

# Microchip MASTERS

2018

QQ群：123768874

## C19L05 FRM11

# 基于FreeRTOS环境的32位 单片机软件开发入门



**MICROCHIP**

# 目标

QQ群：123768874

- 演示基于**RTOS**应用的执行模式
- 获得实用的**FreeRTOS**知识和经验
- 创建可满足所需性能要求的**MPLAB® Harmony**应用
- 调试并解决**Harmony RTOS**应用中的时序问题

# 课程安排

QQ群：123768874

- 实时系统
- **FreeRTOS**简介
- 演示1（任务创建和行为）
- 任务交互
- 演示2（使用互斥）
- **FreeRTOS**和**PIC32**架构
- 演示3（高级调试）

# 课程安排

QQ群：123768874

- 实时系统
  - 什么是RTOS?
  - 为什么要使用RTOS?
  - RTOS与GPOS
  - RTOS基础知识

# 什么是RTOS?

QQ群 : 123768874

- 系统的正确性取决于

- 逻辑正确性



- 时间正确性





# 实时系统规范

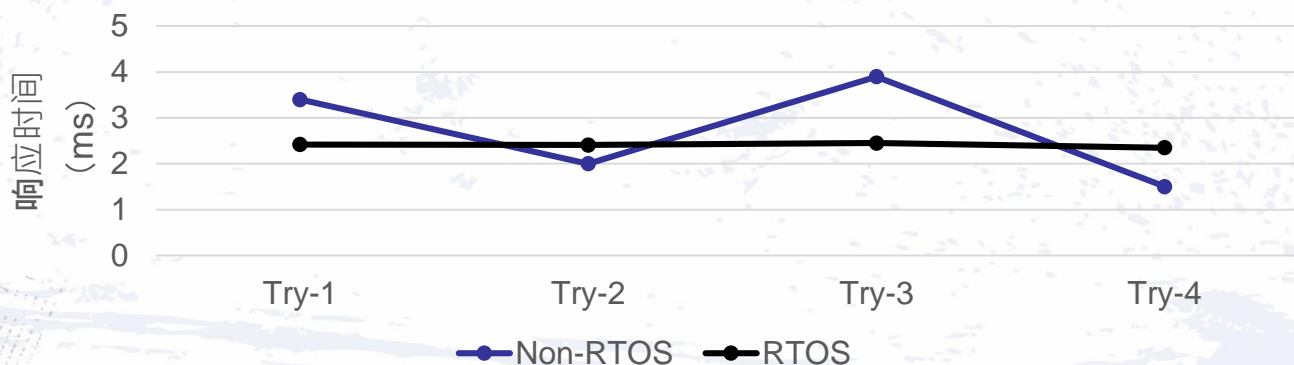
QQ群：123768874

- 逻辑正确性要求：
  - 计算得到正确的结果
  - 对资源、安全性和可靠性等的合理应用
  - 有限状态机
    - 非常适合控制逻辑和协议
    - 状态转换和输入响应
  - 数据流——由输入数据的可用性触发的模块化计算

# 实时系统规范

- 时间正确性要求：
  - 正确的时间，正确的输出
  - 计算何时可以开始，何时应该结束
  - 时间确定性

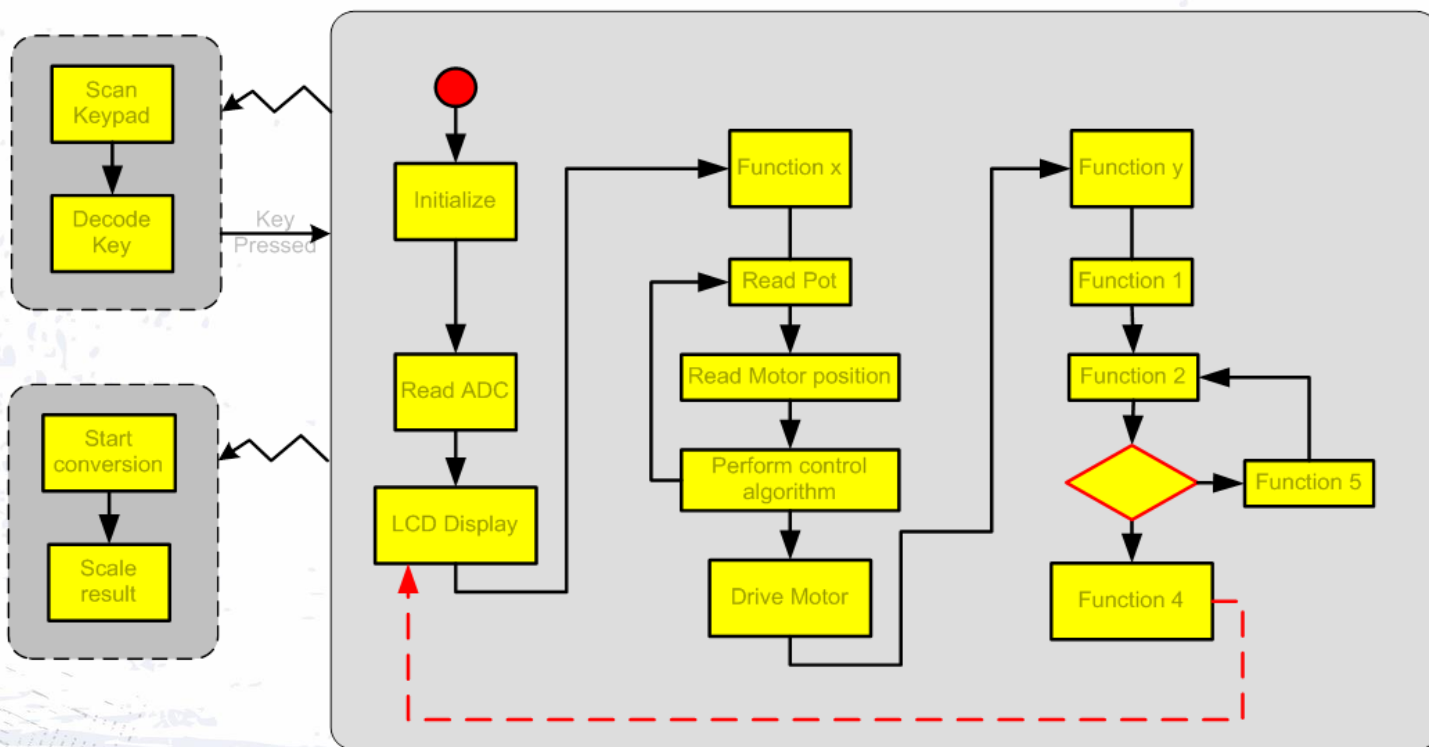
QQ群：123768874



# 为什么要使用RTOS?

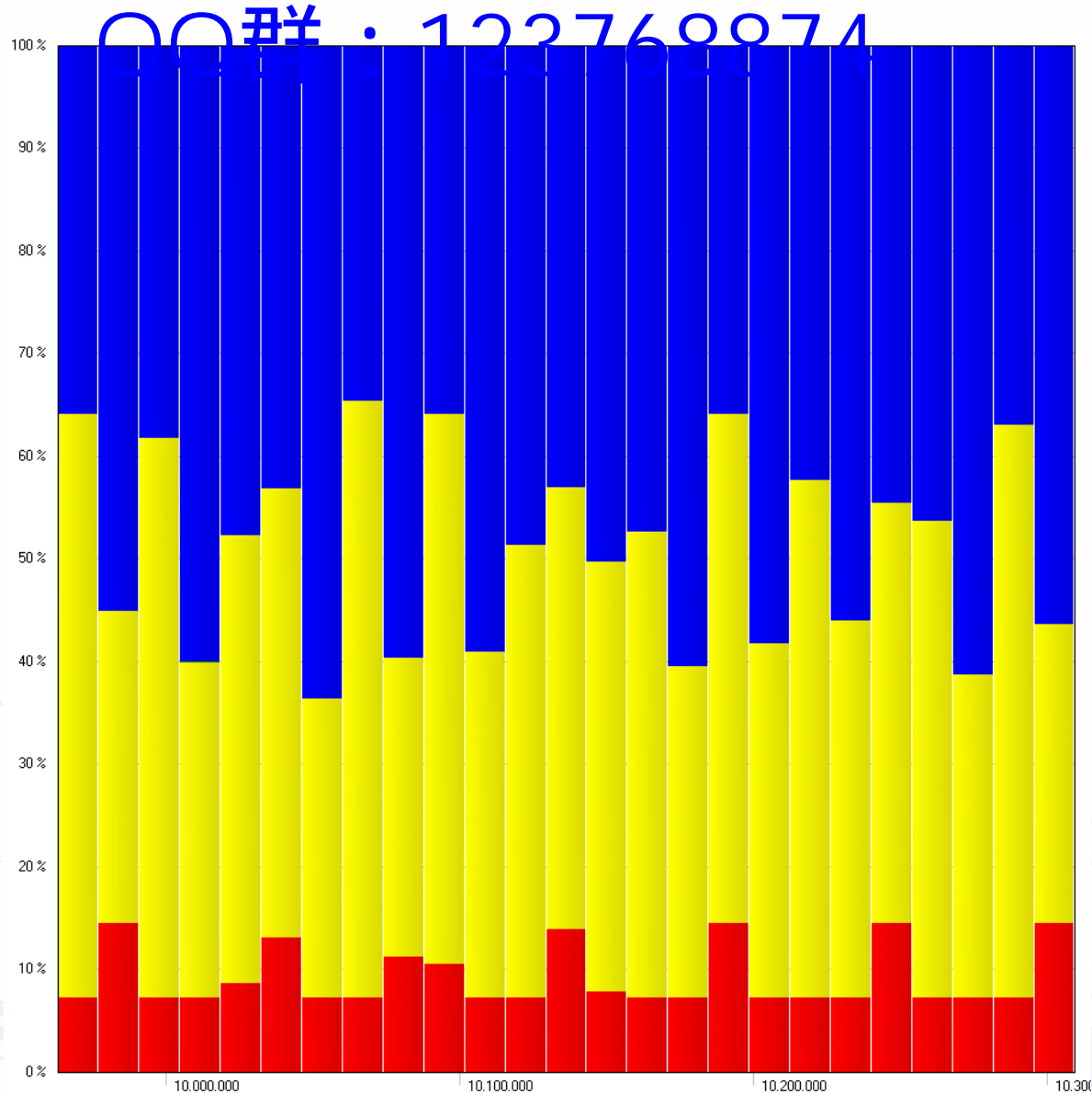
QQ群：123768874

- “多年来我一直在没有RTOS的情况下设计复杂的系统”





