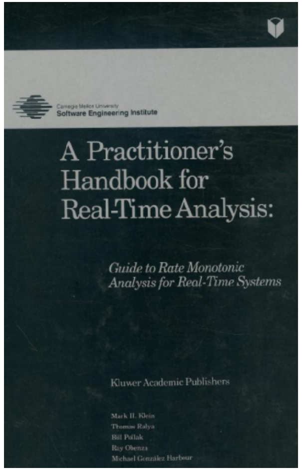
Rate Monotonic Scheduling (RMS)

- With RMS, task priorities are set according to a simple rule
  - The tasks with the highest frequencies are given the highest priorities
- RMS is optimal
  - There is no better scheme for assigning priorities
- Developers can use RMS to determine how many of the tasks in a given application will actually be able to meet their deadlines
- There are a few key assumptions that underlie RMS:
  - Each task runs periodically
  - A given task always completes its work within a fixed amount of time
  - Tasks do not interact
- In order to assign priorities according to RMS, developers must calculate the execution time of each of their tasks

RMS References

A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems, by Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzales Harbour

Multiple articles covering RMS are available from Embedded.com



Section Summary - Tasks

- A task must be created to be managed by Micrium OS Kernel
  - You need to assign a priority, a stack and a TCB
- Micrium OS Kernel has up to 4 internal tasks
  - The Idle task is always the lowest priority (if present)
- Tasks are infinite loops
  - A task must wait for an ‘Event to Occur’
- Configuration allows you to turn ON/OFF services
  - Reduces Code Space and RAM

Lab #2

Scheduling and Context Switching

The Scheduler

- Control of the CPU is passed from one task to another based on the actions of Micrium OS Kernel's scheduler
- The scheduler is called by many of the kernel's API functions
- Micrium OS Kernel has a priority-based scheduler but also supports round-robin scheduling
- The scheduler is not a separate task or ISR, but rather a function that is called

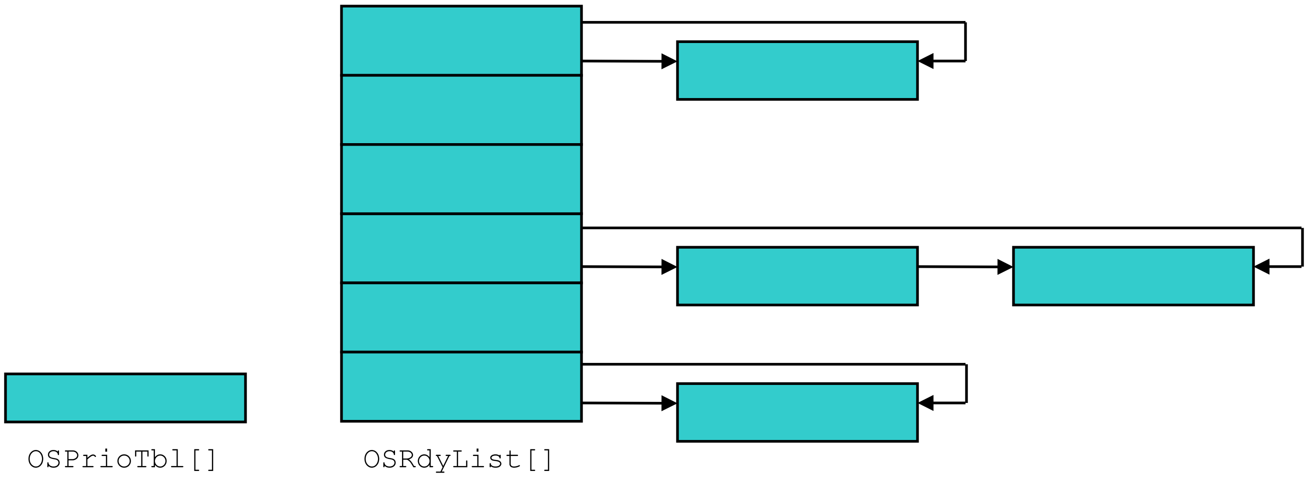
Micrium OS Kernel's Scheduling Algorithm

Once tasks have been assigned priorities, the Micrium OS Kernel can schedule the CPU amongst multiple tasks … identify the single, high-priority task

Task A	**Task B**	**Task C**	**Task D**	**Task E**
State: Ready	**State: Ready**	**State: Pending**	**State: Ready**	**State: Pending**
Priority: 5	**Priority: 3**		**Priority: 8**	

Scheduling Data Structures

- The kernel uses two data structures to make scheduling decisions

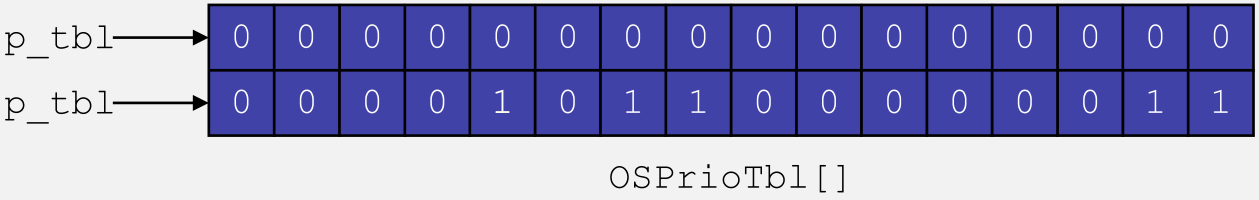


OSPrioTbl[]

- Each priority level from 0 to (`OS_CFG_PRIO_MAX - 1`) is represented by a bit in `OSPrioTbl[]`
- The data type of an OSPrioTbl[] entry is `CPU_DATA`
  - This type is defined in the CPU module
- A set bit indicates that at least one task at the corresponding priority is in the Ready state

OSPrioTbl[]

```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;

    prio  = (OS_PRIO)0;
    p_tbl = &OSPrioTbl[0];
    while (*p_tbl == (CPU_DATA)0) {
        prio += sizeof(CPU_DATA) * 8u;
        p_tbl++;
    }
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);
    return (prio);
}
```



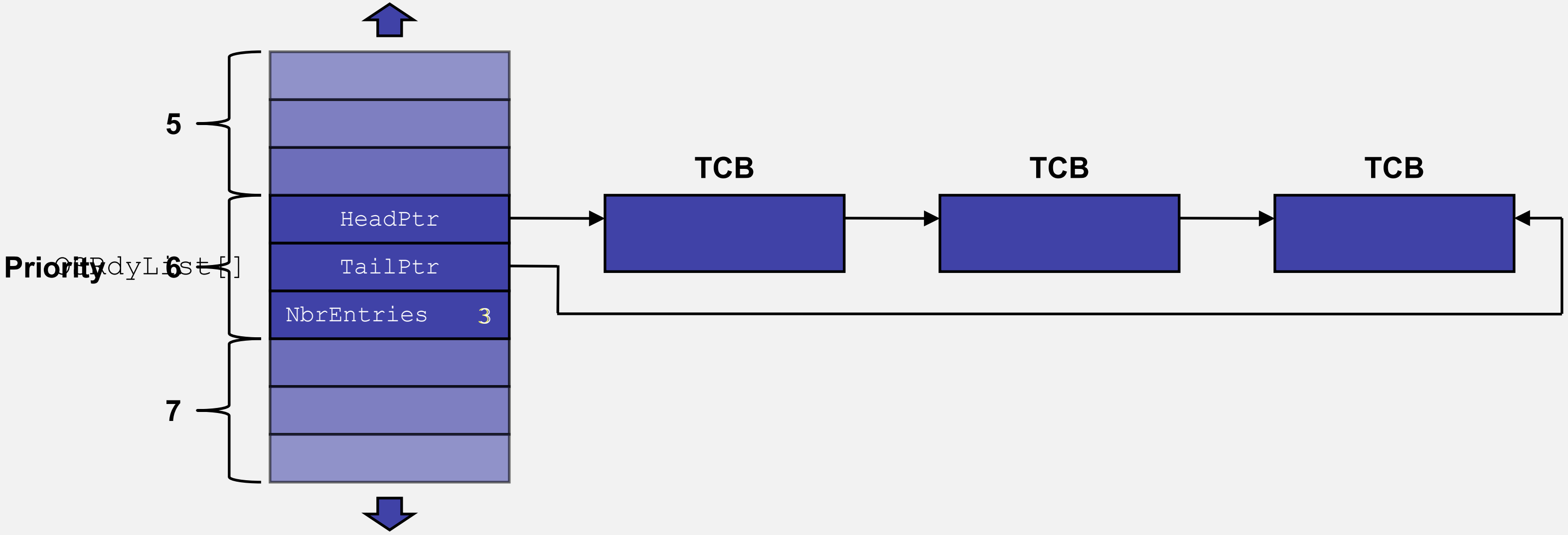
prio = 26

CPU_CntLeadZeros()

- CPU_CntLeadZeros() is declared by the CPU module
- There are a couple of ways to optimize CPU_CntLeadZeros()
  - Assembly language
  - Look-up table
- In most CPU ports, CPU_CntLeadZeros() is optimized

OSRdyList[]

… **TailPtr** … priority that is associated with their **OSRdyList[]** entry



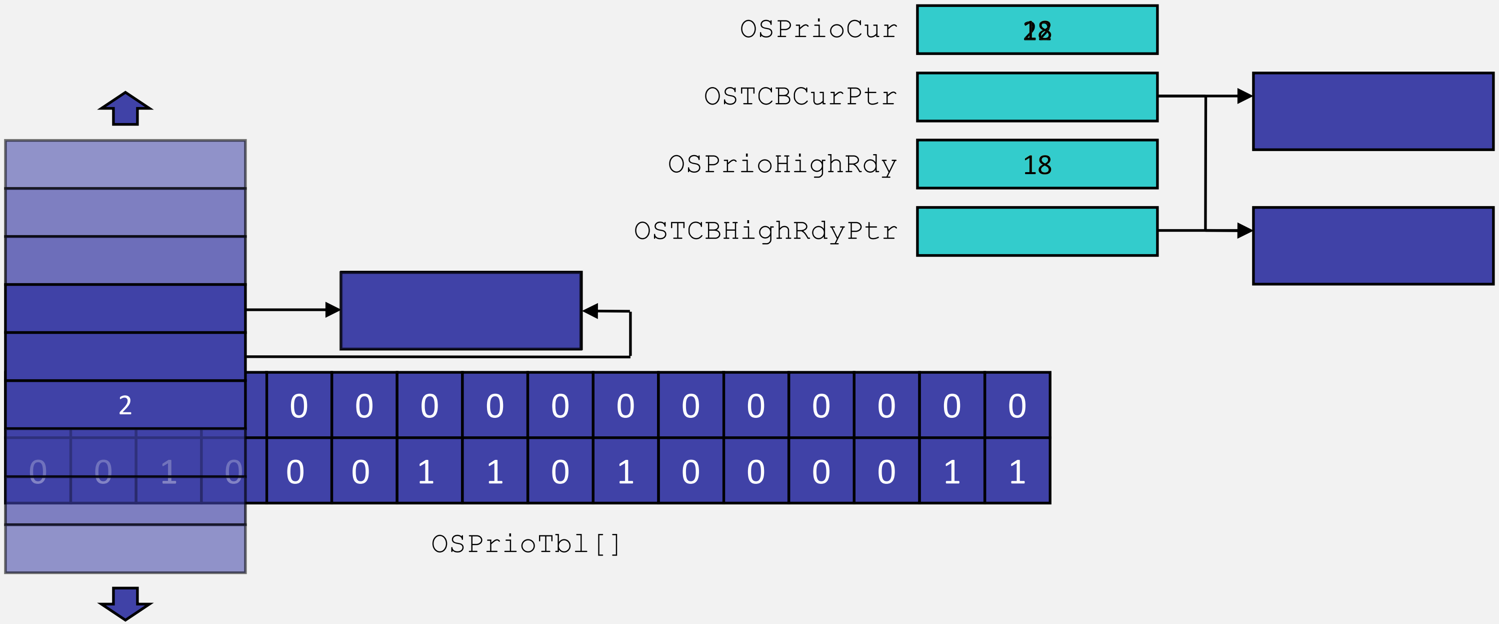
Updating OSPrioTbl[] and OSRdyList[]

