

# 嵌入式软件架构综述

# 嵌入式软件架构

- 分为两部分：
  - 业务逻辑
  - 实时依赖硬件的逻辑
- 抽象层将所需操作的高级请求转换为操作所需的低级命令

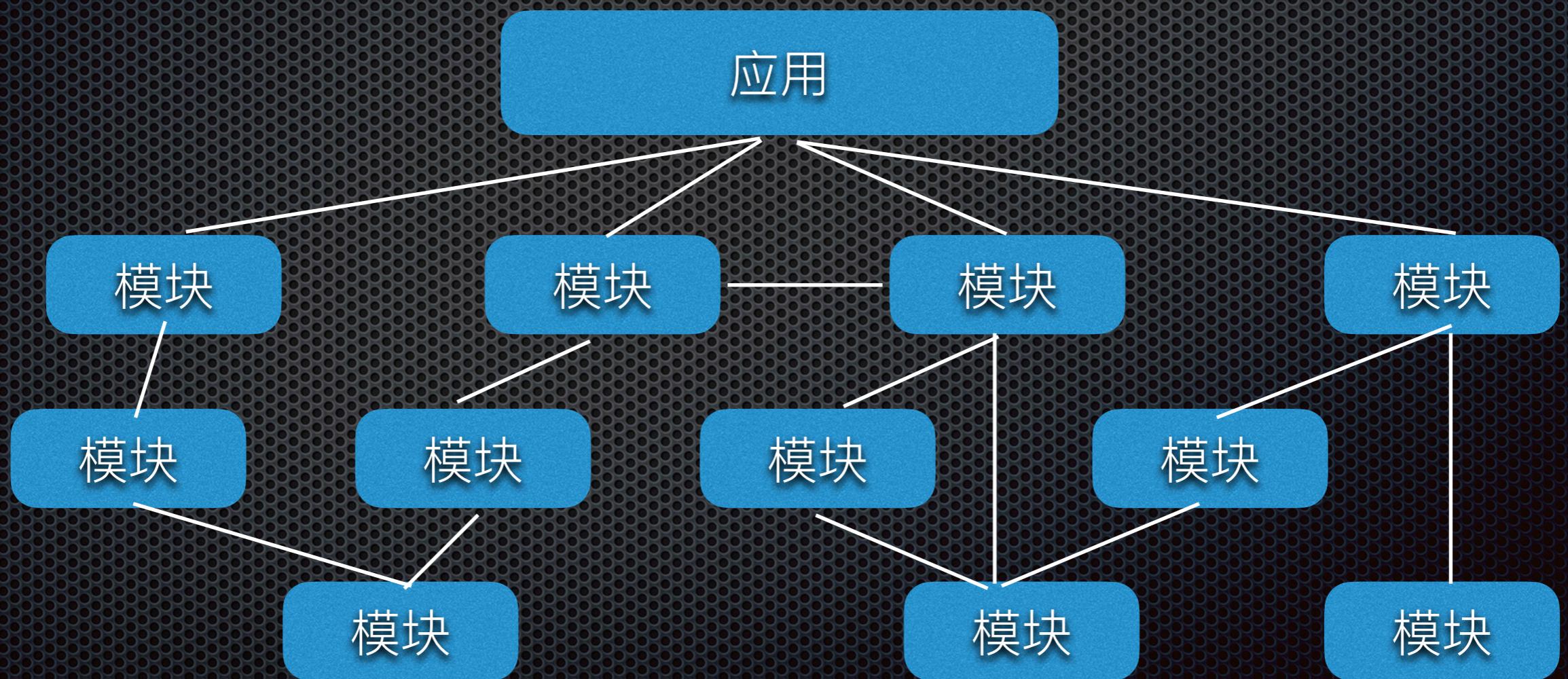


# 嵌入式软件架构模式

- 对于基于微控制器的系统来说，最常用的模式包括：
  - 非结构化单体架构
  - 分层架构
  - 事件驱动架构
  - 微服务架构

# 非结构化单体架构

- 很容易构建，但很难维持规模和移植
- 与应用层的应用程序紧密耦合



# 分层架构

- 当今嵌入式应用程序中最常用的架构
- 将应用程序的逻辑划分为若干独立的层，仅通过定义良好的抽象层进行交互
- 试图通过将应用程序分解为独立的层来改善非结构化单体架构的高耦合性