# 非RTOS设计（无休眠）

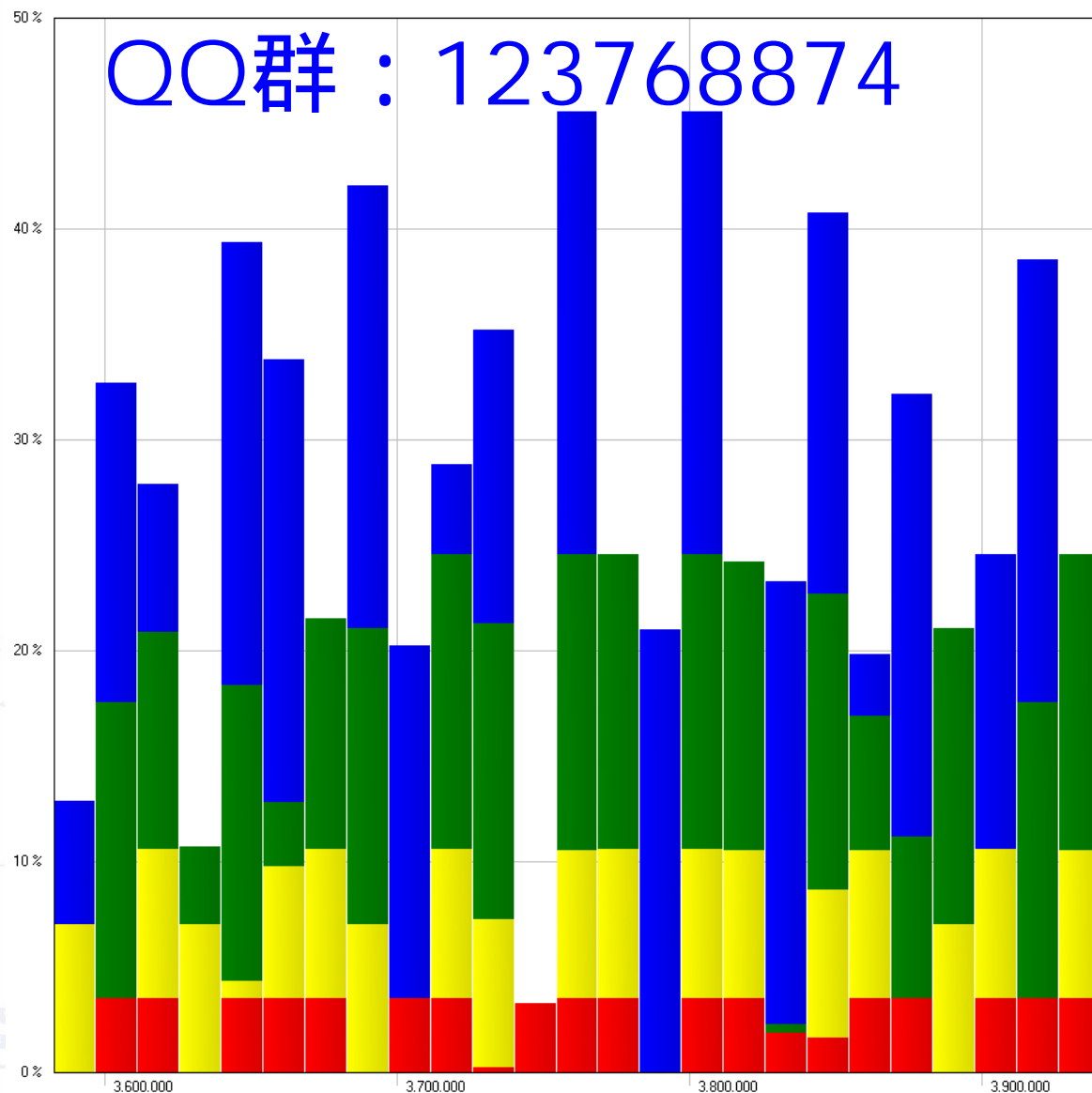


# 为什么要使用RTOS?

QQ群：123768874

- 资源限制和共享
  - 高效利用CPU
  - 临界区
- 并发事件的多任务处理
  - 控制时序
  - 调度
- 硬件独立性和可移植性
  - 软件抽象和模块化设计
  - 软件重用

# RTOS设计



# 为什么要使用RTOS?

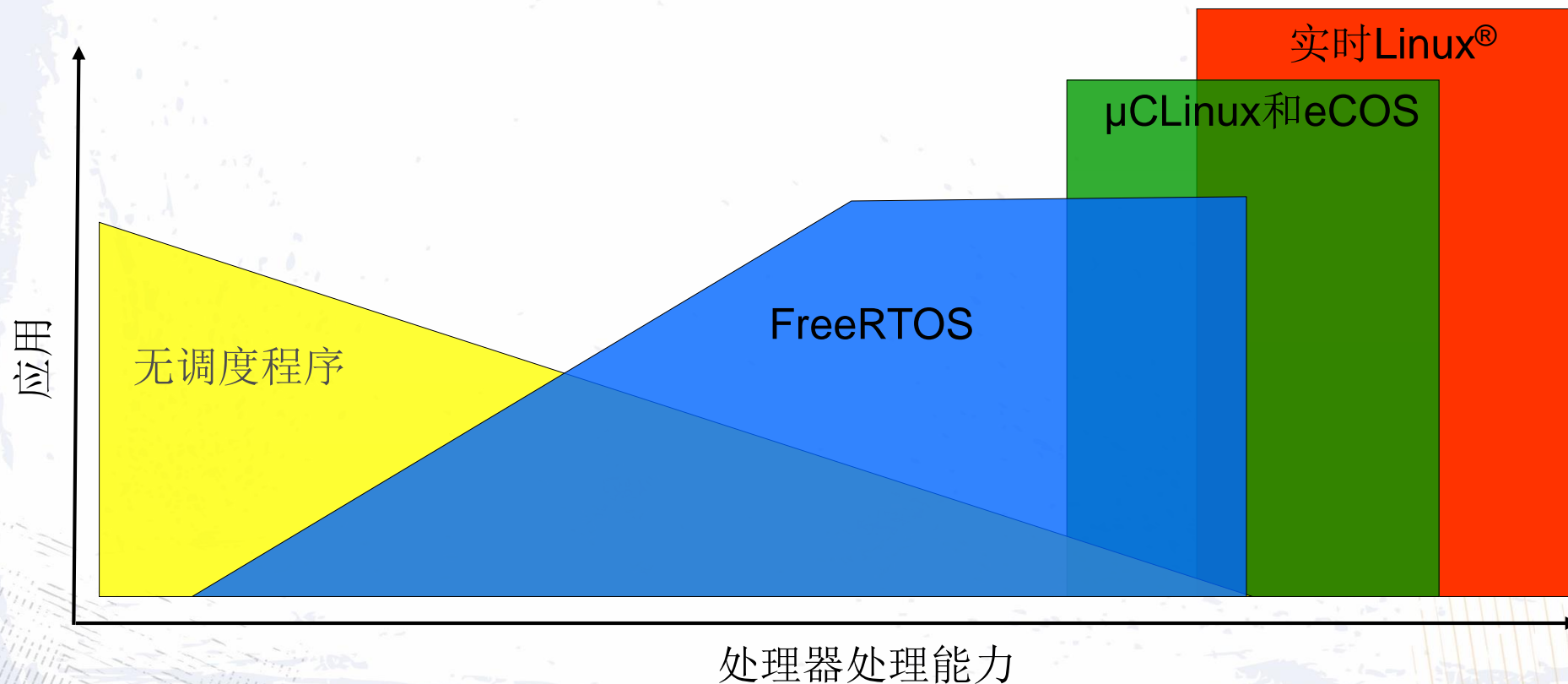
QQ群：123768874

- 但您并非必须使用
  - 调度程序的开销超过了其优势
  - 没有足够的资源来支持RTOS

.....取决于应用

# RTOS范围

QQ群：123768874





# RTOS与GPOS

## • RTOS QQ群 : 123768874 • GPOS

- 专注于时间相关任务
- 主要是基于优先级（抢占式）的调度
- 延时：微调，有时间限制
- 必须避免优先级倒置
- 例如，FreeRTOS ...

- 专注于吞吐量(处理能力)
- 主要是公平的调度
- 延时：可能取决于工作量
- 优先级倒置可能不会影响整体性能
- 例如Windows®和Linux®等

# 构建实时系统时的问题

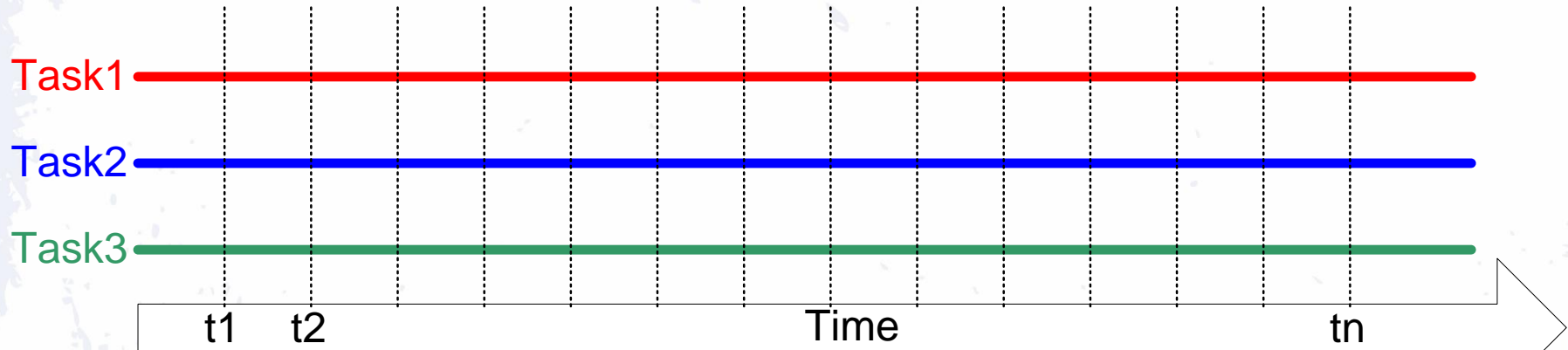
- 并发 QQ群：123768874
  - 几乎所有实时系统本质上都是并发的
    - 需要处理多个事件，以便所有响应都满足其期限
    - 可能发生多个事件，需要并行处理
- 并发控制
  - 需要使能对共享资源的序列化访问
- 通信
  - 并发活动可能需要进行通信（任务之间关联）

# 构建实时系统时的问题

- 同步 QQ群：123768874
  - 需要控制并发活动的执行顺序
- 调度定义了控制共享资源使用的方法
  - 在这种情况下为CPU的处理资源
- 我们将考虑两种调度方法
  - 静态循环调度
  - 固定优先级抢占式调度

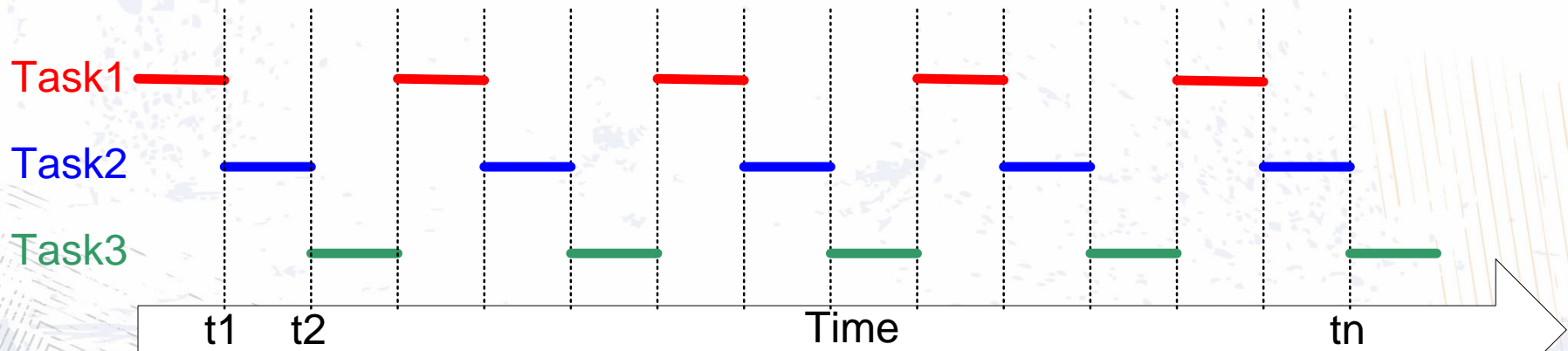
# 调度

All available tasks appear to be executing ...



QQ群 : 123768874

... but only one task is ever executing at any time.