- `OSTaskCreate()` updates `OSPrioTbl[]` and `OSRdyList[]`
  - Tasks enter the Ready state when they are created
- Numerous other API functions update the data structures
  - These functions cause tasks to transition to either the Ready or Pending state
- The kernel's scheduler runs after the data structures have been updated

OSSched()

… is obtained by

```
OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
```

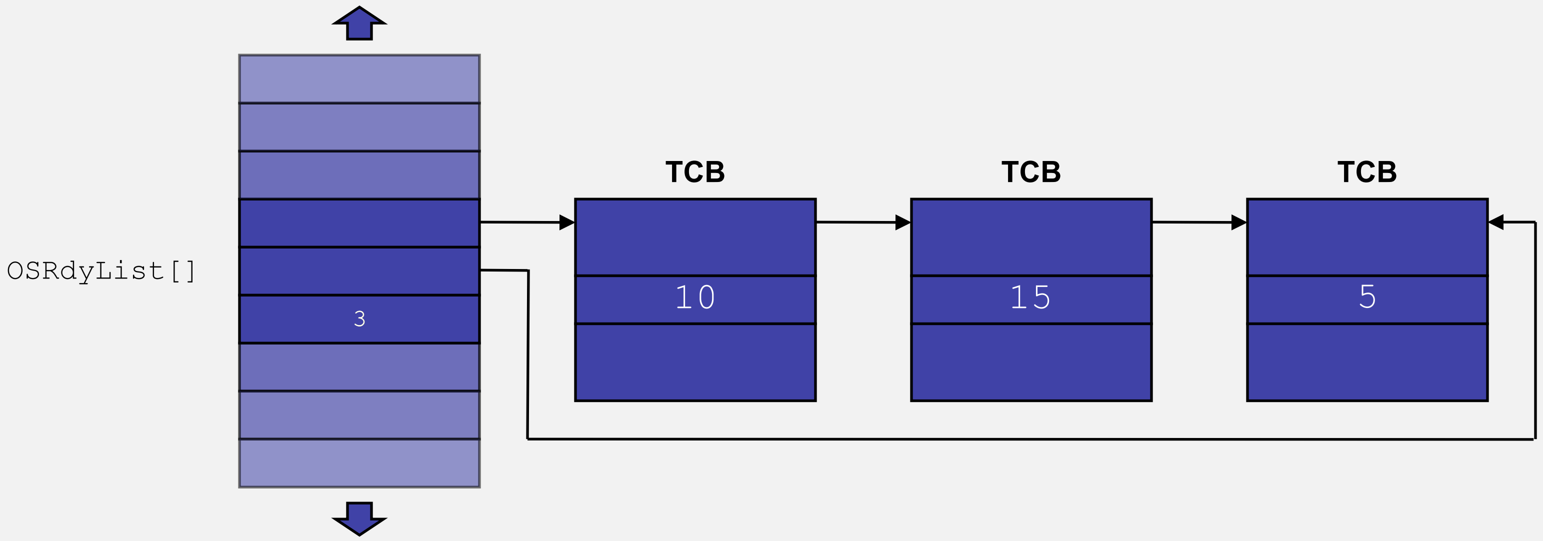


Round-Robin Scheduling

- Round-robin scheduling must be enabled via a call to `OSSchedRoundRobinCfg()`
  - Additionally, the configuration constant `OS_CFG_SCHED_ROUND_ROBIN_EN` must be defined as 1
- Time quanta can be assigned through `OSTaskCreate()`
  - `OSTaskTimeQuantaSet()` allows for changes after task creation
- Round-robin scheduling takes place alongside priority-based scheduling

Round-Robin Scheduling

`OS_SchedRoundRobin()` periodically decrements the leading TCB's `TimeQuantaCtr` field



Delay Functions

```
void  OSTimeDly (OS_TICK     dly,
                 OS_OPT      opt,
                 RTOS_ERR   *p_err);

void  OSTimeDlyHMSM (CPU_INT16U  hours,
                     CPU_INT16U  minutes,
                     CPU_INT16U  seconds,
                     CPU_INT32U  milli,
                     OS_OPT      opt,
                     RTOS_ERR   *p_err);
```

Example Use of Delay Function

```
void  App_TaskKbd (void *p_arg)
{
    RTOS_ERR err;
    Initialize keyboard;

    while (1) {
        Scan keyboard;
        OSTimeDlyHMSM(0u, 0u, 0u, 100u,
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
    }
}
```

Dynamic Tick Rate

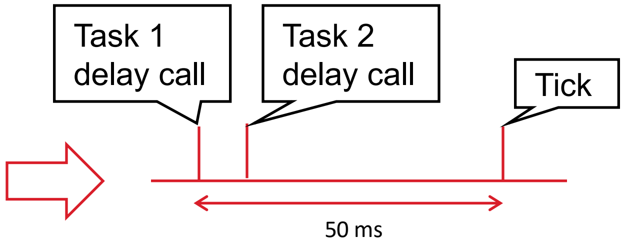
- Recent addition to Micrium's kernels
- Tick rate is adjusted to match period of highest-frequency delayed task
  - Example: Two tasks using time delays

Task 1

```
OSTimeDlyHMSM(0,0,
              0,50,…);
```

Task 2

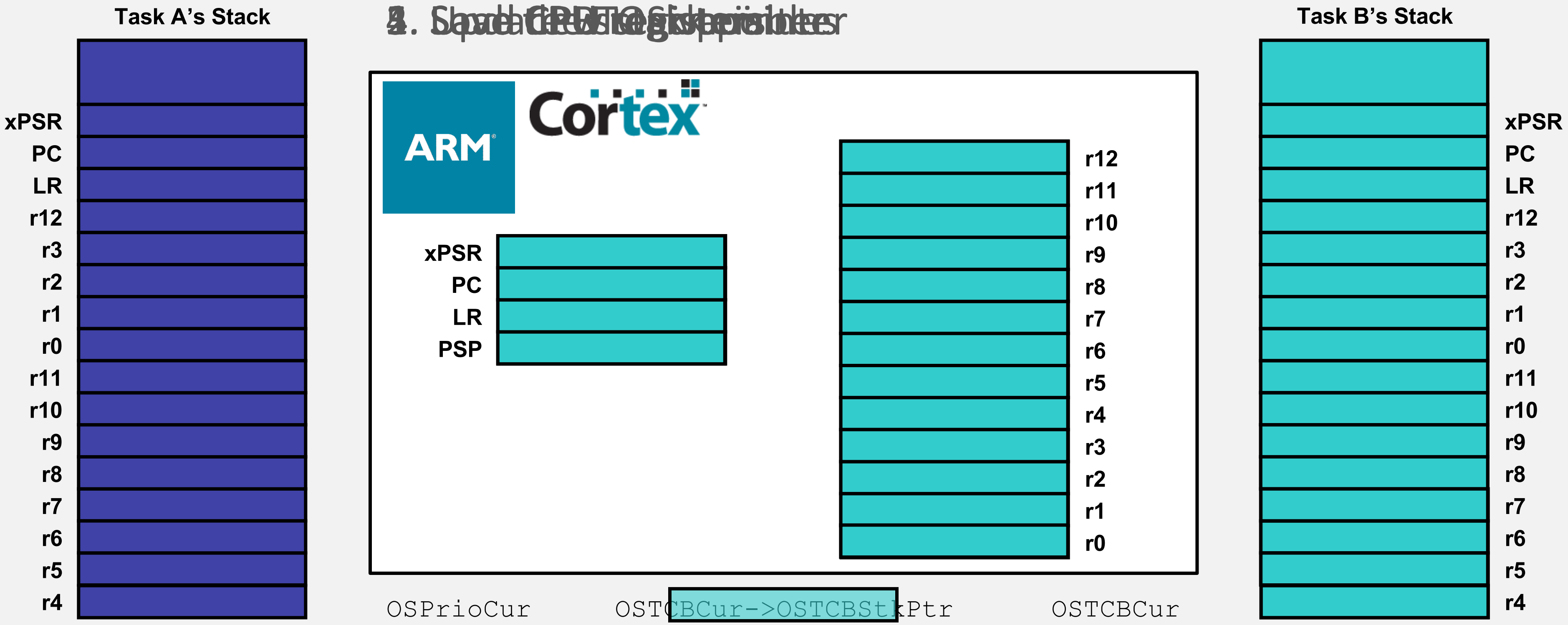
```
OSTimeDlyHMSM(0,0,
              0,100,…);
```



Context Switch

- A context switch is the process via which control of the CPU is passed from one task to another
- “Context” is state information associated with a task
  - CPU registers
- What, exactly, needs to be done during a context switch varies from architecture to architecture

Context Switch



Interrupts and Exceptions

Context Switch from ISRs

- In a preemptive kernel, interrupts can result in context switches

```
ExampleISR:
    Save CPU registers;
    OSIntEnter();
    App_ISR();
    OSIntExit();
    Restore CPU registers;
    Return from interrupt;
```

```
void  App_ISR (void)
{
    /* Clear interrupt */
    /* Signal task     */
}
```

Determine whether a context switch is needed

Interrupts on the Cortex-M

- Vectored interrupt controller
- Each peripheral handler invokes a generic Micrium OS Kernel handler, passing the generic function an ID
- Generic handler takes care of kernel-specific operations

```
/**
*********************************************************************************************************
*                                        CPU_IntHandlerDispatcher()
*
* @brief    Basic ISR dispatcher that gets called for every interrupt.
*
* @note     (1) This dispatcher calls the registered handler for a specific interrupt.
*********************************************************************************************************
*/

void  CPU_IntHandlerDispatcher (void)
{
    CPU_FNCT_VOID  handler;
    CPU_INT_ID     id;
    CPU_SR_ALLOC();

                                                                /* This is a kernel aware ISR.           */
    CPU_CRITICAL_ENTER();
    OSIntEnter();
    CPU_CRITICAL_EXIT();

                                                                /* Get active interrupt ID.              */
    id = CPU_REG_NVIC_ICSR & CPU_MSK_NVIC_ICSR_VECT_ACTIVE;

                                                                /* Should always be true.                */
    if (id < CPU_INT_NBR_OF_INT) {
        handler = CPU_IntVectTbl[id].HandlerPtr;

        if (handler != (CPU_FNCT_VOID)DEF_NULL) {
                                                                /* If registered, call user specified handler. */
            handler();
        } else {
                                                                /* Otherwise, call default empty handler. */
            CPU_IntEmptyHandler();
        }
    }

    OSIntExit();
}
```

Non-Kernel Aware vs Kernel Aware ISRs

On some CPUs, it is possible to remove kernel overhead from select ISRs

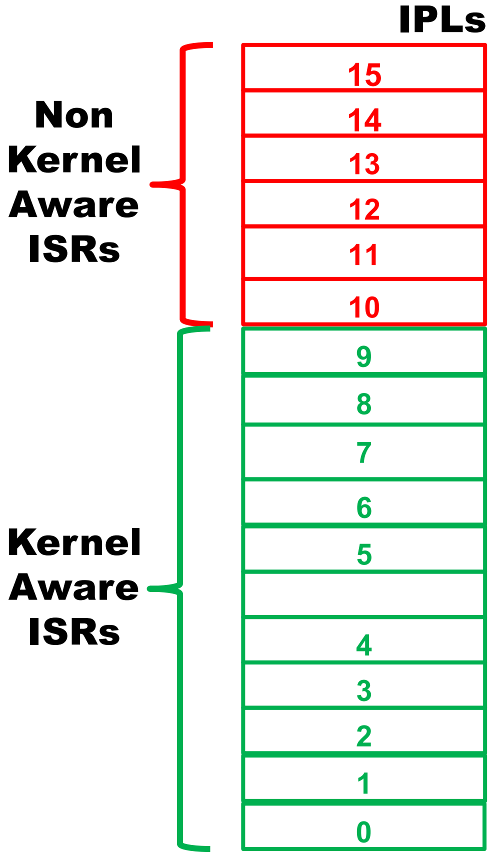
- http://www.electronicproducts.com/Software/System/Microseconds_matter_reducing_interrupt_latency_in_industrial_control_systems.aspx?terms=interrupt%20latency

Non-Kernel Aware ISRs (NKA)

- Higher priority than Kernel Aware ISRs
- Cannot make kernel API calls
- Kernel critical sections don't affect NKAs

Kernel Aware ISRs

- Have lower priority than non-kernel aware
- Kernel critical sections done within the range of kernel aware ISR range