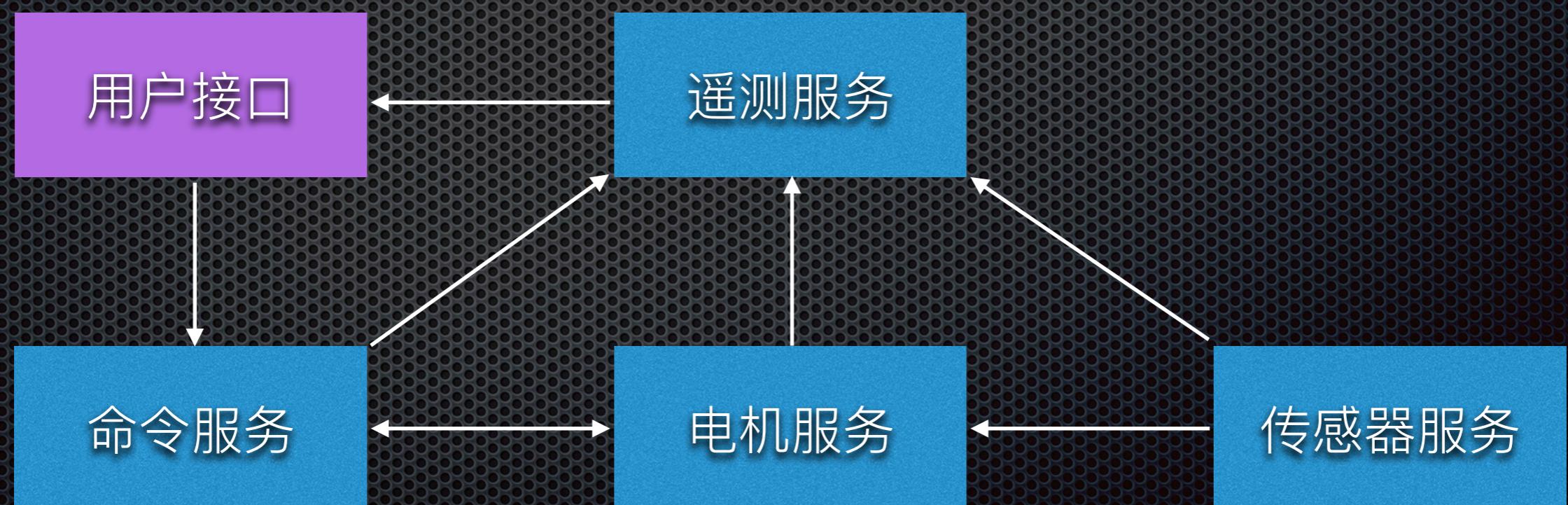
# 事件驱动架构

- 对于实时嵌入式应用程序和与能耗相关的应用程序非常有意义
- 通常利用中断来立即响应事件
- 事件驱动的体系结构通常使用消息队列、信号量和事件标志来表示系统中发生了事件。
- 优势：
  - 具有相对的可扩展性
  - 软件模块通常具有高内聚性
  - 具有低耦合性
- 缺点：无论何时需要做任何事情，都有额外的开销和复杂性



# 微服务架构

- 微服务架构将应用程序构建为为业务领域开发的小型自治服务的集合
- 微服务本质上是低耦合的，使得微服务易于维护和可测试，开发人员可以快速扩展或移植微服务
- 围绕系统的业务逻辑组织的。业务逻辑(有时称为业务功能)是系统行为的业务规则和用例



# 微服务架构的不足

- 缺点
  - 在架构上，增加设计的复杂性
  - 其次，由于具有其他体系结构中可能不需要的通信特性，它们可能会增加额外的开销和内存需求
  - 架构的分散性也意味着实时的、确定性的行为可能更具挑战性。实时和响应可能有额外的抖动
  - 可能会增加开发时间和预算