# 术语

- 时钟节拍：QQ群：123768874
  - 调度程序决定执行任务时的周期性中断
- 强截止时间：
  - 延迟时间非常小，超时可能导致灾难性后果
- 弱截止时间：
  - 延迟导致结果可能仍然有用



# 构建实时系统

QQ群：123768874

- 循环调度是一种为调度**CPU**处理资源而构建的方法
  - 实时系统分成由循环调度程序执行的一系列函数（“任务”）
  - 并发控制，循环调度的静态结构保证了同步
  - 可以通过共享变量实现通信
  - 时序分析很容易
  - 但是
    - **CPU**没有得到有效使用
    - 维护很困难

# 简单的循环调度

QQ群：123768874

任务	所需的采样率 (最小值)	处理时间
t1	3 ms (333 Hz)	0.5 ms
t2	6 ms (166 Hz)	0.75 ms
t3	14 ms (71 Hz)	1.25 ms

```
void main(void)
{
    ...
    while (1)
    {
        t1();
        t2();
        t3();
        delay_until_3ms();
    }
}
```

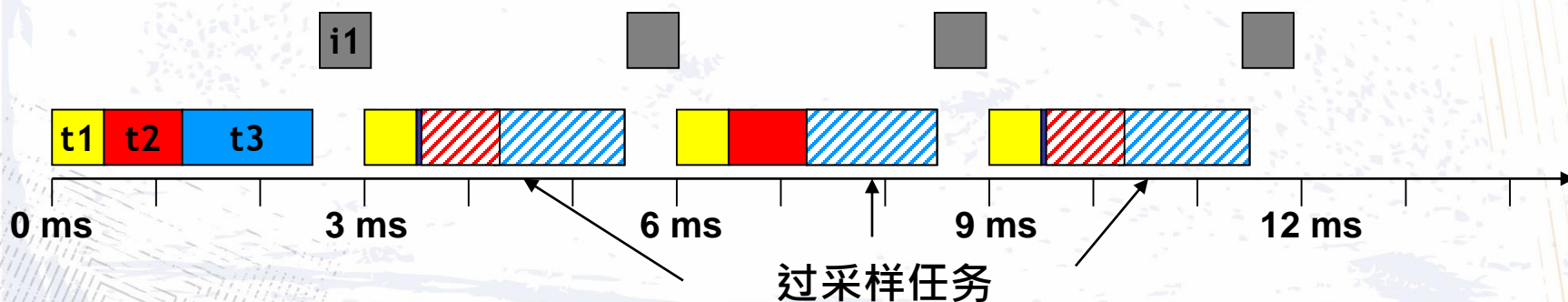


# 简单的循环调度

任务	所需的采样率 (最小值)	处理时间
t1	3 ms (333 Hz)	0.5 ms
t2	6 ms (166 Hz)	0.75 ms
t3	14 ms (71 Hz)	1.25 ms

- **t2需要12.5% CPU ( $0.75/6$ )**
  - 实际使用25% ( $0.75/3$ )
- **t3需要9% CPU ( $1.25/14$ )**
  - 实际使用42% ( $1.25/3$ )
- **每3 ms添加一次中断i1，系统过载**

QQ群：123768874



# 总结：循环调度

QQ群：123768874

## • 优点

- 可预测
  - 易于确定实时性能
  - 通过静态调度确保同步
- 易于实现
  - 利用非常简单的代码实现调度程序
  - 可以在很大程度上忽略并发和并发控制带来的问题

## • 缺点

- 效率低下（10%-30%）
  - 按约定循环调度导致有些任务过度调度
  - 对突发事件处理不佳
- 维护
  - 导致软件结构被调度约束
  - 广泛使用全局变量
  - 软件可重用性差

# 固定优先级抢占式调度（RTOS）

- 抢占调度可以克服静态调度的缺点
  - 系统设计受实现影响最小
- 但是 QQ群：123768874
  - 时序分析更难
  - 运行时开销必须很小
  - 需要添加控制并发和同步的机制



# 固定优先级抢占式调度

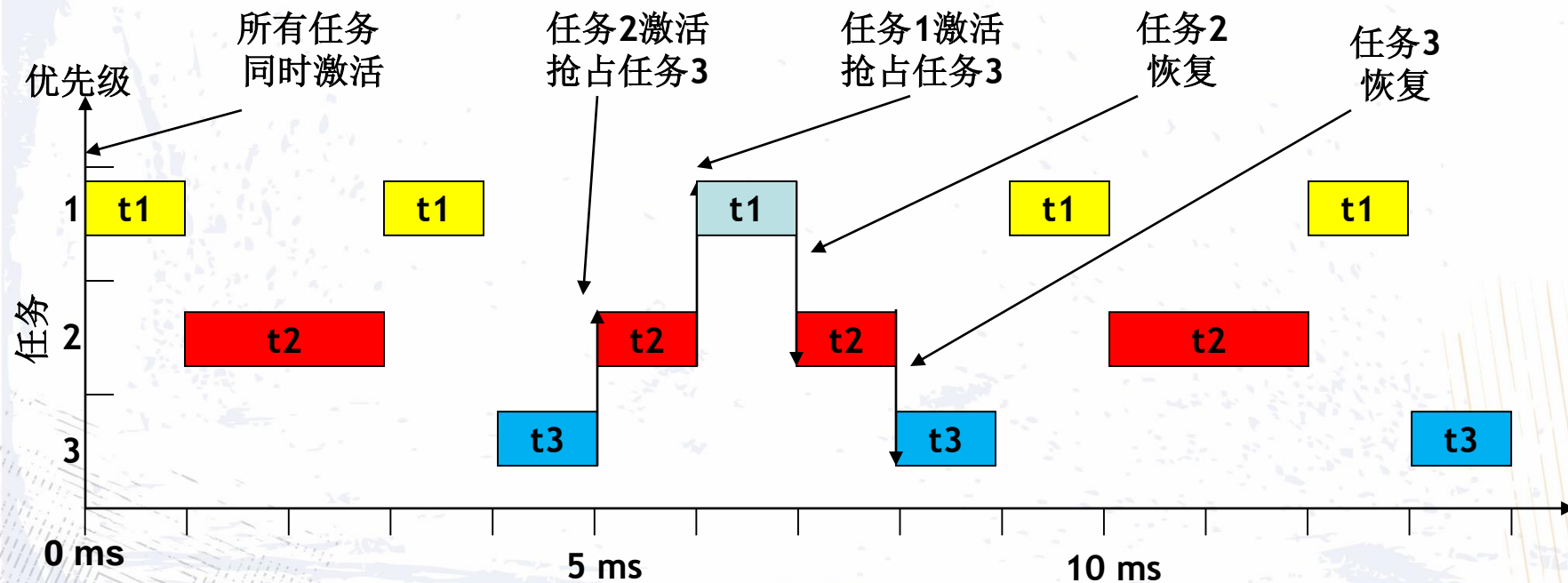
QQ群：123768874

- 系统被分解为若干任务以实现并发活动
- 为每个任务分配固定的优先级
  - 调度程序始终运行最高优先级的就绪任务
  - 高优先级任务抢占低优先级任务
  - 优先级可以更改！
- 任务可周期性就绪或偶发性就绪
  - 有效利用可用的CPU时间
  - 对程序线性度有重大影响

# 固定优先级抢占式调度

QQ群：123768874

任务	所需的采样率 (最小值)	优先级	计算时间
t1	3 ms (1000 Hz)	高	1 ms
t2	5 ms (333 Hz)	中	2 ms
t3	9 ms (200 Hz)	低	2 ms



# 小测试

QQ群：123768874

- **RTOS比裸机代码更快**

真

假

视情况而定

- **抢占式快于非抢占式**

真

假

视情况而定

# 课程安排

QQ群：123768874

- 实时系统
- **FreeRTOS简介**
- 演示1（任务创建和行为）
- 任务交互
- 演示2（使用互斥）
- **FreeRTOS和PIC32架构**
- 演示3（高级调试）

# 课程安排

QQ群：123768874

- **FreeRTOS简介**
  - 什么是FreeRTOS?
  - FreeRTOS的优点
  - 基本概念
  - FreeRTOS API的概述



# 什么是FreeRTOS

- 调度器

- 抢占式
- 合作式
- 任务
- 协同程序
- 高效的软件定时器

QQ群：123768874

- 开发支持

- 堆栈溢出检测
- 综合跟踪宏
- 可配置内核和使用诊断
- 运行时统计
- 全功能的跟踪可视化工具

# 什么是FreeRTOS

- **通信/同步**
  - 报文队列
  - 二进制信号量
  - 计数信号量
  - 递归信号量
  - 互斥（具有简单的优先级继承）
- **事件组/位**
- **延迟的中断处理**
- **支持 低功耗应用的无时钟节拍模式**
- **队列集**
- **Win32模拟器**

# 源代码树

FreeRTOSv10.0.0	Name	Date modified	Type	Size
FreeRTOS	include	11/27/2017 6:57 PM	File folder	
Demo	portable	11/27/2017 6:57 PM	File folder	
License	croutine	11/24/2017 11:02 ...	C File	13 KB
Source	event_groups	11/27/2017 11:00 ...	C File	25 KB
FreeRTOS-Plus	list	11/24/2017 11:02 ...	C File	9 KB
	queue	11/24/2017 11:03 ...	C File	92 KB
	readme	9/17/2013 1:17 AM	Text Document	1 KB
	stream_buffer	11/27/2017 9:00 PM	C File	43 KB
	tasks	11/29/2017 3:40 PM	C File	162 KB
	timers	11/24/2017 11:03 ...	C File	39 KB

- 简单的文件结构
- **100**多个演示

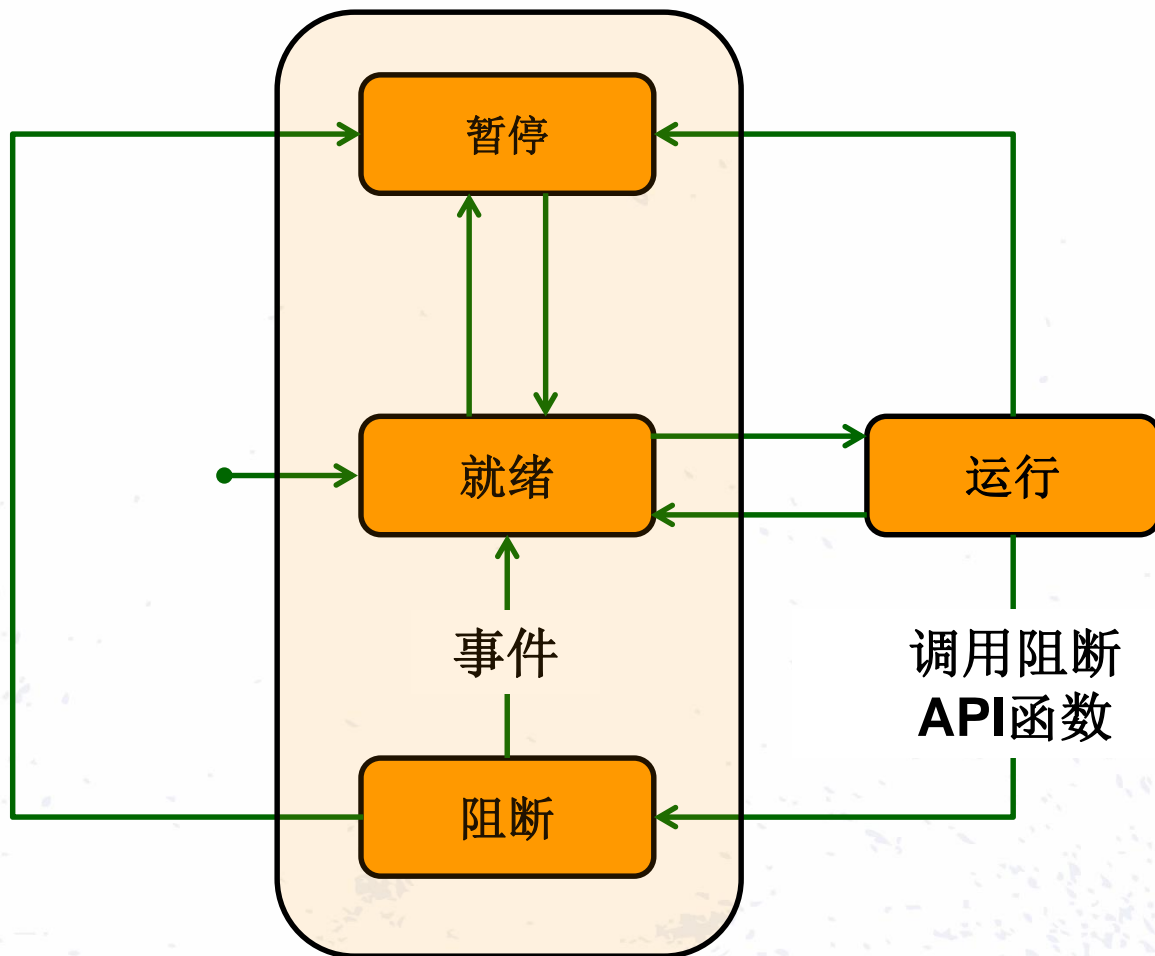
# FreeRTOS的优点

- 稳健
- 小而简单
- 出色的支持
- 开箱即用的项目
- 简单的许可模型（license）

# FreeRTOS API的概述

- **任务**
  - 在系统中调度和执行的作业单元（一组作业）
- **信号量和互斥**
  - 由多个并发任务调用的代码必须是可重入的
- **队列**
  - 任务间通信