Section Summary – ISRs

- Micrium OS Kernel always runs the highest-priority task ready-to-run
  - Interrupts are **more important** than tasks
  - An ISR **can** cause a more important to run **after** the ISR returns
- You can use the **Tick ISR** as an **example** on how to implement ISRs under Micrium OS Kernel
- Non-Kernel Aware ISRs **cannot** make kernel API calls

Lab #3

Synchronization

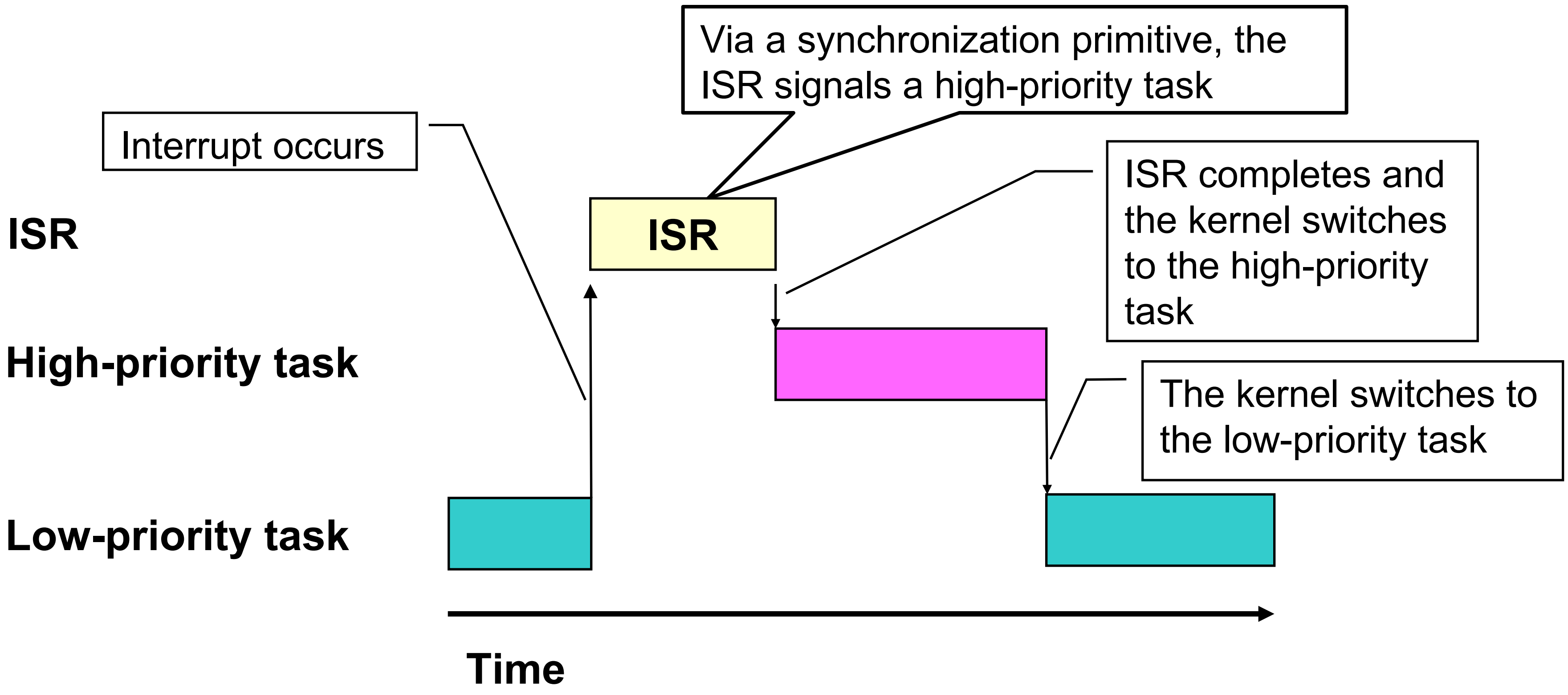
Task/Task Interaction

- The tasks in a Micrium OS Kernel application are not necessarily self-contained
- Tasks may need to interact with each other
  - A typical kernel will provide mechanisms that facilitate such interaction
    - Semaphores
    - Mutexes
    - Event Flags
    - Message Queues

Task/ISR Interaction

- Most applications must manage a collection of peripheral devices
  - UART, I2C, SPI, USB, etc.
- The interrupt service routines (ISRs) associated with a system's peripheral devices should be kept brief
- In applications that incorporate a real-time kernel, ISRs can use synchronization primitives to signal tasks

Synchronizing a Task to an ISR



Problems with Lengthy ISRs

- On many architectures, a long ISR can significantly increase interrupt latency
- Excessively large stacks may be needed in order to support lengthy ISRs
- Debugging interrupt handlers can be difficult
- Many kernel functions cannot be invoked by ISRs

Signaling in Foreground / Background Systems

- Developers with foreground/background systems may need to write signaling code themselves

```
while (1) {
    ADC_Read();
    SPI_Read();
    USB_Packet();
    LCD_Update();
    Audio_Decode();
    File_Write();
}
```

```
void  USB_ISR (void) {
    USB_Packet++;
    Clear interrupt;
}
```

```
void  USB_Packet (void) {
    while (USB_Packet == 0){
        ;
    }
    Process packet;
}
```

Semaphores

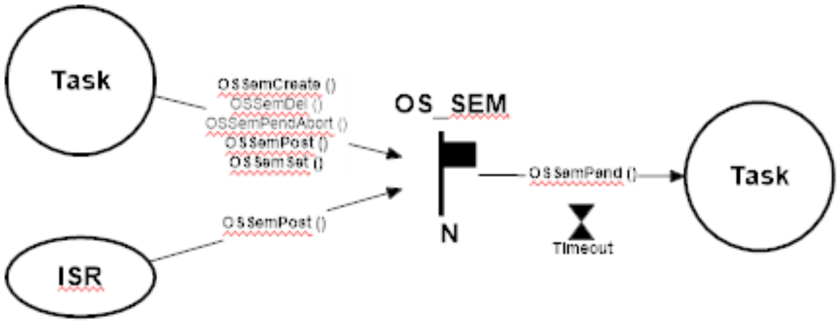
- Using a semaphore, a task can synchronize to another task or to an ISR
- Semaphores are based on counters
- A semaphore can be classified as either binary or counting

- Pend
  - While the semaphore's counter has a value of zero, allow other tasks to run
    - One of the parameters accepted by Micrium OS Kernel's pend functions is a timeout value that indicates how long the calling task is willing to wait

- Post
  - Increment the semaphore's counter
    - If a task is waiting on the semaphore, that task will be placed in the Ready state when the post operation occurs

Semaphore API

```
void  OSSemCreate (OS_SEM       *p_sem,
                   CPU_CHAR     *p_name,
                   OS_SEM_CTR    cnt,
                   RTOS_ERR     *p_err);

OS_SEM_CTR  OSSemPend (OS_SEM      *p_sem,
                       OS_TICK      timeout,
                       OS_OPT       opt,
                       CPU_TS      *p_ts,
                       RTOS_ERR    *p_err);

OS_SEM_CTR  OSSemPost (OS_SEM     *p_sem,
                       OS_OPT      opt,
                       RTOS_ERR   *p_err);
```

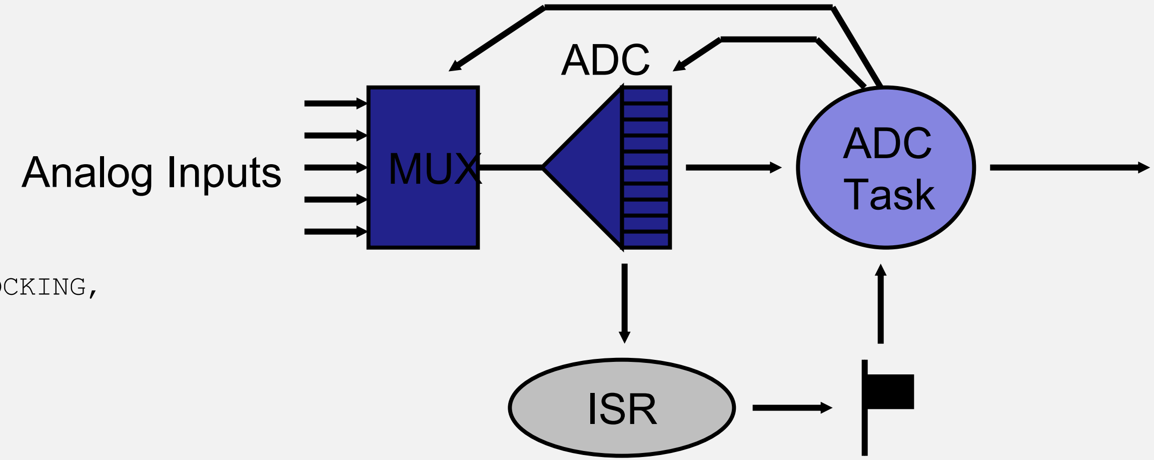


Semaphore Example

```
OS_SEM  App_SemADC;

/* Initialization Code */

OSSemCreate((OS_SEM   *)&App_SemADC,
            (CPU_CHAR *)"ADC Sem",
            (OS_SEM_CTR)0,
            (OS_ERR   *)&err);
```

```
void  App_ISRADC (void)
{
    Clear interrupt;
    OSSemPost((OS_SEM *)&App_SemADC,
              (OS_OPT  )OS_OPT_POST_1,
              (OS_ERR *)&err);
}
```

```
void  App_TaskADC (void *p_arg)
{
    Perform initializations;
    while (1) {
        Start conversion;
        OSSemPend((OS_SEM *)&App_SemADC,
                  (OS_TICK )0,
                  (OS_OPT  )OS_OPT_PEND_BLOCKING,
                  (CPU_TS *)&ts,
                  (OS_ERR *)&err);
        Process converted value;
    }
}
```



Counting Semaphores

- Micrium OS Kernel's semaphores are counting semaphores
  - The kernel does not limit the semaphore state to 1 or 0
- It is **not** possible to set an upper limit for the count value of the semaphore
  - Because of this, in the previous example the potential for the semaphore's value to rise above 1 could actually be detrimental
- If a binary semaphore would be more advantageous than a counting semaphore, there are other kernel objects that can be used instead
  - Event Flags

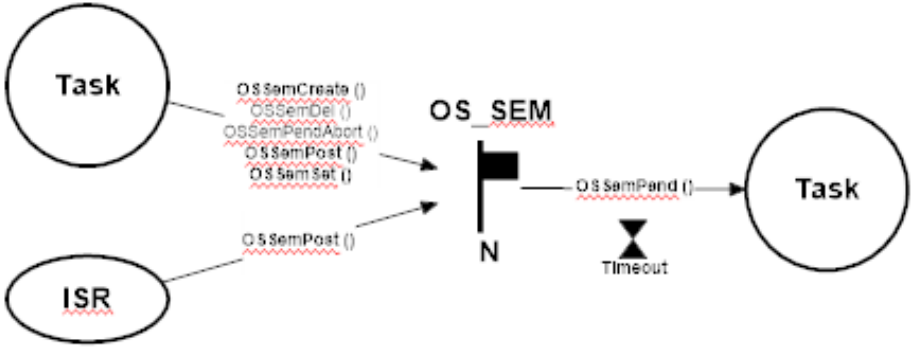
Task Semaphores

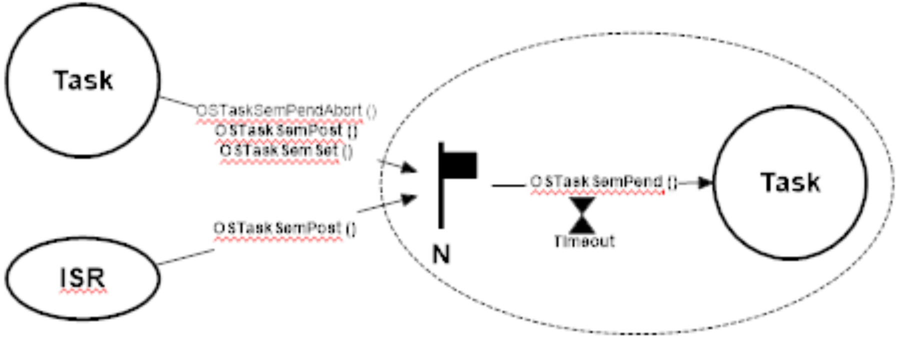
- Built-in to all Micrium OS Kernel tasks
  - Uses the `OS_TCB` to store the semaphore data rather than an `OS_SEM` object
- Any task or ISR can call `OS_TaskSemPost()`
  - Calling task or ISR just needs the TCB pointer
- Only the task referenced by the TCB can Pend
  - No need to pass the TCB when calling `OS_TaskSemPend()`