# 任务状态



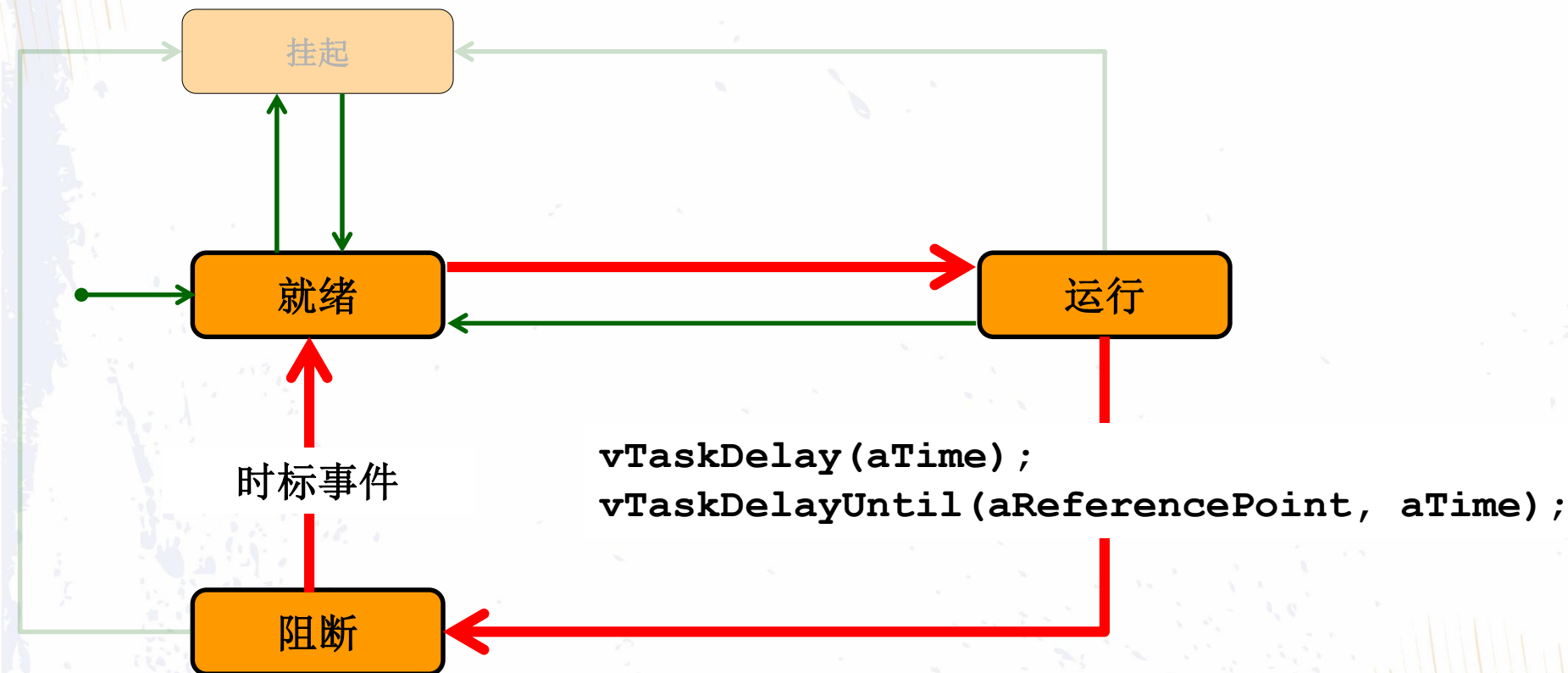
**RTOS**的工作是将任务（您的代码）在各种状态之间转换



# FreeRTOS任务函数

```
/* Tasks always have the same prototype. */  
void aTask( void *pvParameters )  
{  
    for( ;; )  
    {  
        /* Task processing goes here. */  
    }  
  
    /* A task cannot exit without first deleting itself. */  
    vTaskDelete( NULL );  
}
```

# 任务延时



- 时间以“节拍”指定
- 效率很高，只在需要时才执行

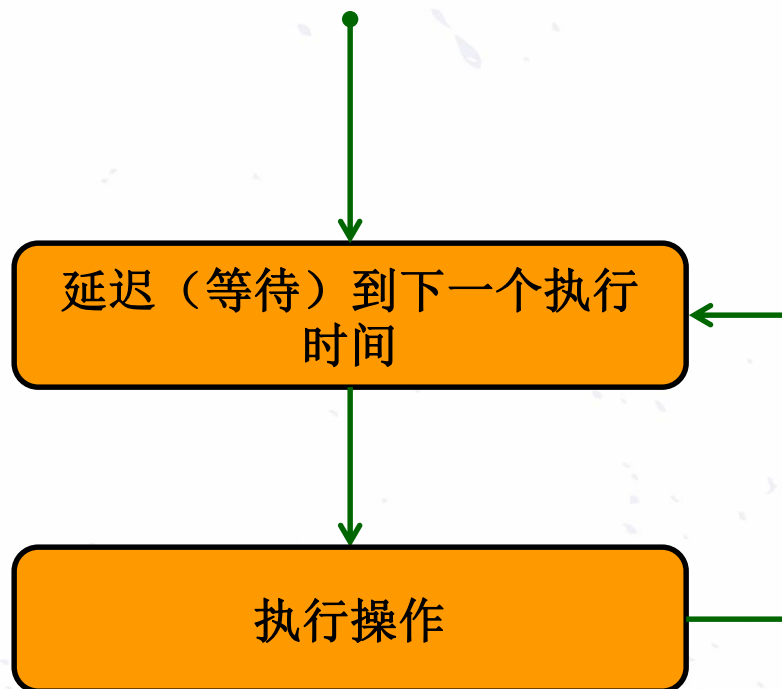
# 创建任务

```
xTaskCreate( /* A pointer to the task function. */  
            aTask,  
  
            /* Text name. */  
            "LED",  
  
            /* Size of task stack in words. */  
            100,  
  
            /* Parameters passed into the task. */  
            (void *) 0,  
  
            /* The priority of the task. */  
            2,  
  
            /* A handle for the task. */  
            NULL  
        );
```

# RTOS main()函数

```
int main( void )  
{  
    /* Create tasks to implement the program. More  
       can be created later if required.*/  
    xTaskCreate( ... );  
    xTaskCreate( ... );  
  
    /* Start the scheduler. Once started only tasks  
       and interrupts will execute.*/  
    vTaskStartScheduler();  
  
    /* Should never reach here! */  
}
```

# 示例任务



一段简单的自主连续代码

# 示例任务

```
xTaskHandle task1handle, task2handle; // declare task handles
```

```
int main(void)
```

```
{
```

```
    xTaskCreate(periodic_task, // pointer to task function  
                "Task1",      // task name  
                100,          // task stack size (400 bytes)  
                (void*)500,    // parameter (period in ms)  
                3,            // priority  
                &task1handle); // pointer to task handle
```

```
    xTaskCreate(periodic_task, "Task2", 100, (void*)1000,  
                1, &task2handle);
```

```
    vTaskStartScheduler();
```

```
    printf("Exited scheduler!");
```

```
    return EXIT_FAILURE;
```

```
}
```



# 示例任务

```
static void periodic_task(void *pvParameters)
{
    TickType_t xPeriod, xNextWakeTime;

    /* Convert input period to ticks */
    xPeriod = pdMS_TO_TICKS((int) pvParameters);

    /* Initialize xNextWakeTime - only needs to be done once */
    xNextWakeTime = xTaskGetTickCount();
    for (;;)
    {
        /* Place this task in the blocked state until it is time
        to run again */
        vTaskDelayUntil(&xNextWakeTime, xPeriod);
        // ... Task Operations...
        // ... Task Operations...
    }
}
```

# FreeRTOS配置

- 每个项目都有一个**FreeRTOSConfig.h**文件

```
#define configUSE_PREEMPTION          1
#define configUSE_TICK_HOOK          0
#define configTICK_RATE_HZ           1000
#define configCPU_CLOCK_HZ            80000000UL
#define configMAX_PRIORITIES          5
#define configTOTAL_HEAP_SIZE         28000
#define configCHECK_FOR_STACK_OVERFLOW 2

#define configUSE_TIMERS               1

#define INCLUDE_vTaskDelete            0
#define INCLUDE_vTaskDelayUntil        1

#define configKERNEL_INTERRUPT_PRIORITY 0x01
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 0x03
```

# 钩子函数

- 使用周期性**ISR**以“节拍”为单位测量时间
- “时钟节拍钩子”可以在每次节拍递增时执行
- “**idle hook**”可以通过空闲任务调用
  - 低优先级处理
  - 测量空闲时间
  - 将处理器置于低功耗模式

```
void vApplicationIdleHook( void )  
{  
    /* Perform low priority  
    processing here. */  
}
```

```
void vApplicationTickHook( void )  
{  
    /* Perform quick periodic  
    processing here - timers,  
    give semaphores, etc. */  
}
```

# 小测试

- 什么可以用来测量应用程序的空闲时间？
  - 空闲钩子函数（idle hook function）
- 如果所有任务具有相同的优先级会怎样？
  - 如果使能时间片调度 -> 循环执行
  - 否则会发生任务饥饿

# 使用MPLAB® Harmony

- **MPLAB Harmony组件以状态机形式实现**
  - 必须反复调用
- **可以执行状态机**
  - 在裸机超级循环中
  - 在中断中
  - 在RTOS任务中

# 裸机操作

```
void SYS_Tasks( void )
{
    /* Maintain system services */

    SYS_CONSOLE_Tasks(sysObj.sysConsole0);

    /* Maintain middleware */

    USB_HOST_Tasks(sysObj.usbHostObject0);

    /* Maintain application state machines */
    APP0_Tasks();
    APP1_Tasks();
    APP2_Tasks();
}
```



# MPLAB® Harmony中的 FreeRTOS

```
void SYS_Tasks( void )
{
    /* Create OS Thread for Sys Tasks. */
    xTaskCreate((TaskFunction_t) _SYS_Tasks,
                "Sys Tasks", 1024, NULL, 1, NULL);

    /* Create OS Thread for APP0 Tasks. */
    xTaskCreate((TaskFunction_t) _APP0_Tasks,
                "APP0 Tasks", 1024, NULL, 3, NULL);

    /* Create OS Thread for APP1 Tasks. */
    xTaskCreate((TaskFunction_t) _APP1_Tasks,
                "APP1 Tasks", 1024, NULL, 3, NULL);

    /* Create OS Thread for APP2 Tasks. */
    xTaskCreate((TaskFunction_t) _APP2_Tasks,
                "APP2 Tasks", 1024, NULL, 3, NULL);

    /* Start RTOS */
    vTaskStartScheduler(); /* This function never returns. */
}
```

# MPLAB® Harmony中的 FreeRTOS

```
static void _SYS_Tasks ( void)
{
    while(1)
    {
        /* Maintain system services */

        /* Maintain Middleware */

        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

static void _APP1_Tasks(void)
{
    while(1)
    {
        APP1_Tasks();
    }
}

...
```



# 演示1： 使用Tracealyzer创建3个任务并 监控任务执行

# 演示1——目的

## 观察任务优先级如何影响任务执行和时序

- 观察：
  - 基于优先级的任务抢占

# 演示1——目标

- 创建**FreeRTOS**项目
- 使用**FreeRTOS**创建**3**个任务
- 为每个任务设置优先级
- 运行**FreeRTOS**应用
- 使用**Tracealyzer**监控任务调度

# 演示1——总结

- 我们使用**FreeRTOS API**创建了**3**个任务
- 我们设置了任务的优先级
- 我们将**Tracelyzer**集成到项目中
- 我们观察了每个任务的优先级如何影响他们的运行时间



# 课程安排

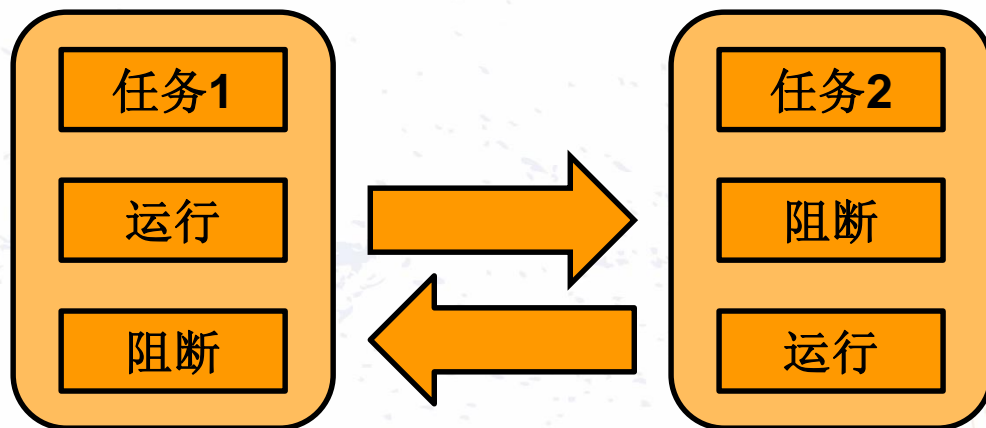
- 实时系统
- FreeRTOS简介
- 演示1（任务创建和行为）
- 任务交互
- 演示2（使用互斥）
- FreeRTOS和PIC32架构
- 演示3（高级调试）

# 课程安排

- 任务交互
  - 任务同步/共享资源
  - 竞争条件
  - 信号量
  - 队列

# 任务同步

- 什么是任务同步？
  - 交互任务
  - 任务间依赖关系
    - 报文
    - 事件
    - 数据



# 任务同步

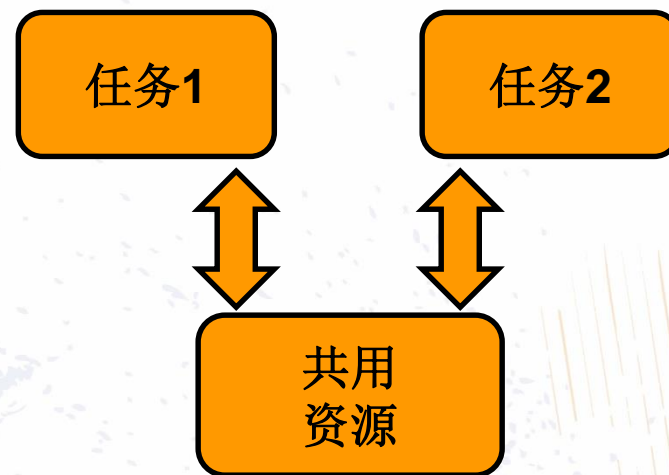
- 同步类型
  - 进程同步
  - 数据同步
- 何时需要同步
  - 生成方-消耗方
  - 读取方-写入方
- 同步实现
  - 共享变量
  - 信号

# 小测试

- 举例说明 什么时候需要同步？

# 共享资源

- 多任务共享的公共资源
- **MCU**中的资源
  - 外设
  - 变量
  - 数组
  - 数据结构





# 小测试

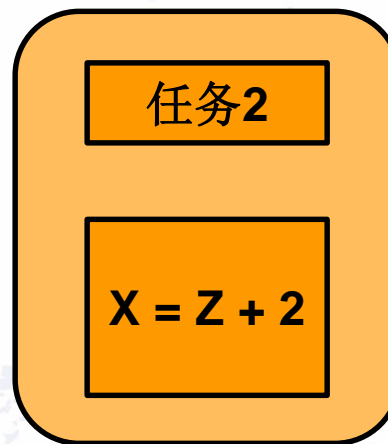
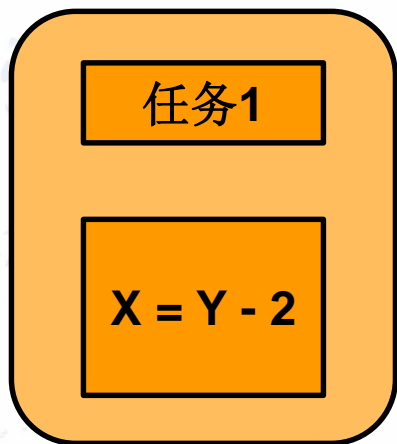
- 可能需要在任务间共享资源的情况有哪些？