1. vim

```
OS_TCB  App_SomeTaskTCB;

void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_SEM_CTR  ctr;

                                                                 /*  Block until another task signals this task.        */
    ctr = OSTaskSemPend( 1000,                                   /*  Wait for a maximum of 1000 OS Ticks.               */
                         OS_OPT_PEND_BLOCKING,                   /*  Task will block.                                   */
                         DEF_NULL,                               /*  No timeout specified.                              */
                        &err);
    if (err.Code == RTOS_ERR_NONE) {
        // Task has been signaled by App_SomeOtherTask.
    } else {
        // Handle error on task semaphore pend.
    }
}


void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_SEM_CTR  ctr;

                                                                 /* Post the pending App_SomeTask task.                 */
    ctr = OSTaskSemPost(&App_SomeTaskTCB,                        /* Pointer to App_SomeTask's TCB.                      */
                         OS_OPT_POST_NONE,                       /* No special option.                                  */
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        // Handle error on task semaphore post.
    }
}
```

Semaphores vs. Task Semaphores



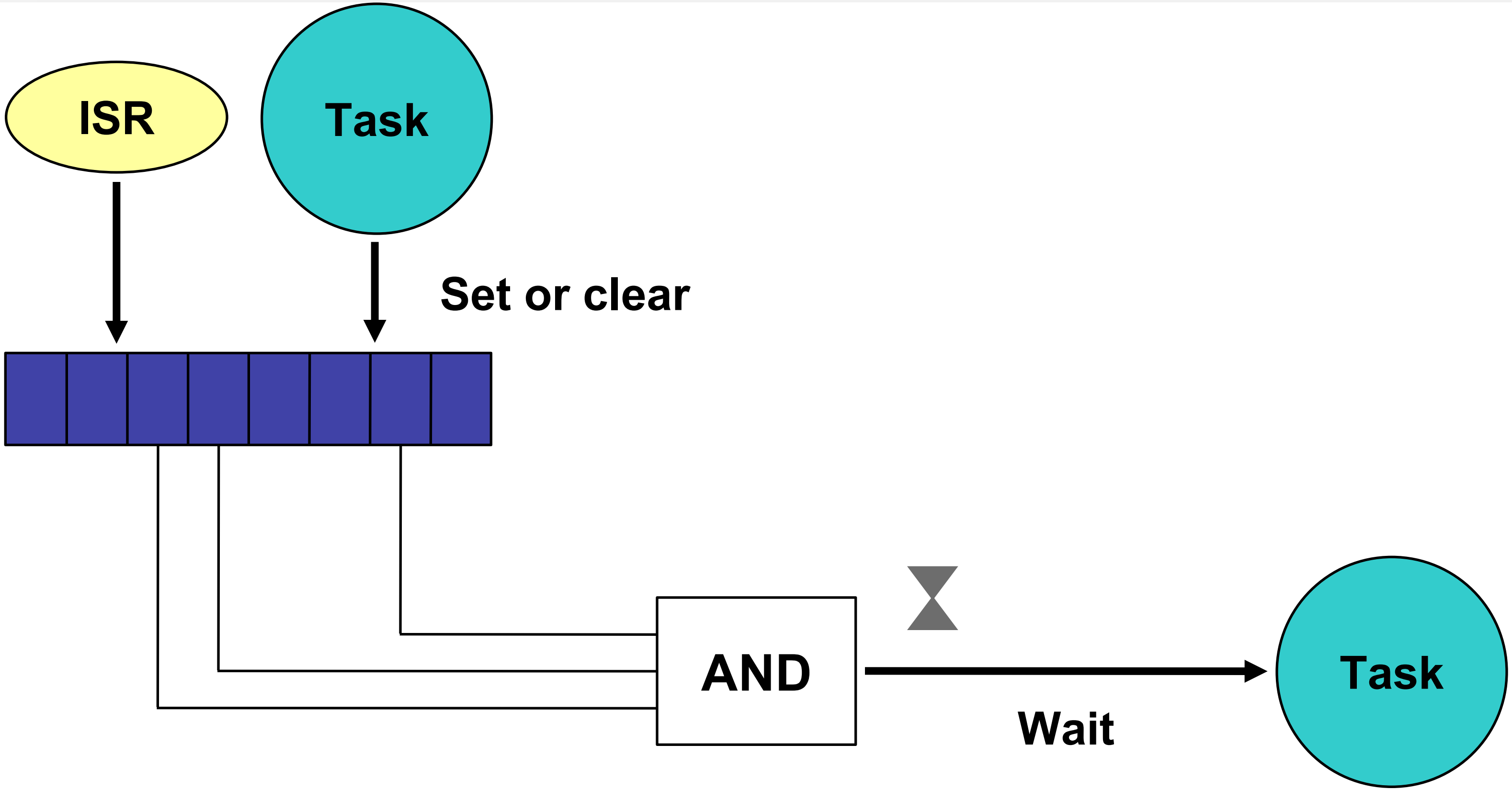


- Very common to signal a task
- No need for wait list
- Less overhead - faster

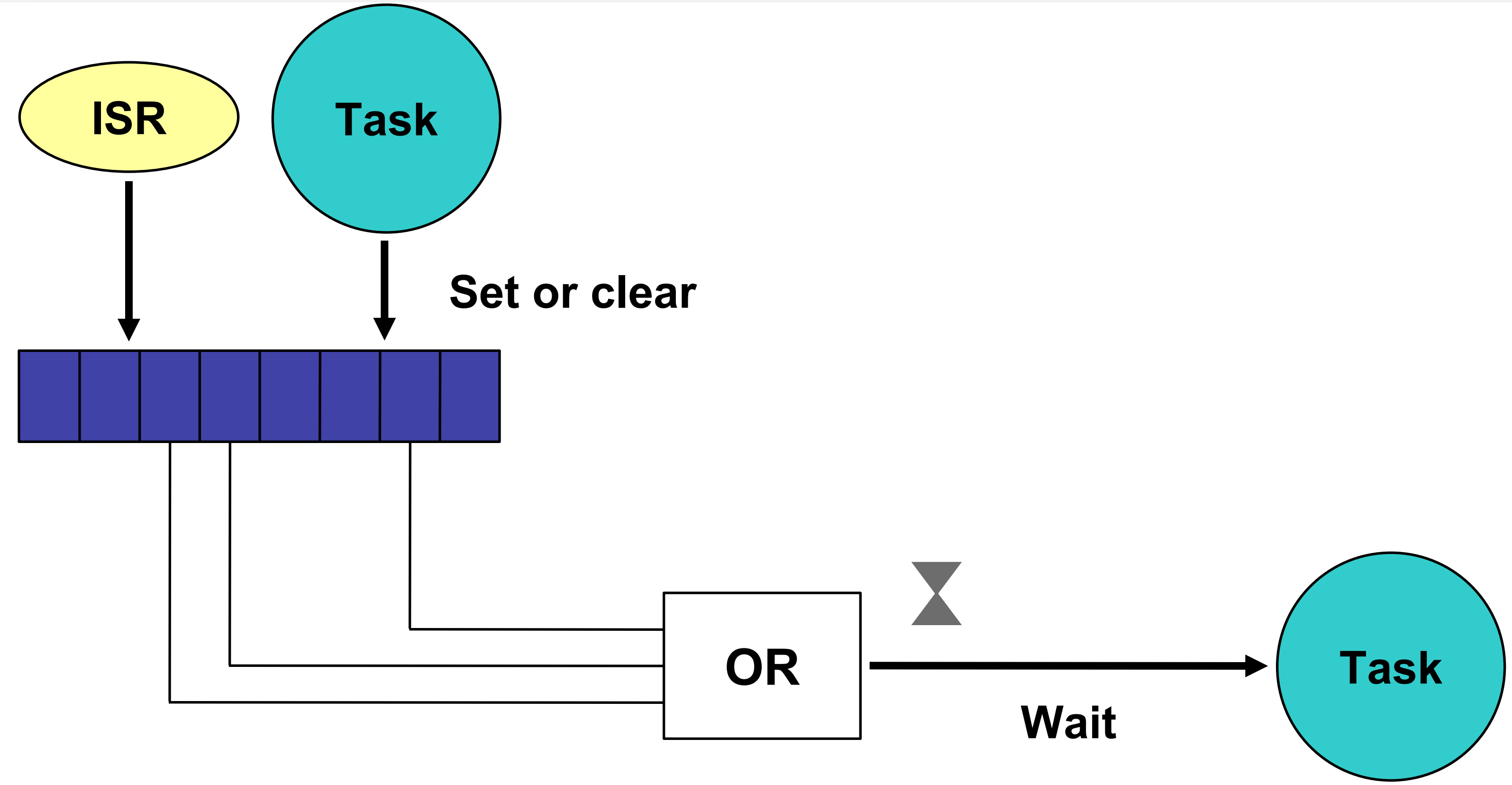
Event Flags

- Using event flags, a task can easily wait for multiple events to take place
- A single 8-, 16-, or 32-bit variable, contained in a structure known as an event flag group, represents a collection of events
  - Each bit in the variable corresponds to a single event
  - Application code determines whether a set or cleared bit indicates the occurrence of an event