# 资源竞争

- 什么是资源竞争？
  - 操作的结果取决于不可控事件的时序
- 关键竞争条件
- 非关键竞争条件
- 多任务使用的资源

# 临界区

- 什么是临界区
  - 需要防止其受多任务并发执行影响的代码

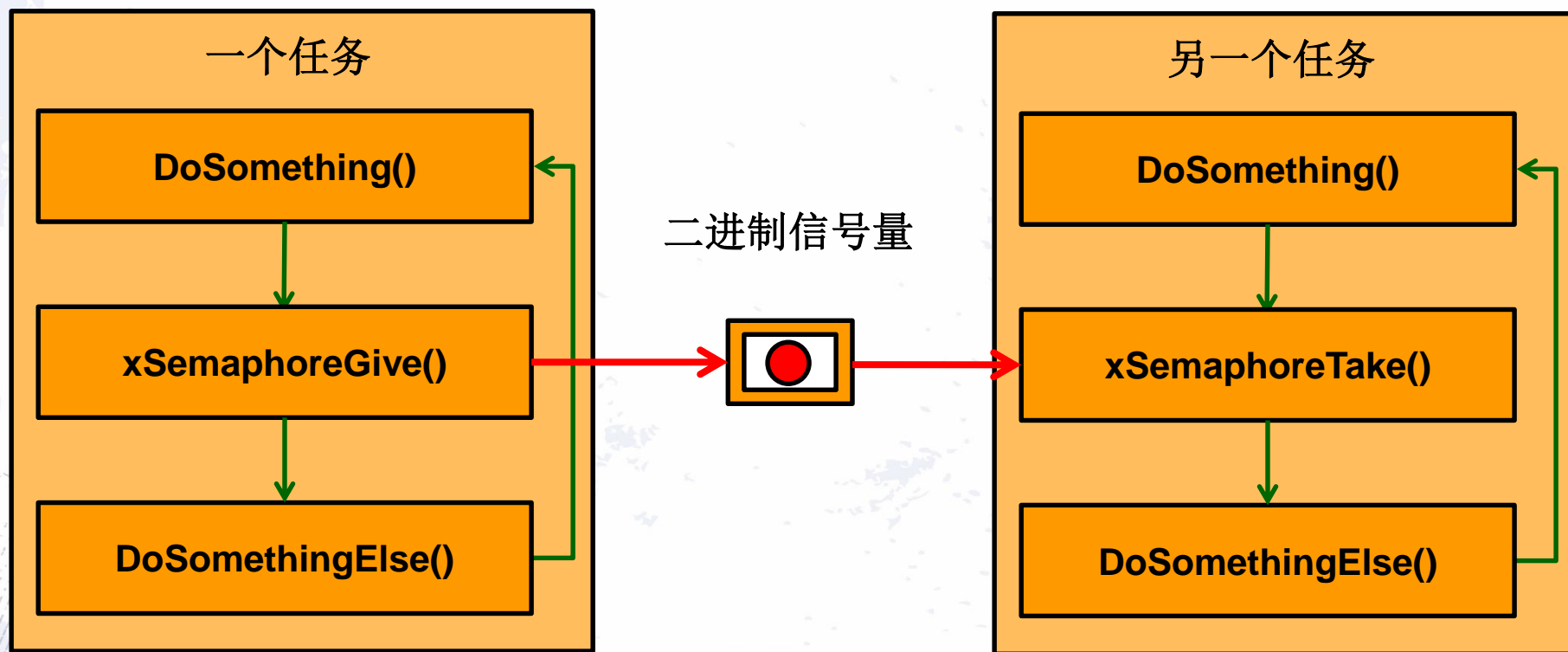


# 信号量

- 用于保护临界区
- 用于同步任务
- 信号量类型：
  - 二进制
  - 计数
  - 互斥

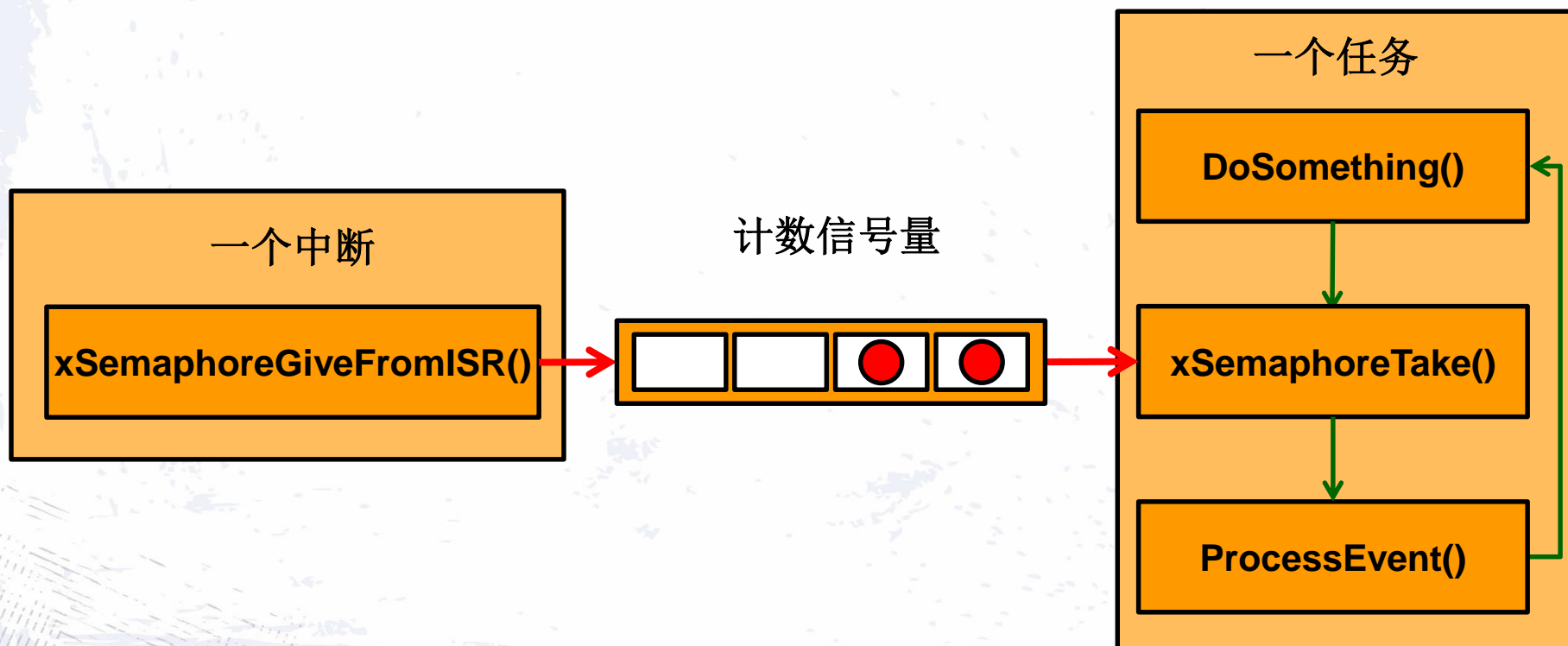
# 二进制信号量

- 信号任务



# 计数信号量

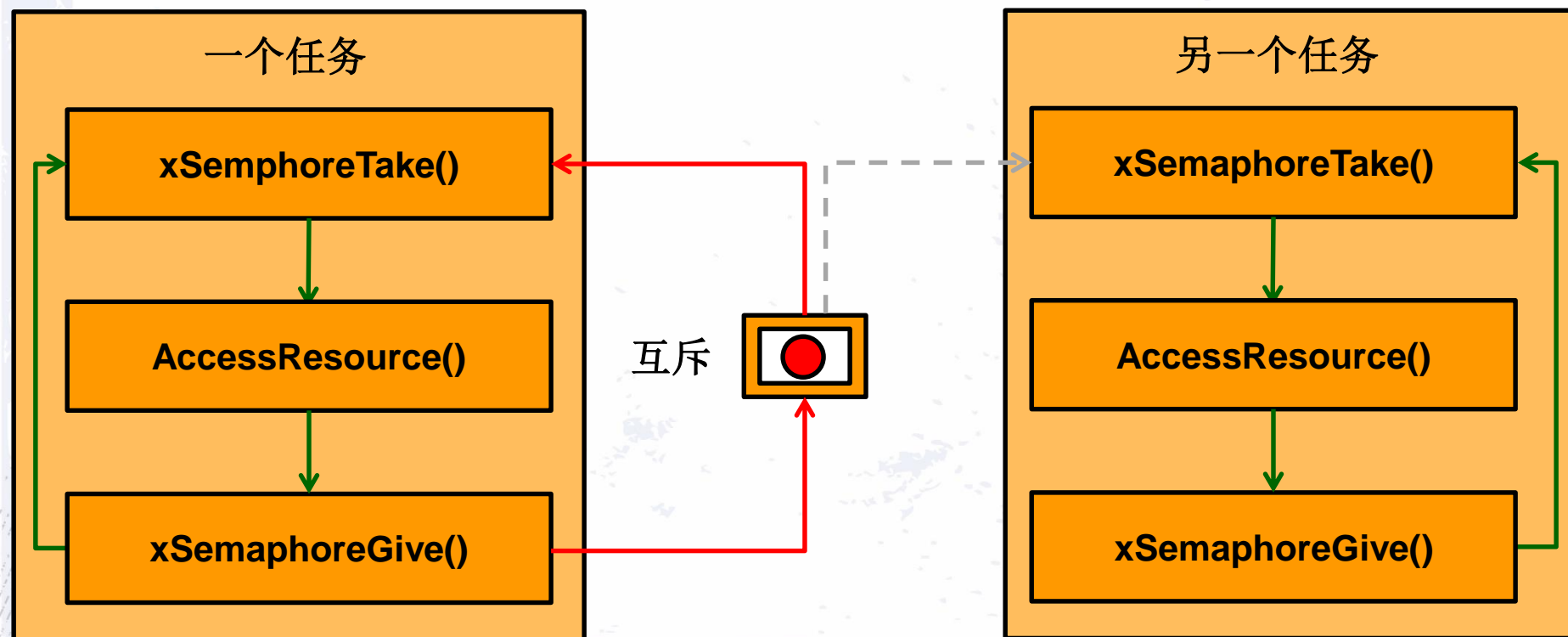
- 管理有限的资源
- 计数事件





# 互斥器

- 互斥
- 保护资源



# 小测试

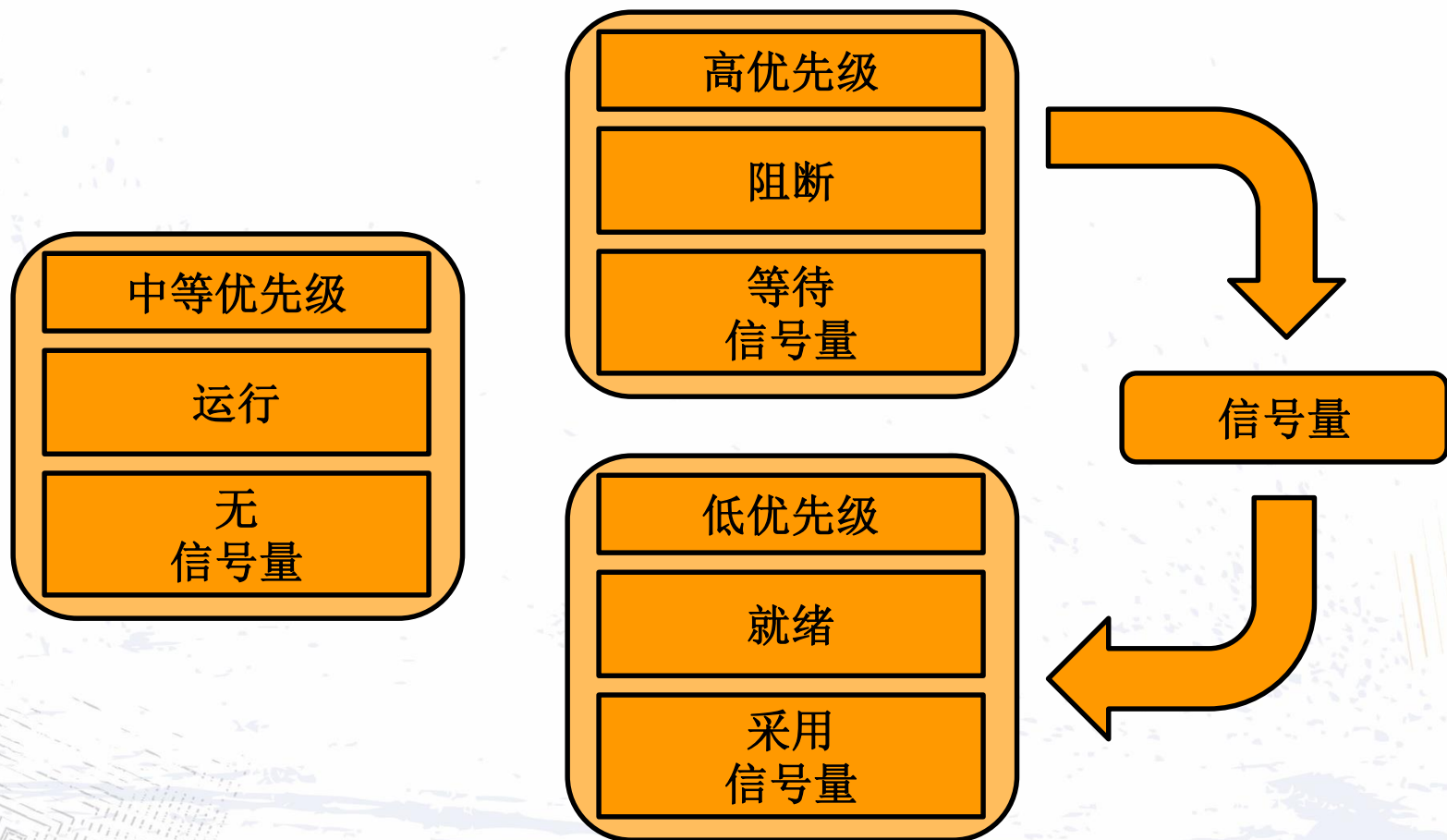
- 为什么RTOS环境中需要信号量？

# 信号量的危险

- 信号量的复杂相互作用
  - 优先级反转
  - 信号量死锁
- 信号量性能问题
  - 信号量抖动

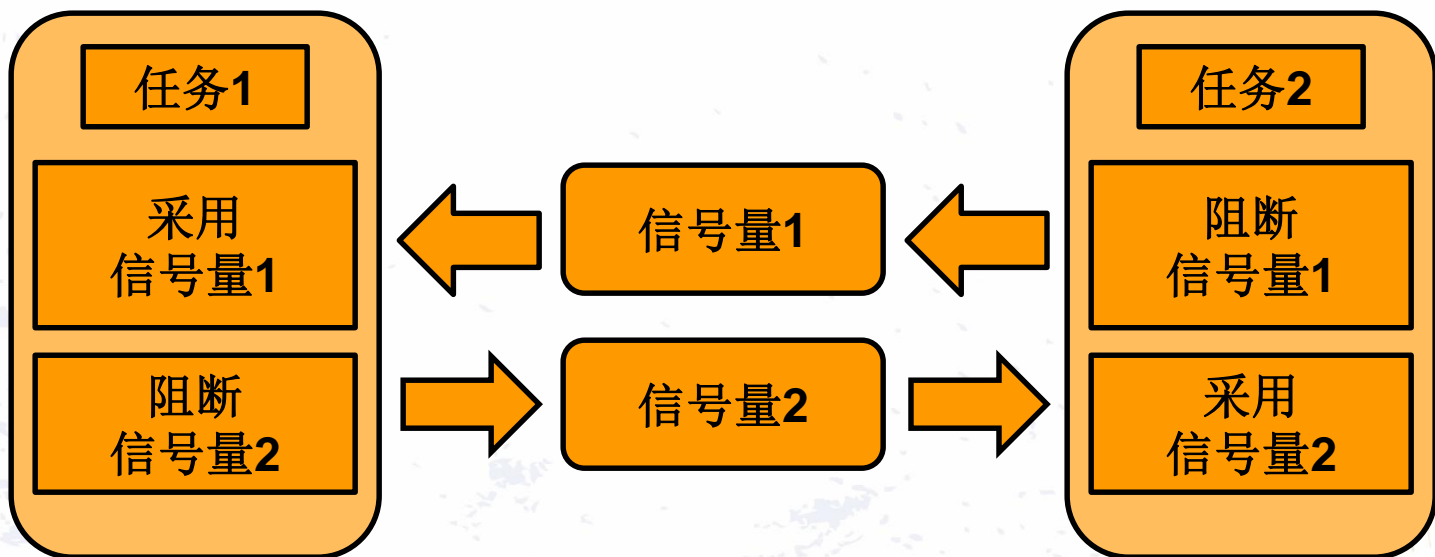
# 优先级反转

- 高优先级被低优先级阻断



# 信号量死锁

- 所有任务都在等待信号量



# 信号量抖动

- 阻断时间长于运行时间
- 通常是太多任务争用信号量访问的结果
- 导致性能限制



# 小测试

- 如何避免死锁？
- 如何避免优先级反转？

# 信号量

- **FreeRTOS API提供4种类型**
  - 二进制
  - 计数
  - 互斥
  - 递归互斥
- 同步信号量
- 互斥信号量

# 信号量API

- 创建信号量函数
  - xSemaphoreCreateBinary()
  - xSemaphoreCreateCounting()
  - xSemaphoreCreateMutex()
  - xSemaphoreCreateRecursiveMutex()
- 静态**API**创建函数

# 信号量创建

```
SemaphoreHandle_t xSemaphore = NULL;  
  
mySemaphore = xSemaphoreCreateCounting  
(  
    /* The max count */  
    3,  
  
    /* The initial count */  
    0  
);
```

# 信号量API

- 获取和释放信号量函数
  - xSemaphoreTake()
  - xSemaphoreGive()
- 递归获取和释放函数
- **FromISR**获取和释放函数

# 信号量API示例

```
// Declare the Semaphore Handle and Shared Resource
SemaphoreHandle_t xSemaphore = NULL;
int sharedResource = 0;

int StoreData(int data)
{
    // Take Semaphore
    if (xSemaphoreTake(xSemaphore, (TickType_t)10) == pdTRUE)
    {
        // Write Data to Shared Resource
        sharedResource += data;

        // Release Semaphore
        xSemaphoreGive(xSemaphore);

        return 1;
    }

    return -1;
}
```



# 小测试

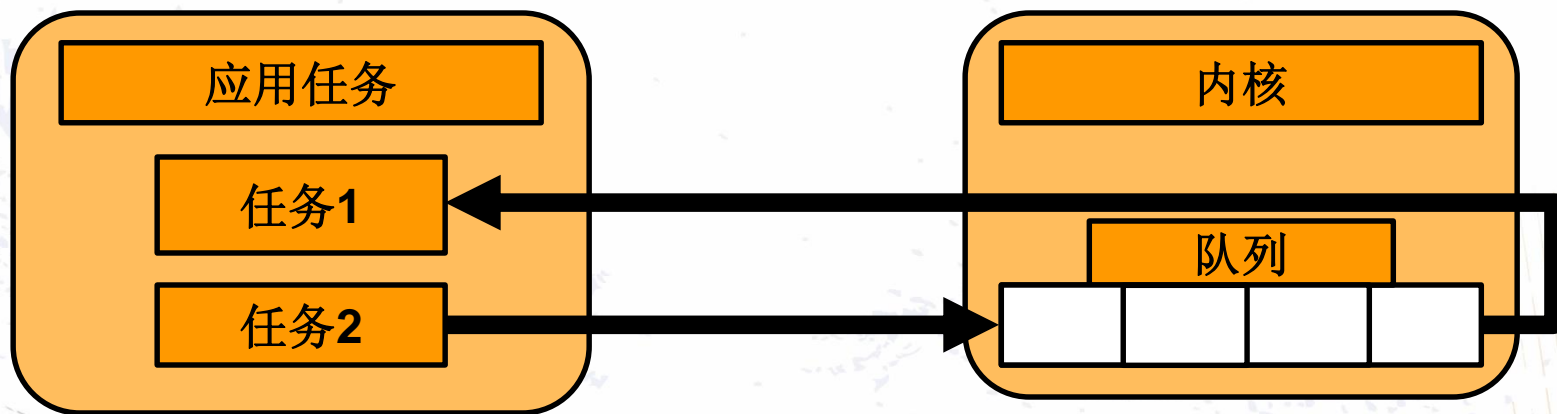
- 什么是获取信号量的**FreeRTOS API**调用？
- 什么是释放信号量的**FreeRTOS API**调用？

# 队列

- 用于管理队列的**FreeRTOS API**
- 先进先出（**FIFO**）
- 在任务之间发送报文/数据
- 可以从任务和中断访问队列
- 创建队列时，将设置每个队列项的编号和大小
- 队列保存数据的副本

# 队列

- 安全地从任何任务访问数据
- 内核负责信号量和存储器管理



# 队列API

- 创建和删除队列函数
  - xQueueCreate()
  - xQueueDelete()
- 静态创建队列函数

# 创建队列

```
QueueHandle_t myQueue;
```

```
myQueue = xQueueCreate
```

```
(
```

```
/* The length of the queue */
```

```
3,
```

```
/* The size of each item. */
```

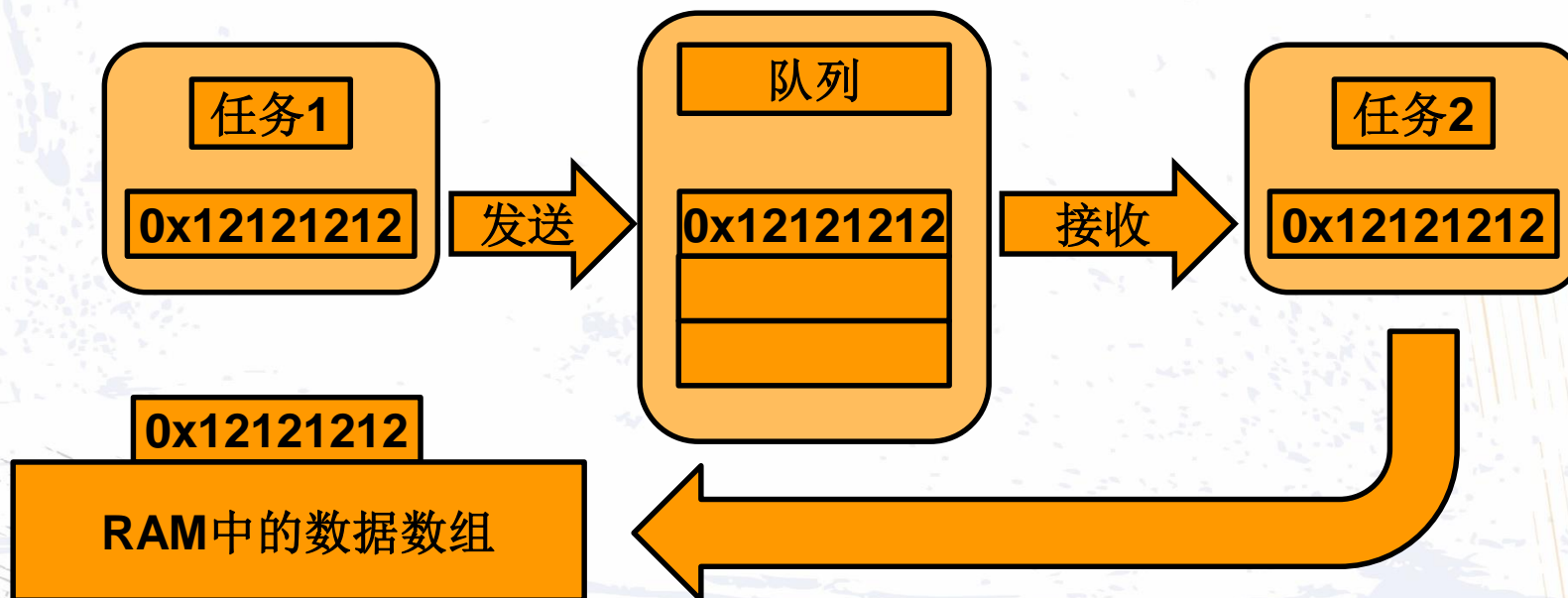
```
sizeof(uint32_t)
```

```
);
```



# 队列

- 通过复制或引用存储数据？
  - `xQueueCreate(3, sizeof(uint32_t);`
  - `xQueueCreate(3, sizeof(uint32_t *);`



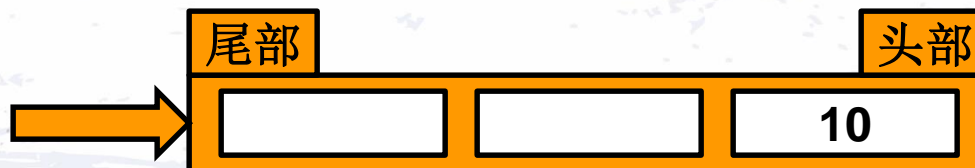


# 队列API

- 发送和接收函数
  - xQueueSendToBack()
  - xQueueSendToFront()
  - xQueueReceive()
- **FromISR**发送和接收函数

# 发送到队列

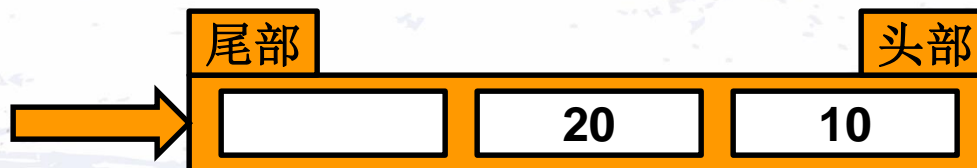
```
uint32_t ulValueToSend = 10;  
xQueueSendToBack  
(  
    /* The handle of the queue */  
    myQueue,  
  
    /* The item being sent */  
    &ulValueToSend,  
  
    /* The number of ticks to block if the queue is  
    already full. */  
    portMAX_DELAY  
);
```





# 发送到队列

```
uint32_t ulValueToSend = 20;  
xQueueSendToBack  
(  
    /* The handle of the queue */  
    myQueue,  
  
    /* The item being sent */  
    &ulValueToSend,  
  
    /* The number of ticks to block if the queue is  
    already full. */  
    portMAX_DELAY  
);
```





# 从队列接收

```
uint32_t ulValueBeingReceived;
```

```
xQueueReceive
```

```
(
```

```
/* The handle of the queue */
```

```
myQueue,
```

```
/* Buffer into which the item will be placed.*/
```

```
&ulValueBeingReceived,
```

```
/* The number of ticks to block, if the queue  
is already empty.*/
```

```
portMAX_DELAY
```

```
);
```



# 小测试

- 与信号量相比，使用队列有哪些优点？



# 演示2: 使用Tracealyzer创建互斥和监视 互斥锁定



# 演示2——目的

## 观察互斥争用如何影响任务执行和时序

- 观察：
  - 任务在等待互斥时阻断

# 演示2——目标

- 在**FreeRTOS**中创建互斥
- 使用互斥访问任务之间的共享数据
- 使用**Tracealyzer**监控互斥锁定

## 演示2——总结

- 我们创建了一个互斥
- 我们创建了一个共享数据变量
- 我们创建了用于修改变量的任务
- 我们使用**Tracealyzer**观察了互斥争用

# 课程安排

- 实时系统
- FreeRTOS简介
- 演示1（任务创建和行为）
- 任务交互
- 演示2（使用互斥）
- **FreeRTOS和PIC32架构**
- 演示3（高级调试）

# 课程安排

- **FreeRTOS和PIC32架构**
  - 任务切换
  - 中断
  - 存储器

# 任务的上下文

- 堆栈
- 堆栈指针
- 通用寄存器
- **CP0**寄存器
- **FPU**寄存器
- **DSP**寄存器
- 程序计数器



# FreeRTOS节拍

- **FreeRTOS节拍是一个周期定时器**
- **Harmony使用Timer1中断**
- **递增时标计数器**
- **确定要运行的下一个任务**
- **将上下文切换为就绪任务**

# 上下文切换

- 调度程序在任务之间切换
- 调度程序必须保存任务的状态
- 每个任务都有自己的堆栈
- 保存和恢复任务状态
- 大约**100**条切换指令



# 小测试

- 哪些因素导致上下文切换？

# 处理中断

- 控制环需要以精确的速率运行
- 中断服务程序（**ISR**）在所有任务的上下文之外运行
- **ISR**注意事项
  - **ISR**处理程序的执行时间应尽可能短
  - 使用**FromISR** API调用

# 中断

- **FreeRTOS内核优先级**
  - `configKERNEL_INTERRUPT_PRIORITY`
- **FreeRTOS Syscall优先级**
  - `configMAX_SYSCALL_INTERRUPT_PRIORITY`

# 应用中断

- 在内核和**Syscall**中断优先级之间允许**FromISR** API调用
- 高优先级中断无法使用**FromISR**
- 高于优先级的中断不受**FreeRTOS**影响



# 中断到任务的通信

- 共享存储器和环形缓冲区：
  - ISR可以与任务级代码共享变量、缓冲区和环形缓冲区
- 信号量：
  - 允许释放信号量

# 中断到任务的通信

- 报文队列
  - ISR可以将报文发送到报文队列以接收任务
- 信号
  - ISR可以发出任务信号，从而异步调度其信号处理程序

# 小测试

- 在使用**FreeRTOS**实现中断时，我们需要注意哪些关键问题？

# 任务堆栈

- 每个任务都有自己的堆栈
- 堆栈大小在**xTaskCreate**中定义
- 堆栈在**FreeRTOS**堆中分配
- 可以选择**xTaskCreateStatic**
- 堆栈溢出保护

# 堆存储器

- **FreeRTOS管理的存储器**
- 堆大小由以下变量定义
  - configTOTAL\_HEAP\_SIZE
- 在创建以下各项时使用：
  - 任务
  - 队列
  - 互斥/信号量
  - 软件定时器
  - 事件组

# 堆存储器

- **堆API**
  - pvPortMalloc()
  - vPortFree()
- **提供5种实现**
  1. 简单而不可释放
  2. 最佳匹配
  3. 标准C malloc
  4. 支持合并的最佳匹配
  5. 与4相同，但使用不连续的存储器