Conjunctive Synchronization

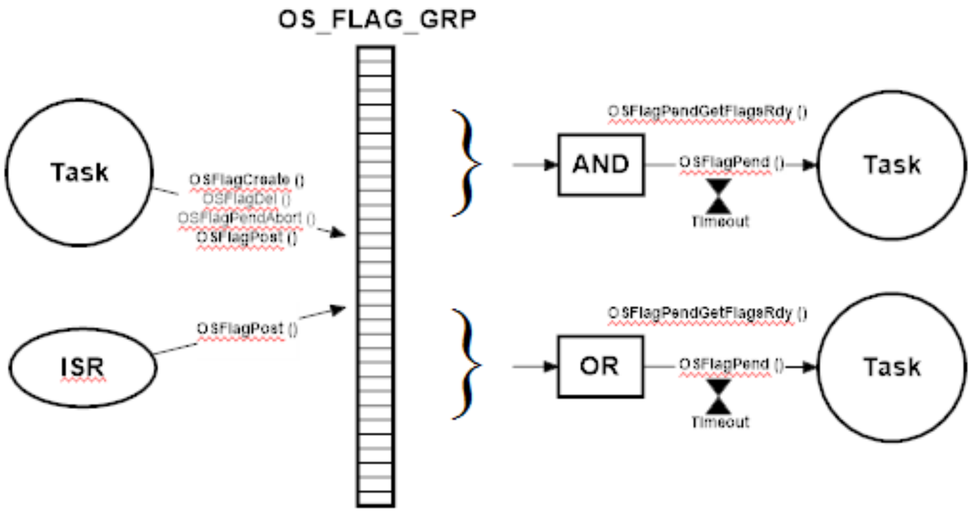


Disjunctive Synchronization

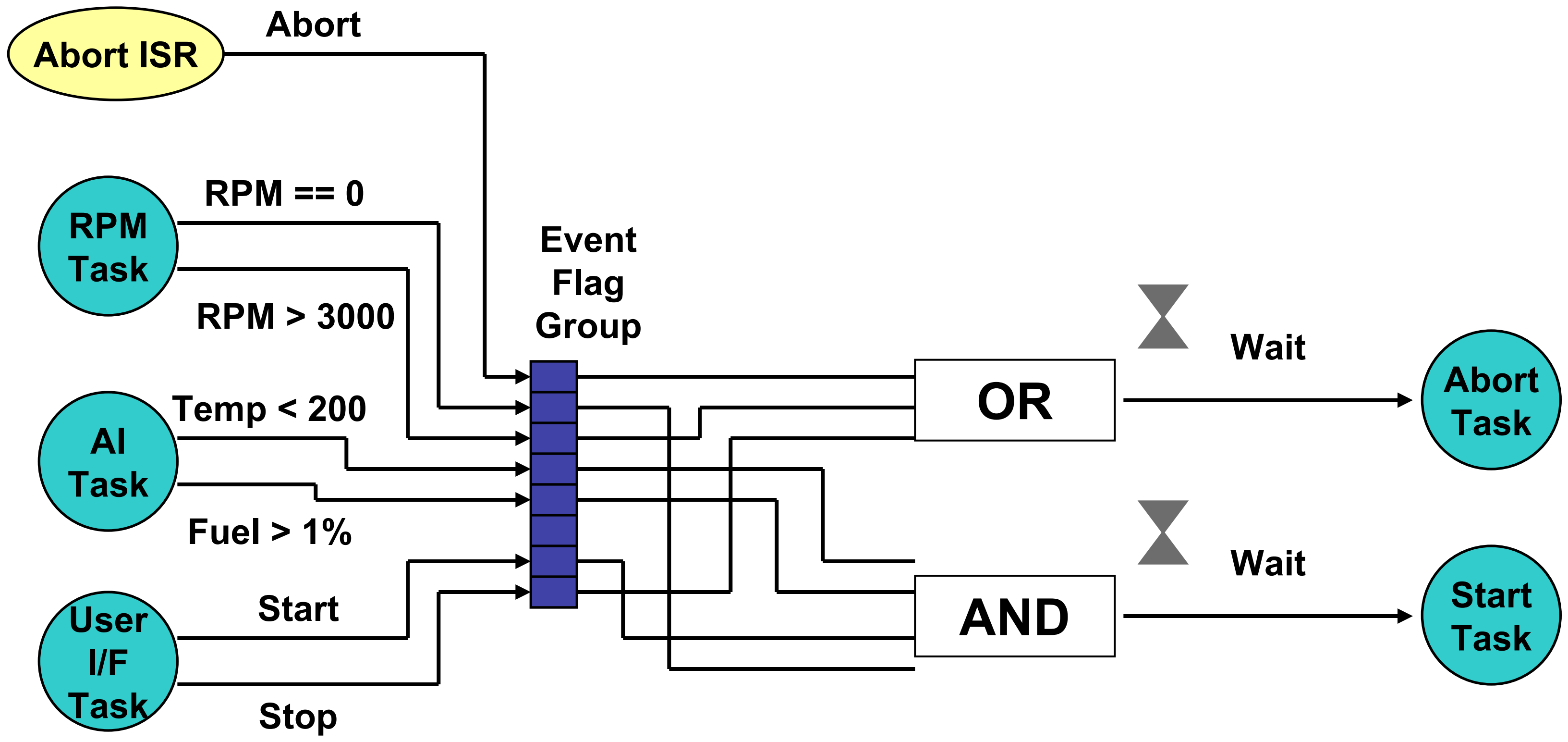


Event Flags API

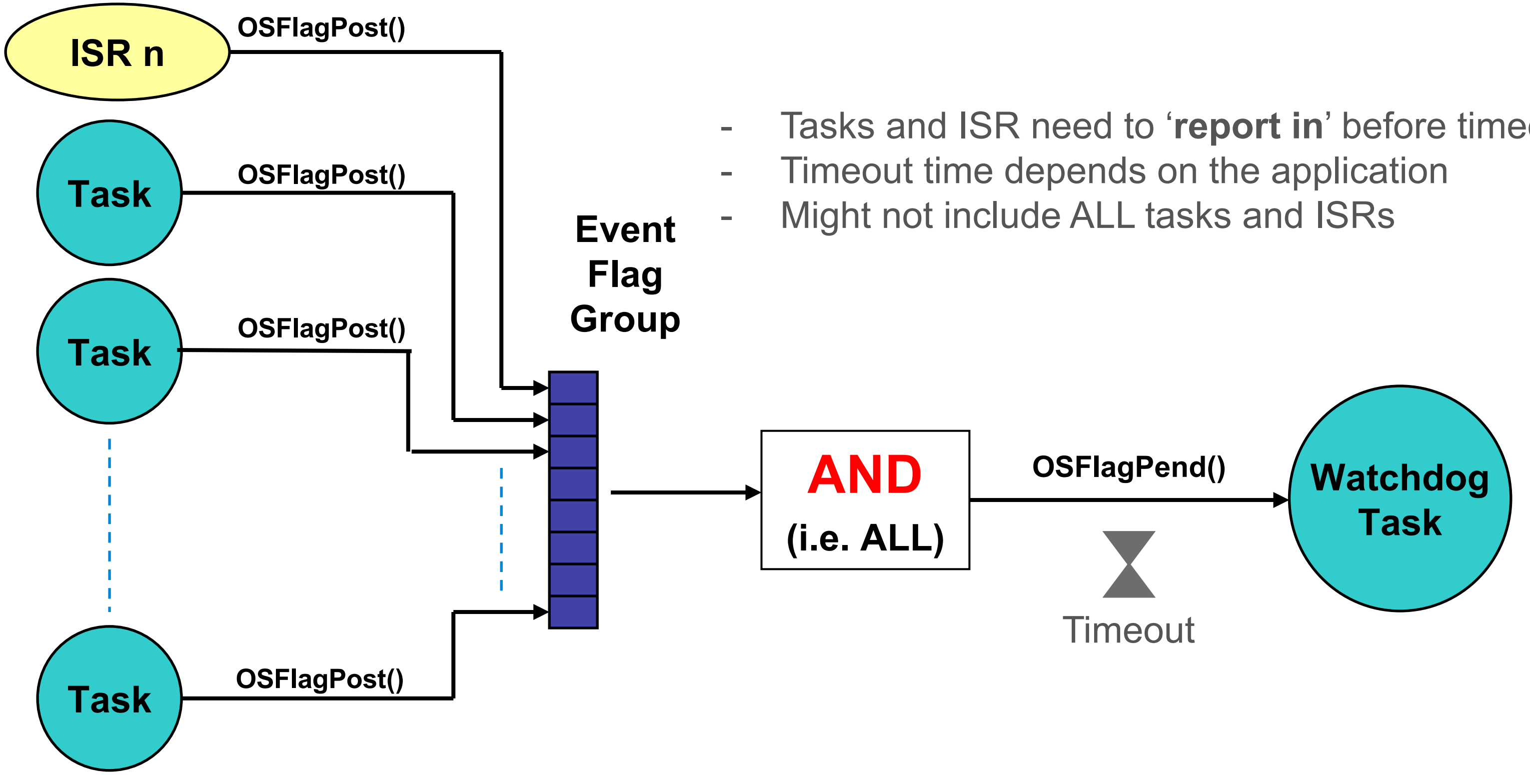
```
void  OSFlagCreate (OS_FLAG_GRP   *p_grp,
                    CPU_CHAR      *p_name,
                    OS_FLAGS       flags,
                    RTOS_ERR      *p_err);

OS_FLAGS  OSFlagPend (OS_FLAG_GRP *p_grp,
                      OS_FLAGS     flags,
                      OS_TICK      timeout,
                      OS_OPT       opt,
                      CPU_TS      *p_ts,
                      RTOS_ERR    *p_err);

OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_OPT        opt,
                      RTOS_ERR     *p_err);
```



Event Flags – Engine Control Example



Event Flags – Watchdog Example



- Tasks and ISR need to '**report in**' before timeout
- Timeout time depends on the application
- Might not include ALL tasks and ISRs

Section Summary - Synchronization

- Semaphores and Event Flags are used to **signal** a task that an event occurred.
  - Semaphores are counting (i.e. 0..N)
  - Event Flags are binary (i.e. 0 or 1)
- **Only tasks** can **wait** for a signal
  - ISRs cannot wait on a semaphore or event flags
- Signaling is so common that Micrium OS Kernel's tasks have a **built-in** semaphore
- **Event Flags** allows tasks to wait for any or **multiple events** to occur

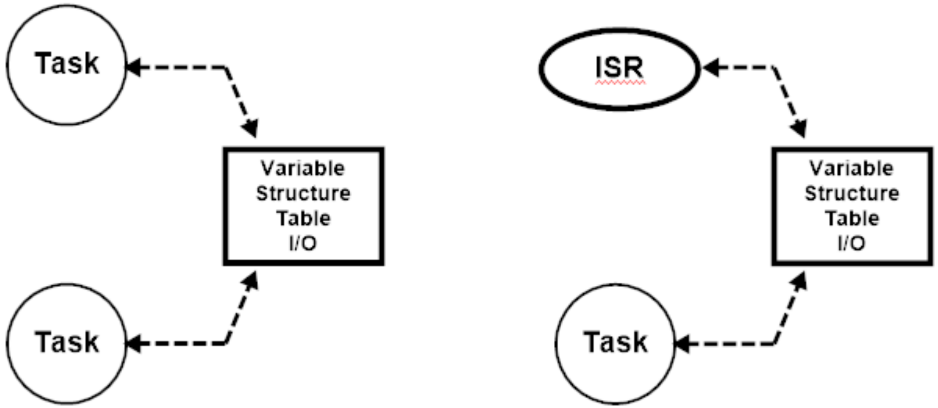
Review

Lab #4

Mutual Exclusion

Shared Resources

- A global variable or data structure that is used by multiple tasks is considered a shared resource
  - Variables accessed by both tasks and ISRs are also shared resources
- Oftentimes, peripheral devices are shared resources
  - For example, an Ethernet controller that is accessed by multiple tasks



Shared Resource Example

```
typedef  struct  app_pressure {
    INT32U  A;
    INT32U  B;
} APP_PRESSURE;

APP_PRESSURE  Pressure;
```

`Pressure` is a shared resource

```
void  App_TaskDisplay (void *p_arg)
{
    RTOS_ERR err;

    while (1) {
        if (Pressure.A > Pressure.B) {
            Open valve;
        }
        OSTimeDlyHMSM(0u, 0u, 0u, 100u,
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
    }
}
```

```
void  App_ISRSensor (void)
{
    Pressure.A = SensorA;
    Pressure.B = SensorB;
    Clear interrupt;
}
```

Problems Created by Shared Resources

- While one task is manipulating a shared resource, other tasks should not be able to gain access to that resource
  - Remember, tasks can always be preempted by interrupts or higher tasks
- If this rule is not enforced, tasks might read corrupt data
  - The kernel provides mechanisms to protect resources but it is up to the application developer to protect the shared resource
- Bugs resulting from the corruption of shared resources can be highly frustrating
  - Typically erratic errors due to corruption from context switching while accessing the resource
  - Extremely hard to track down

What needs to be protected?

- Read-modify-write passages are notorious sources of data corruption

C
`App_FileCnt++;`

Assembly		
LDR.N	r0,	??DataTable2
LDR	r0,	[r0]
ADDS	r0,	r0, #1
LDR.N	r1,	??DataTable2
STR	r0,	[r1]

Short pieces of code that simply read or write global variables can cause problems too

Disabling Interrupts

- Interrupts are disabled before each shared resource is accessed and then re-enabled afterwards
- Micrium OS CPU provides macro functions for disabling and enabling interrupts
  - `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`
  - Macro definitions are part of the CPU port

Disabling Interrupts

```
void  App_TaskExample (void *p_arg)
{
    CPU_SR_ALLOC();

    while (1) {
        CPU_CRITICAL_ENTER();
        Access shared resource;
        CPU_CRITICAL_EXIT();
    }
}
```

Disabling Interrupts

- Micrium OS Kernel, like other kernels, uses this method to protect its own global variables
  - Application code can disable interrupts for short periods of time without negatively impacting interrupt latency
- Below is an excerpt from `OSTaskCreate()`

1. vim

```
                                                /* -------------- ADD TASK TO READY LIST -------------- */
    CPU_CRITICAL_ENTER();
    OS_PrioInsert(p_tcb->Prio);
    OS_RdyListInsertTail(p_tcb);

#if (OS_CFG_DBG_EN == DEF_ENABLED)
    OS_TaskDbgListAdd(p_tcb);
#endif

    OSTaskQty++;                                /* Increment the #tasks counter                         */

    if (OSRunning != OS_STATE_OS_RUNNING) {     /* Return if multitasking has not started               */
        CPU_CRITICAL_EXIT();
        return;
    }

    CPU_CRITICAL_EXIT();

    OSSched();
```

Locking the Scheduler

- Locking the scheduler is another means of protecting shared resources
- This technique cannot be used for variables that are accessed by interrupt handlers
- Application code that locks the scheduler for extended periods of time might experience performance problems
- Micrium OS Kernel provides scheduler-locking functions via two API calls
  - `OSSchedLock()`
  - `OSSchedUnlock()`
- Locking the scheduler is the same idea as making the current task the only task in the system.

Schedule Locking Example

```
void  App_TaskExample (void *p_arg)
{
    RTOS_ERR  err;

    while (1) {
        OSSchedLock(&err);
        Access shared resource;
        OSSchedUnlock(&err);
    }
}
```

Interrupts can occur while resource is being accessed

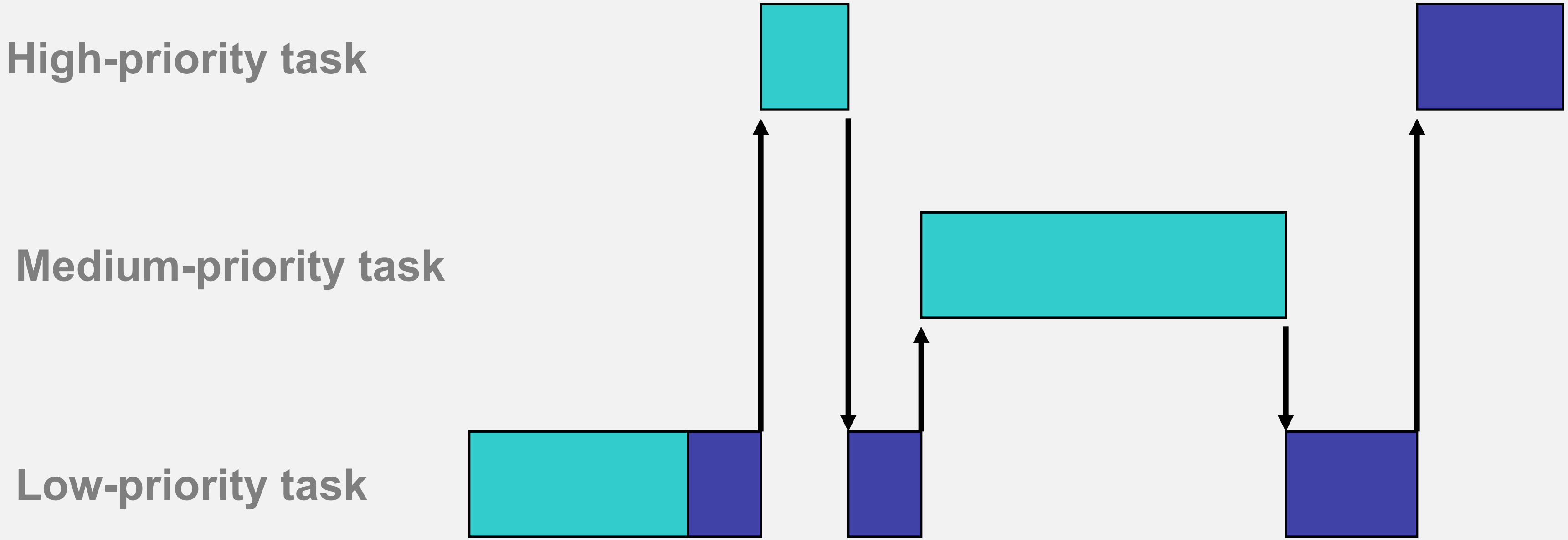
Semaphores

- In addition to being well suited for synchronization, semaphores can be used for protecting shared resources
- Semaphores were originally designed for this purpose
- The same semaphore API functions are used for both synchronization and resource protection
- When a semaphore is used for protecting a shared resource, tasks must pend on the semaphore before accessing the resource
  - This method cannot be used for resources that are accessed by ISRs
- If the value of the semaphore's counter is 0, the resource is unavailable
- Interrupts and context switches can occur while shared resources are being accessed

Priority Inversions

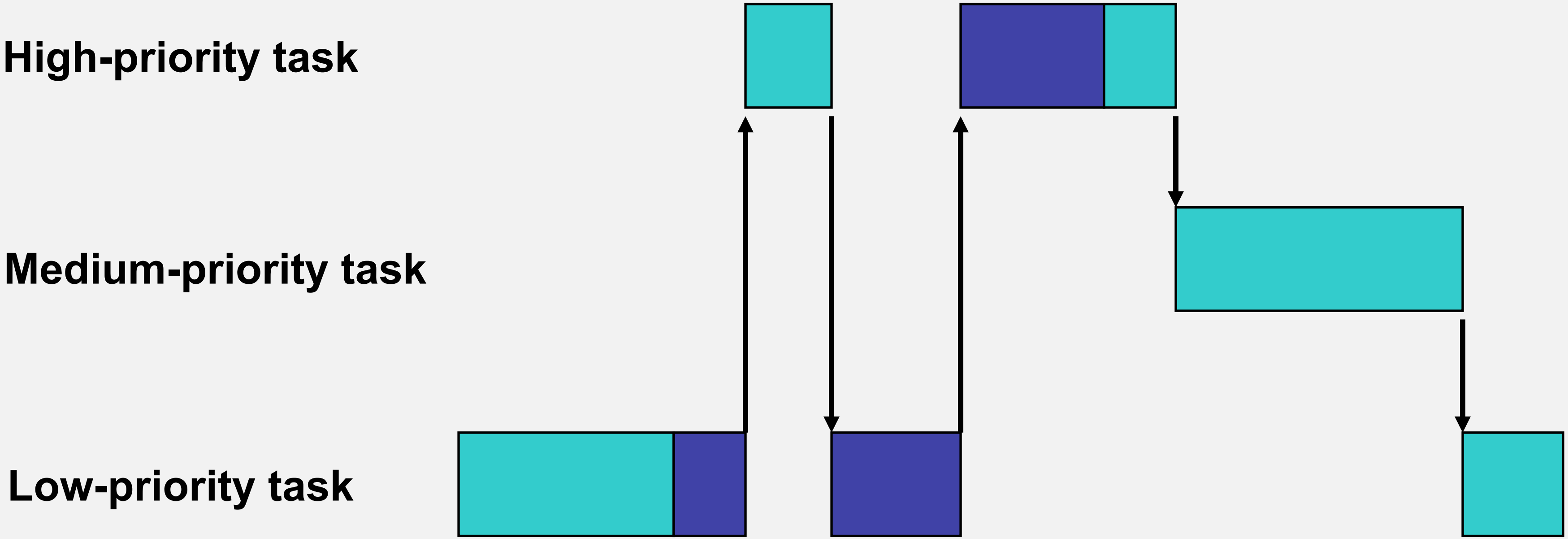
- There is a well documented problem associated with the use of semaphores for resource protection
- The problem, known as priority inversion, can arise when a low priority task is in the midst of accessing a resource that is needed by a higher priority task

Priority Inversion



Priority Inheritance

[illegible]

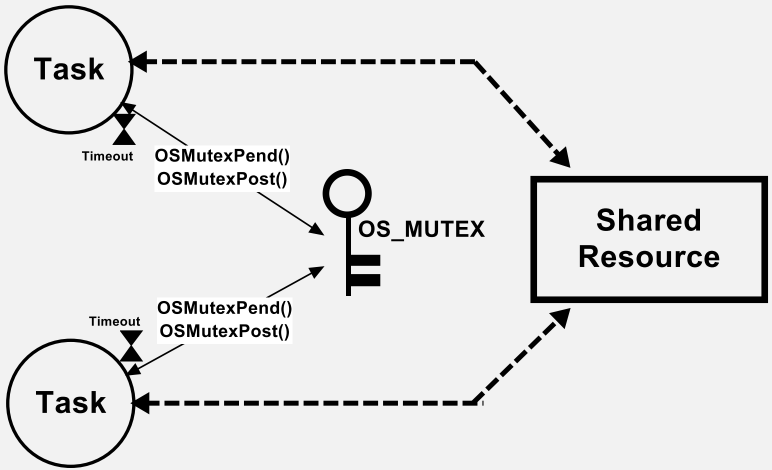


Mutexes

- A Mutex is another mechanism for protecting shared resources
- In Micrium OS Kernel, Mutexes provide built-in protection from priority inversion
  - Priority inheritance
- Unlike a semaphore, a Micrium OS Kernel Mutex does not incorporate a counter
  - The mutex is either available or in use

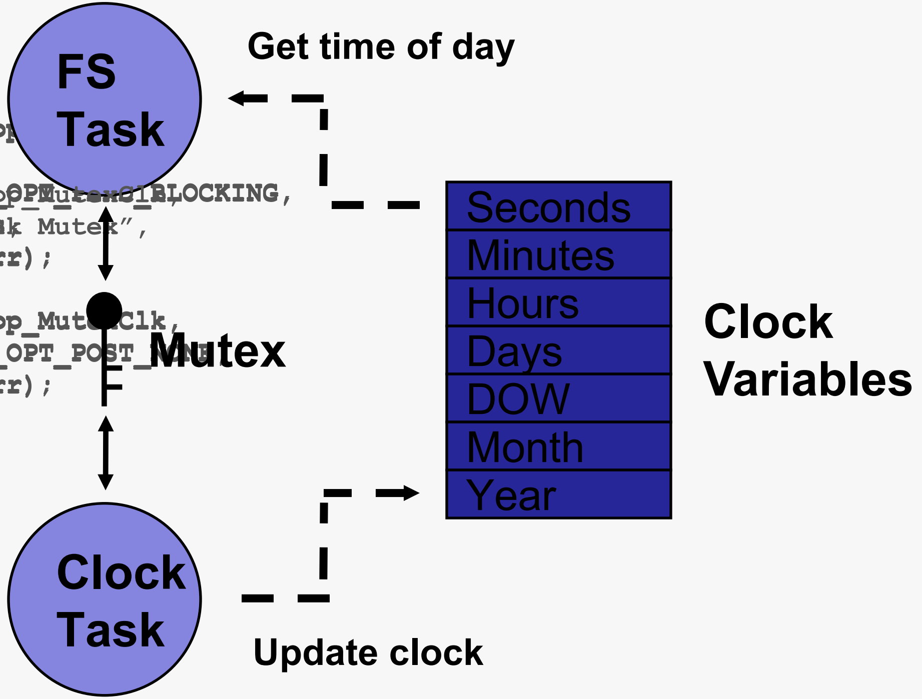
Mutex API

```
void  OSMutexCreate (OS_MUTEX    *p_mutex,
                     CPU_CHAR    *p_name,
                     RTOS_ERR    *p_err);

void  OSMutexPend (OS_MUTEX   *p_mutex,
                   OS_TICK     timeout,
                   OS_OPT      opt,
                   CPU_TS     *p_ts,
                   RTOS_ERR   *p_err);

void  OSMutexPost (OS_MUTEX  *p_mutex,
                   OS_OPT     opt,
                   RTOS_ERR  *p_err);
```



Mutex Example

```
void  App_TaskFS (void *p_arg)
void  App_TaskClk (void *p_arg)
{
    OS_MUTEX  App_MutexClk;
    while (1) {
        Create file;
        Wait for signal from timer
        OSMutexPend((OS_MUTEX *)&App_MutexClk,
                    (OS_TICK   )0,
                    (OS_OPT    )OS_OPT_PEND_BLOCKING,
                    (CPU_TS   *)&ts,
                    (RTOS_ERR *)&err);
    OSMutexCreate((OS_MUTEX *)&App_MutexClk,
                  (CPU_CHAR *)"Clk Mutex",
                  (RTOS_ERR *)&err);
        Read clock;
        Update clock;
        OSMutexPost((OS_MUTEX *)&App_MutexClk,
                    (OS_OPT    )OS_OPT_POST_NONE,
                    (RTOS_ERR *)&err);
        Add timestamp to file;
    }
}
```



Choosing How to Protect Shared Resources

- Shared resources that are accessed by ISRs can only be protected by disabling interrupts
- Application code should not disable either interrupts are the scheduler for extended periods of time
- Semaphores should only be used for synchronization
- Mutexes protect against priority inversion

Which resource sharing method is best?