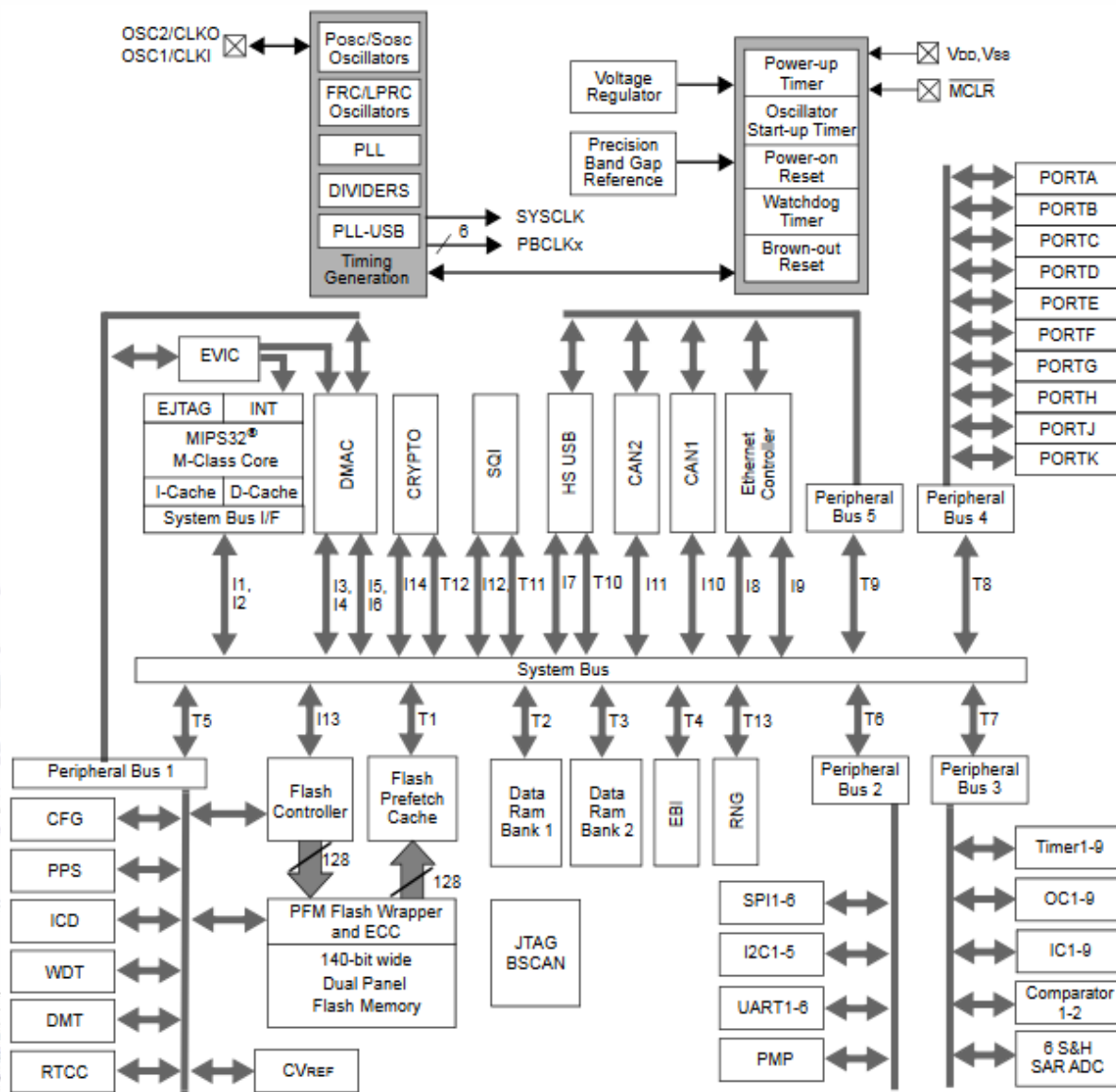
# 线程局部存储

- 每个任务的全局存储
- 对于每个任务均惟一
- 指针数组
- 大小由以下变量定义
  - configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS
- 存储器通过**pvPortMalloc**分配
- **Set和Get指针**
  - vTaskSetThreadLocalStoragePointer
  - vTaskGetThreadLocalStoragePointer

# 小测试

- **FreeRTOS在其堆中存储了哪些类型的对象？**

# PIC32MZ架构



# CPU模式

- 内核模式
  - 4 GB虚拟地址空间可用
  - CP0寄存器可用
- 用户模式
  - 2 GB虚拟地址空间可用
  - CP0寄存器访问可配置
- **FreeRTOS任务在内核模式下运行**

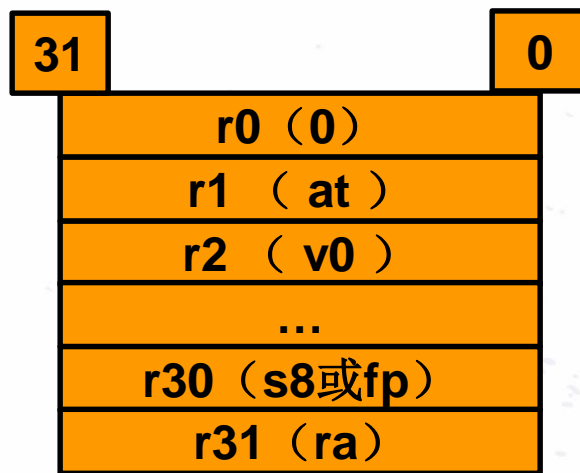
# 寻址和数据宽度

- 统一虚拟存储器寻址
- 虚拟和物理寻址
- 多总线架构
- **32位地址宽度**
- **32位数据宽度**



# 通用寄存器

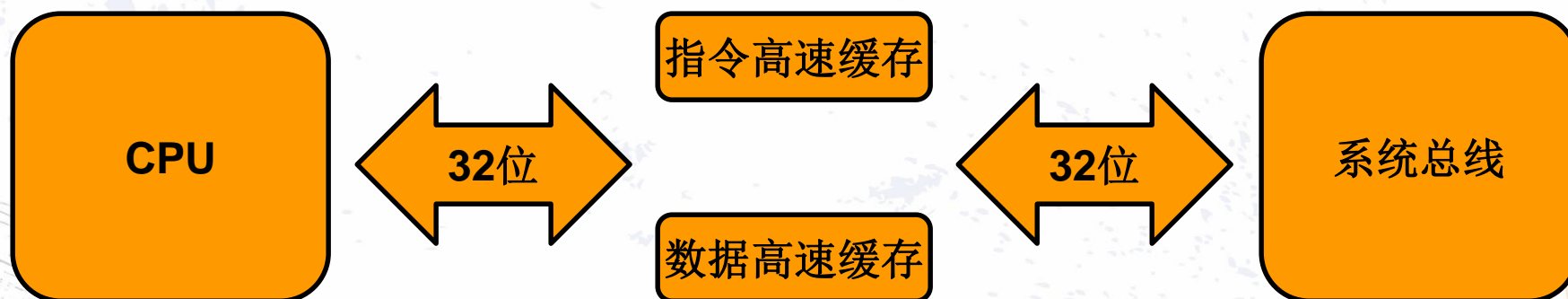
- **32 x 32位通用寄存器**
- **7个额外的影子寄存器组**
- **FreeRTOS仅使用一组**





# L1高速缓存

- **16 KB指令高速缓存和4 KB数据高速缓存**
- **高速缓存一致性**
- **KSEG0和KSEG1段**
- **保持一致性的KSEG1寻址**



# 闪存预取高速缓存

- 4个高速缓存行，每行16字节
- 一行用于**CPU**指令
- 一行用于**CPU**数据
- 两行用于外设数据
- 减少因等待状态造成的停顿





# 演示3： 使用Tracealyzer进行高级 RTOS调试

# 演示3——目的

## 调试MPLAB® Harmony FreeRTOS应用， 查找和解决时序/通信错误

- 观察：
  - 通信/时序错误如何影响任务的执行

# 演示3——目标

- 添加外设驱动程序（**UART**等）
- 编译并运行给定的应用程序文件
- 分析跟踪输出以确定应用错误
- 修复代码中的错误并利用跟踪验证结果是否正确






# 演示3——总结

- 将**Harmony**驱动程序添加到项目中
- 使用跟踪修复**FreeRTOS**应用中的代码错误



# MPLAB® Harmony中的FreeRTOS

- **Harmony中的FreeRTOS演示**

(C:) ▸ microchip ▸ harmony ▸ v2_05_01 ▸ apps ▸ rtos ▸ freertos ▸		
Share with ▾      New folder		
Name	Date modified	Type
 basic	3/23/2018 9:12 AM	File folder
 basic - Copy	5/29/2018 10:31 AM	File folder
 cdc_com_port_dual	3/23/2018 9:12 AM	File folder
 cdc_msd_basic	3/23/2018 9:12 AM	File folder
 tcpip_client_server	3/23/2018 9:12 AM	File folder

# 常见问题解答

- 支持的器件？
- 何时使用**RTOS**？
- 要使用**FreeRTOS**，单片机的最小**RAM**和**ROM**大小应是多少？

# 常见问题解答

- 最低**CPU**频率应是多少？
- 最小堆栈和堆大小应是多少？
- 如何确定任务的堆栈大小？

# 课程总结

- 使用**Harmony**框架创建**FreeRTOS**应用
- 将跟踪记录器库与**MPLAB® X IDE**项目相集成
- 确定实时应用中的时序/通信错误

# 开发工具

- **Explorer 16/32开发板**
  - **DM240001-2**
- **PIC32MZ EF接插模块**  
**(PIC32MZ2048EFH100)**
  - **MA320019**

# 更多资源

- **FreeRTOS**
  - “FreeRTOS Kernel Developer Guide”  
(<https://docs.aws.amazon.com/freertos-kernel/latest/dg/freertos-kernel-dg.pdf#about>)
  - <https://aws.amazon.com/freertos/>
  - <https://freertos.org/>
- **Tracealyzer**
  - <https://percepio.com/tz/freertos/trace/>



# 更多资源

- **MPLAB® Harmony**软件框架
  - Microchip开发人员帮助网站  
(<http://microchipdeveloper.com/harmony:start>)
  - Harmony帮助文档  
(C:\microchip\harmony\v2\_06\doc\help\_harmony.htm)
  - Harmony演示  
(C:\microchip\harmony\v2\_06\apps)

# 法律声明

## 软件：

Microchip软件仅允许用于Microchip产品。此外，Microchip软件的使用受软件附带的版权声明、免责声明以及任何授权许可的限制，无论这些内容是在安装各个程序时阐明还是在头文件或文本文件中公告。

尽管有上述限制，但Microchip和第三方提供的软件的某些组件仍可能被“开源”软件许可覆盖，其中包括要求分发者提供软件源代码的许可。在开源软件许可要求的范围内，许可条款将起主导作用。

## 注意事项和免责声明：

这些材料和随附信息（例如，包括任何软件以及对第三方公司和第三方网站的引用）仅供参考，并且按“现状”提供。Microchip对第三方公司做出的声明或第三方可能提供的材料或信息不承担任何责任。

MICROCHIP不承担任何形式的保证，无论是明示的、暗示的或法定的，包括有关无侵权性、适销性和特定用途的暗示保证。在任何情况下，对于与MICROCHIP或其他第三方提供的材料或随附信息有关的任何直接或间接的、特殊的、惩罚性的、偶然的或间接的损失、损害或任何类型的开销，MICROCHIP概不承担任何责任，即使MICROCHIP已被告知可能发生损害或损害可以预见。请注意，使用此处所述的知识产权时可能需要第三方许可。

## 商标：

Microchip的名称和徽标组合、Microchip徽标、AnyRate、AVR、AVR徽标、AVR Freaks、BitCloud、chipKIT、chipKIT徽标、CryptoMemory、CryptoRF、dsPIC、FlashFlex、flexPWR、Heldo、JukeBlox、KeeLoq、Kleer、LANCheck、LINK MD、maXStylus、maXTouch、MediaLB、megaAVR、MOST、MOST徽标、MPLAB、OptoLyzer、PIC、picoPower、PICSTART、PIC32徽标、Prochip Designer、QTouch、SAM-BA、SpyNIC、SST、SST徽标、SuperFlash、tinyAVR、UNI/O及XMEGA均为Microchip Technology Inc.在美国和其他国家或地区的注册商标。

ClockWorks、The Embedded Control Solutions Company、EtherSynch、Hyper Speed Control、HyperLight Load、IntelliMOS、mTouch、Precision Edge和Quiet-Wire均为Microchip Technology Inc.在美国的注册商标。

Adjacent Key Suppression、AKS、Analog-for-the-Digital Age、Any Capacitor、AnyIn、AnyOut、BodyCom、CodeGuard、CryptoAuthentication、CryptoAutomotive、CryptoCompanion、CryptoController、dsPICDEM、dsPICDEM.net、Dynamic Average Matching、DAM、ECAN、EtherGREEN、In-Circuit Serial Programming、ICSP、INICnet、Inter-Chip Connectivity、JitterBlocker、KleerNet、KleerNet徽标、memBrain、Mindi、MiWi、motorBench、MPASM、MPF、MPLAB Certified徽标、MPLIB、MPLINK、MultiTRAK、NetDetach、Omniscient Code Generation、PICDEM、PICDEM.net、PICkit、PICtail、PowerSmart、PureSilicon、QMatrix、REAL ICE、Ripple Blocker、SAM-ICE、Serial Quad I/O、SMART-I.S.、SQI、SuperSwitcher、SuperSwitcher II、Total Endurance、TSHARC、USBCheck、VariSense、ViewSpan、WiperLock、Wireless DNA和ZENA均为Microchip Technology Inc.在美国和其他国家或地区的商标。

SQTP为Microchip Technology Inc.在美国的服务标记。

Silicon Storage Technology为Microchip Technology Inc.在除美国外的国家或地区的注册商标。

GestIC为Microchip Technology Inc.的子公司Microchip Technology Germany II GmbH & Co. & KG在除美国外的国家或地区的注册商标。

在此提及的所有其他商标均为各持有公司所有。

© 2018, Microchip Technology Inc.版权所有。