- **ISR <-> Tasks:**
  - Shared resources that are accessed by ISRs can only be protected by disabling interrupts
- **Task <-> Task**
  - Application code should not disable either interrupts or the scheduler for extended periods of time
    - Avoid locking the scheduler unless absolutely necessary
  - Use Mutexes (eliminate unbounded priority inversions)
  - **DO NOT** use Semaphores … only use them for synchronization

Section Summary – Sharing Resources

- What's a resource?
  - Shared variable, structure, table or I/O device
- It is the developer's responsibility to protect shared resources
  - Micrium OS Kernel only gives you services to help you
- Methods:
  - Disable/Enable interrupts (Affects interrupt latency – ISRs/Tasks)
  - Lock the Scheduler (Defeats the task priority)
  - Semaphore (Should not be used – Priority Inversion)
  - Mutex (Preferred)

Lab #5

Inter-Task Communication & Dynamic Memory Pools

Inter-Task Communication Services

- Tasks in a Micrium OS Kernel application can send and receive messages using services that the kernel provides
- Application developers determine the contents of these messages
- Micrium OS Kernel's message passing services (or more formally, its inter-task communication services) have much in common with its synchronization and mutual exclusion services

Inter-Task Communication Example

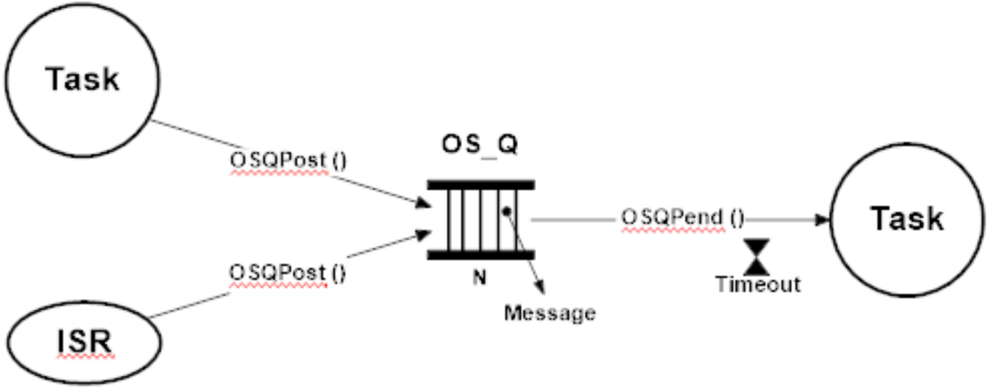
```
void  App_TaskFIR (void *p_arg)
{
    while (1) {
        Read next sample;
        Calculate filter output;
        Send output value to App_TaskLog();
    }
}


void  App_TaskLog (void *p_arg)
{
    while (1) {
        Receive output from App_TaskFIR();
        Write filter output to file;
    }
}
```

Message Queues

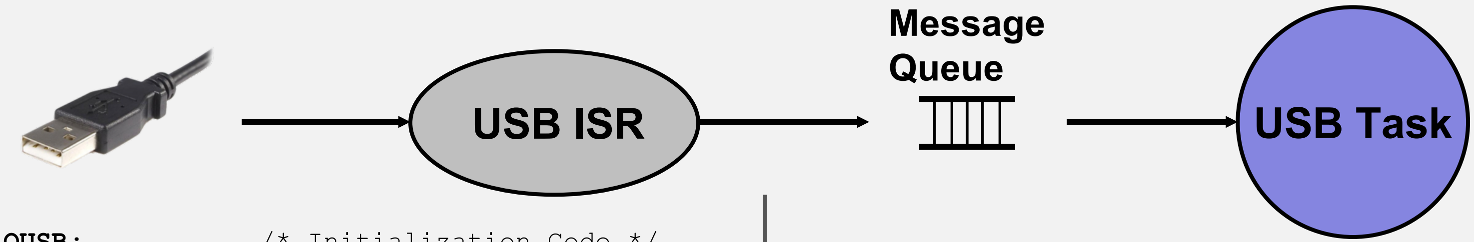
- In Micrium OS Kernel, a message queue is a list of OS_MSG structures
  - The kernel manages the list
  - Through API functions, tasks can request the insertion or removal of messages
- A message is a void pointer
  - The sender and the receiver must agree on the meaning of the message
- Tasks or ISRs can send messages
  - Only tasks can receive messages
- When a task is waiting on a message, the kernel runs other tasks

Message Queue API

```
void  OSQCreate (OS_Q          *p_q,
                 CPU_CHAR      *p_name,
                 OS_MSG_QTY     max_qty,
                 RTOS_ERR      *p_err);

void  *OSQPend (OS_Q          *p_q,
                OS_TICK        timeout,
                OS_OPT         opt,
                OS_MSG_SIZE   *p_msg_size,
                CPU_TS        *p_ts,
                RTOS_ERR      *p_err);

void   OSQPost (OS_Q          *p_q,
                void          *p_void,
                OS_MSG_SIZE    msg_size,
                OS_OPT         opt,
                RTOS_ERR      *p_err);
```



Message Queue Example




```
OS_Q  App_QUSB;            /* Initialization Code */

OSQCreate((OS_Q     *)&App_QUSB,
          (CPU_CHAR *)"USB Queue",
          (OS_MSG_QTY)20,
          (RTOS_ERR *)&err);


void  App_ISR_USB (void)
{
    Clear USB (or DMA) interrupt;
    p_buf = Get Buffer from pool;
    OSQPost((OS_Q       *)&App_QUSB,
            (void       *)p_buf,
            (OS_MSG_SIZE)buf_size,
            (OS_OPT      )OS_OPT_POST_FIFO,
            (RTOS_ERR   *)&err);
}
```

```
void  App_Task_USB (void *p_arg)
{
    while (1) {
        p_buf = OSQPend((OS_Q         *)&App_QUSB,
                        (OS_TICK       )0,
                        (OS_OPT)OS_OPT_PEND_BLOCKING,
                        (OS_MSG_SIZE *)&msg_size,
                        (CPU_TS       *)&ts,
                        (RTOS_ERR     *)&err);
        Process packet;
        Free buffer back to pool;
    }
}
```

Type Casting Messages

- By casting messages, developers can sometimes avoid dealing with shared data
- Micrium OS Kernel Queues actually provide two variables that can be casted
  - void*
  - msg_size
- It is imperative that the sending task and receiving task agree on the message being sent when typecasting
  - If not, receiving task my try to dereference the value, resulting in a hard fault

```
#define APP_STATUS_OFF 0
#define APP_STATUS_ON  1

typedef OS_MSG_SIZE APP_STATUS;

OS_Q        AppQ;
CPU_INT32U AppCount  = 10;
APP_STATUS AppStatus = APP_STATUS_ON;

OSQPost(              &AppQ,
        (void*)        count,
        (OS_MSG_SIZE)  status,
                       OS_OPT_POST_FIFO,
                      &err);
```

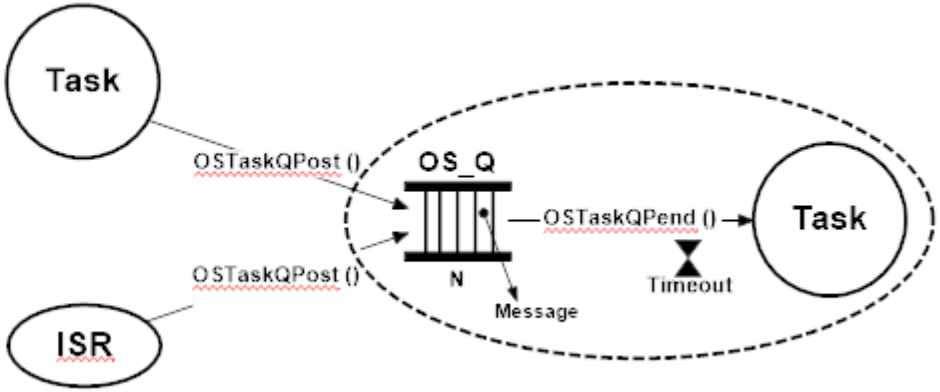
Message Queues for Resource Protection

- Queues can be used to regulate access to controlled resources
- Only one task directly accesses the resource and that task receives messages from others

- “Using design patterns to identify and partition RTOS tasks: Part 2,” Michael C. Grischy and David E. Simon, Embedded.com, www.embedded.com/columns/technicalinsights/179103020?_requestid=206440

Task Message Queue

- Similar to the Task Semaphores, message queues are so common they've been added to Micrium OS Kernel Tasks
  - Low-overhead message queue contained in TCB
- Task queue size is specified during `OSTaskCreate()`
- Other tasks and ISRs can post to the task queue
  - Need the `OS_TCB` pointer rather than an `OS_Q` object



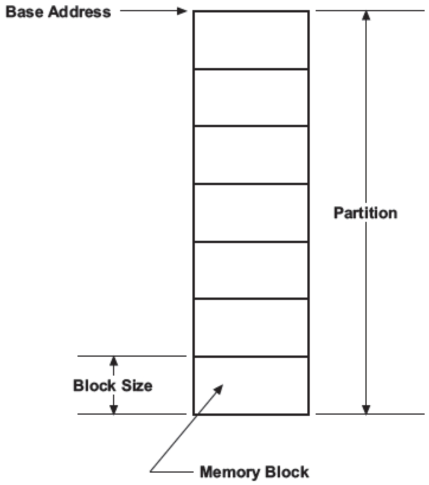
Dynamic Memory Pools

Dynamic Memory Pools

- Dynamic memory pools are a pool of memory blocks that can be dynamically allocated from either the general-purpose heap or a specific memory segment.
- Pools are configured to an initial number of blocks specified at the creation of the pool
  - You have the option to set a maximum or to allow it to expand into the general heap
- Dynamic memory pools can allocate the following:
  - General-purpose memory blocks
  - Persistent blocks that keep the data stored in them even when freed
  - Hardware memory blocks

Dynamic Memory Pools

- Most kernels provide fixed-sized memory block management (buffer pools)
  - Prevents fragmentation
  - Allows messages to be in scope
- Multiple pools can be created with each having a different block size
- You must ensure that you return blocks to the proper partition

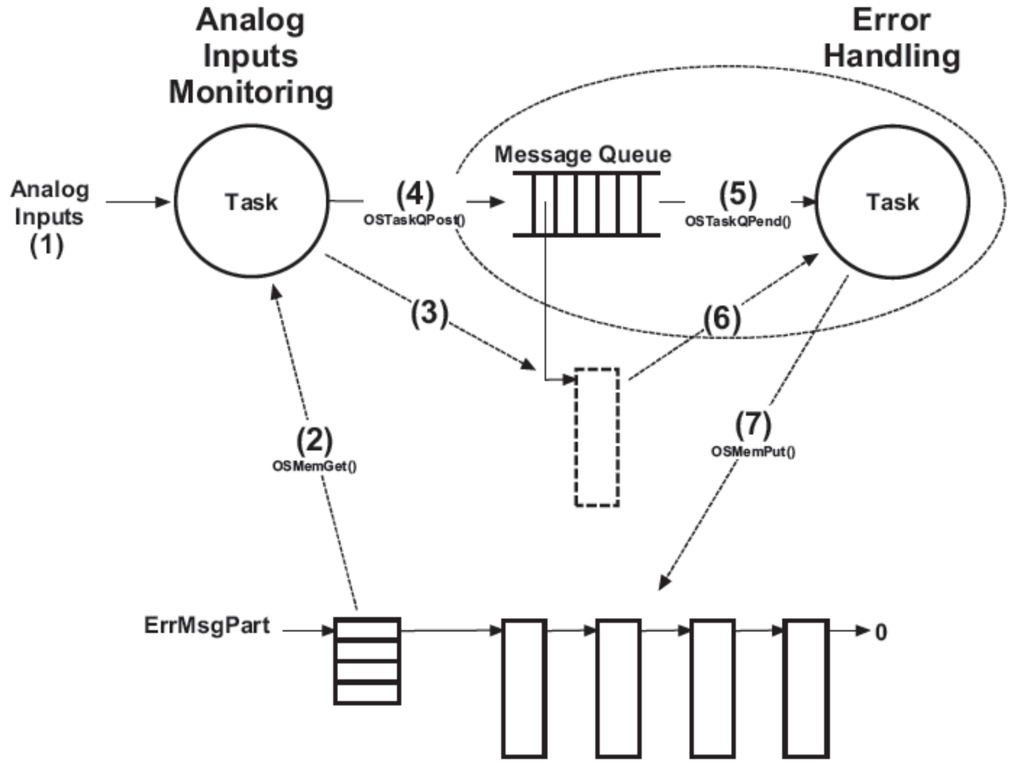


- Services:
  - **`Mem_DynPoolCreate`**() Create a memory pool
  - **`Mem_DynPoolBlkGet`**() Get a block from a pool
  - **`Mem_DynPoolBlkFree`**() Return a block to a partition

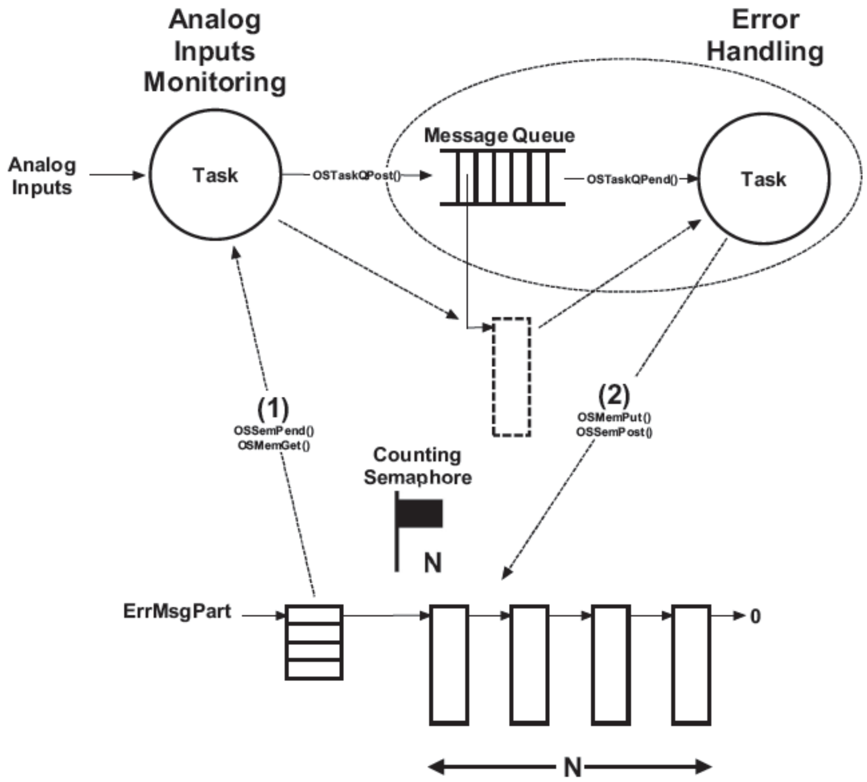
Dynamic Memory Pool API

```
void Mem_DynPoolCreate (const CPU_CHAR      *p_name,
                        MEM_DYN_POOL  *p_pool,
                        MEM_SEG       *p_seg,
                        CPU_SIZE_T     blk_size,
                        CPU_SIZE_T     blk_align,
                        CPU_SIZE_T     blk_qty_init,
                        CPU_SIZE_T     blk_qty_max,
                        RTOS_ERR      *p_err)


void* Mem_DynPoolBlkGet (MEM_DYN_POOL  *p_pool,
                         RTOS_ERR      *p_err)


void Mem_DynPoolBlkFree (MEM_DYN_POOL  *p_pool,
                         void          *p_blk,
                         RTOS_ERR      *p_err)
```

Message Queue and Memory Pool – Non Blocking



Message Queue and Memory Pool- Blocking



Summary – Inter-Task Communication & Dynamic Memory Allocation

- ISRs and Tasks can send messages to other tasks
  - Sender and recipient **need** to agree on the meaning of the message
- A Micrium OS Kernel message queue message is a **pointer**
  - Can point to data or a function
- The message **needs** to remain in scope
  - Sender: Allocate a buffer, populate it, send the address of buffer
  - Recipient: Receive address, process, return buffer to pool
- Micrium OS Kernel Tasks have built-in message queues

Lab #6

Software Timers

Timer Overview

- Timers are a relatively recent addition to Micriµm's kernels
  - Not part of µC/OS or earlier versions of µC/OS-II
- Managed by a separate timer task
- Periodic and one-shot timers can be created
- Functionality somewhat different from that of time delays
  - Starting a timer does not result in a task state change

Timer Restrictions

- Timer implementation intended to minimize overhead
- The timer code does not utilize critical sections for protecting shared resources
  - Either schedule-locking functions or mutexes are used, depending on configuration
- As a result of the approach to resource protection, timer functions cannot be invoked from ISRs

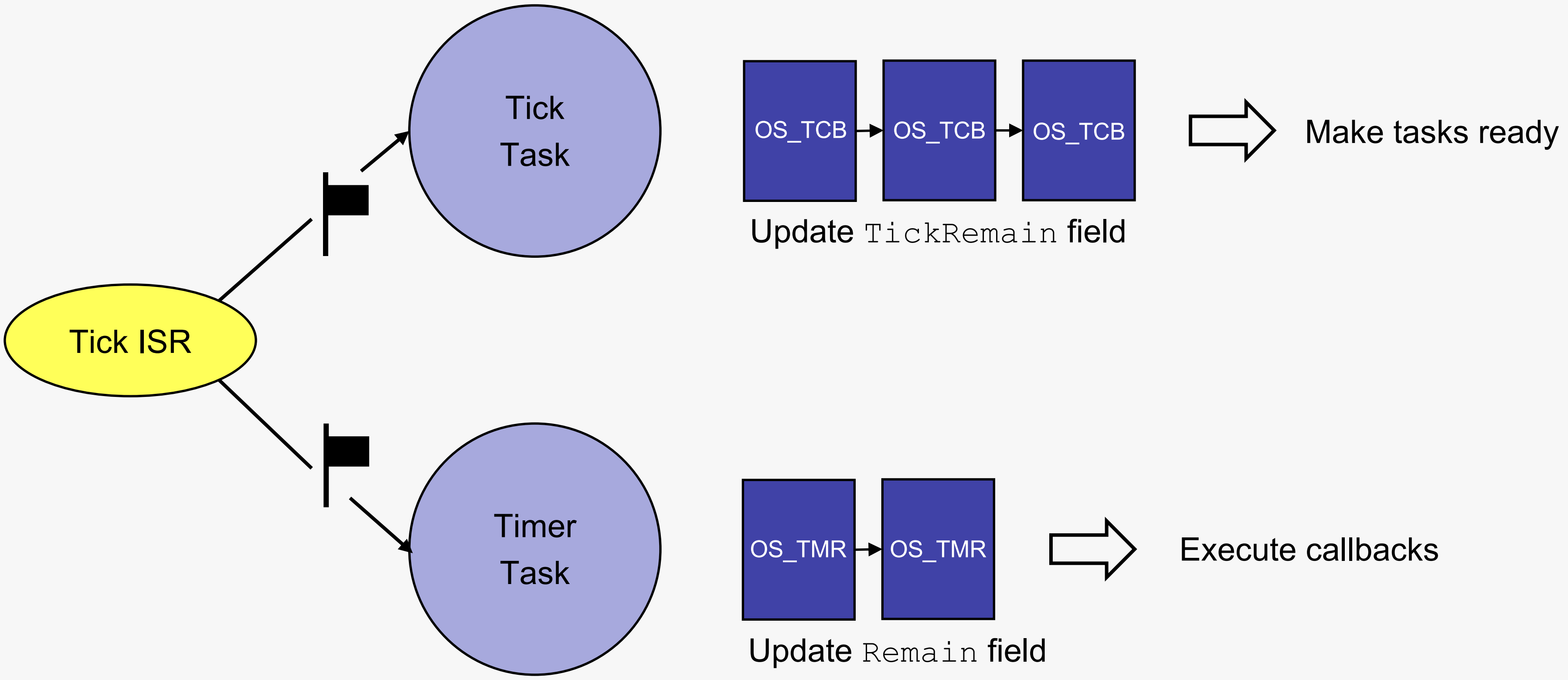
Software Timer API

```
void OSTmrCreate (OS_TMR                 *p_tmr,
                  CPU_CHAR               *p_name,
                  OS_TICK                 dly,
                  OS_TICK                 period,
                  OS_OPT                  opt,
                  OS_TMR_CALLBACK_PTR     p_callback,
                  void                   *p_callback_arg,
                  RTOS_ERR               *p_err);
```

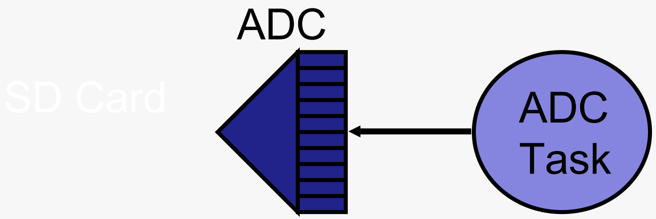
Software Timer API

```
CPU_BOOLEAN  OSTmrStart (OS_TMR     *p_tmr,
                         RTOS_ERR  *p_err);

CPU_BOOLEAN  OSTmrStop  (OS_TMR     *p_tmr,
                         OS_OPT     opt,
                         void      *p_callback_arg,
                         RTOS_ERR  *p_err);
```

Timer Implementation



Timer Example

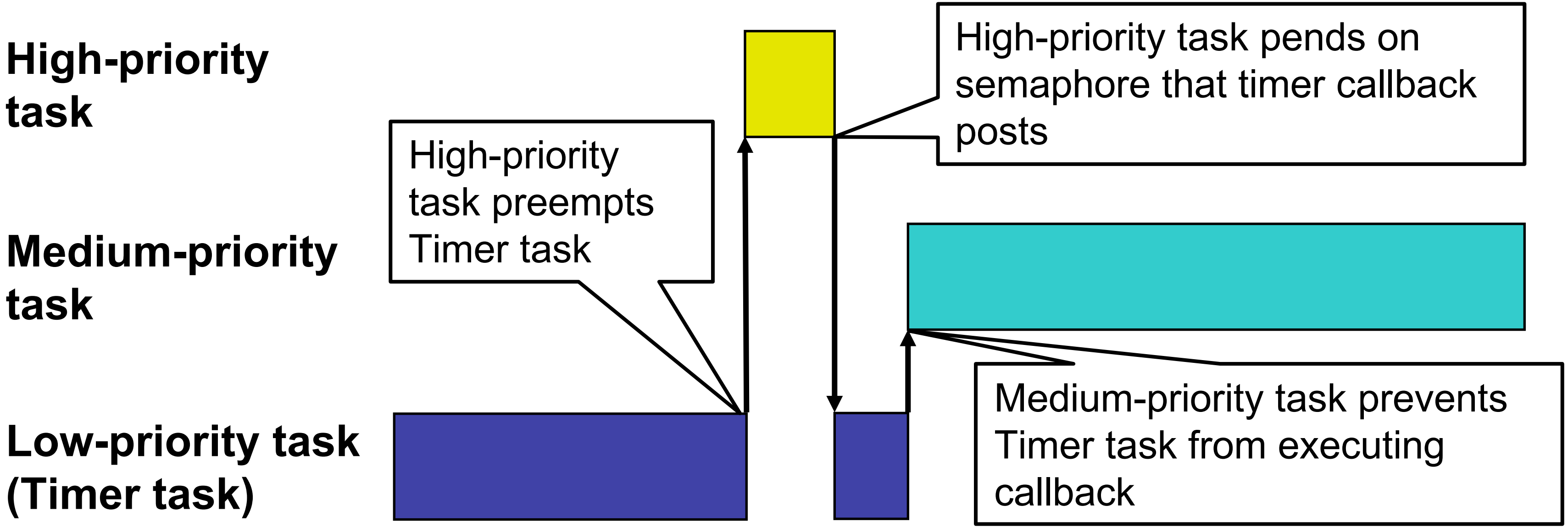



```
OS_TMR  AppTmrADC;

/* Initialization Code */

OSTmrCreate(         &AppTmrADC,
                     "ADC Timer",
                      0,
                      OS_CFG_TMR_TASK_RATE_HZ,
                      OS_OPT_TMR_PERIODIC,
                      AppTmrADC_Callback,
            (void *)  0,
                     &err);

OSTmrStart(&AppTmrADC,
           &err);
```

```
void  AppTmrADC_Callback (void *p_tmr, void *p_arg)
{
    Read ADC value;
}
```

Task Priorities

- High priority tasks should avoid using services implemented by lower priority tasks



Lab #7

Conclusion

Summary (Cont.)

- The primary function of a kernel is task management, and µC/OS 5 offers a highly efficient scheduler

- There is often a need for task interaction in multi-task systems, so kernels like µC/OS 5 offer services for synchronization, resource protection, and inter-task communication

- Additional services from µC/OS 5 include dynamic memory allocation and software timers

Thank You!