# 实时嵌入式软件常用的设计模式

- 单核
- 多核
- 发布和订阅模型
- RTOS模式
- 处理中断和低功耗设计

# 管理外设数据

- 主要的设计理念之一是数据决定设计。在设计嵌入式软件时，必须遵循数据。
- “如何将数据从外围设备获取到应用程序？”事实证明，我们可以使用几种不同的设计机制，例如
  - 轮询
  - 中断
  - 直接存储器访问(DMA)
- 这些设计机制中的每一种都有几个设计模式，可用于确保不会遇到数据丢失。

# 外设轮询

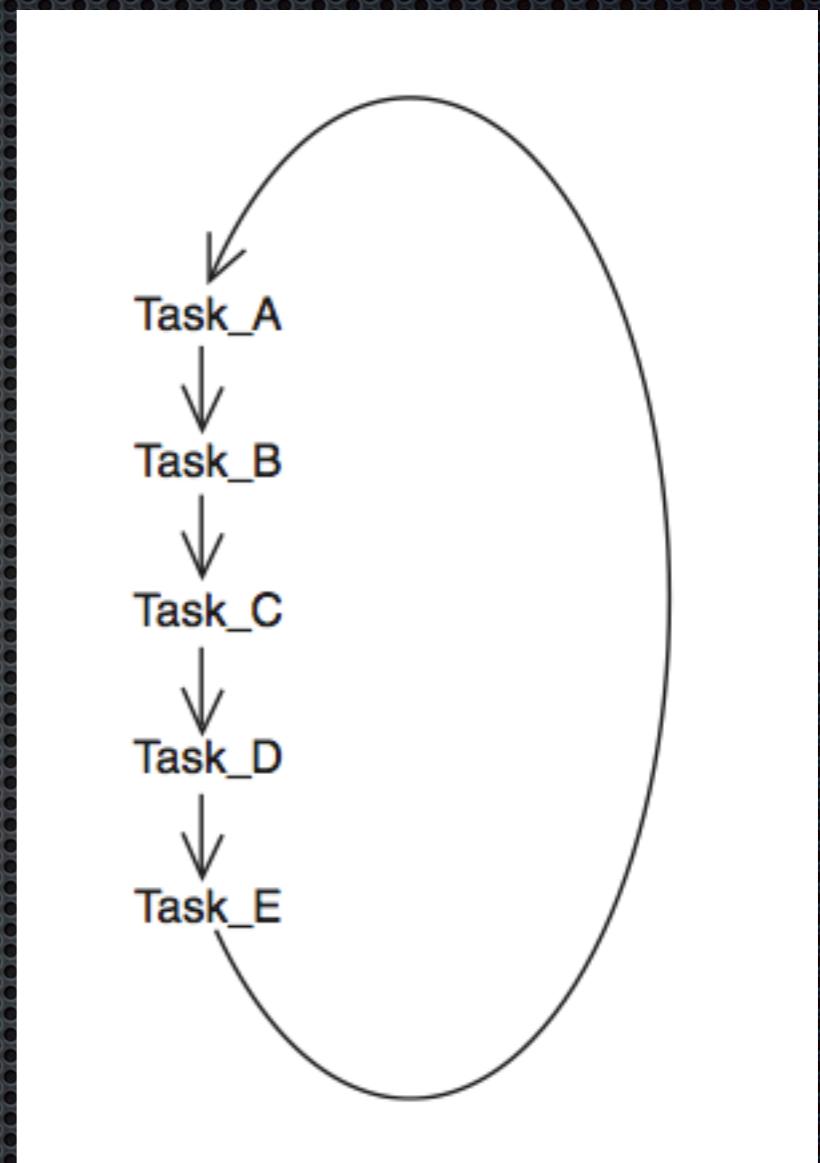
- 从外设收集数据的最直接的设计机制是让应用程序定期轮询外设，以查看是否有任何数据可供管理和处理。

# 轮询

Control Loop

Everything is a function call from the main loop

```
void main(void) {  
    while(TRUE) {  
        if (device_A requires service)  
            service device_A  
        if (device_B requires service)  
            service device_B  
        if (device_C requires service)  
            service device_C  
        ... and so on until all devices have been serviced, then start over again  
    }  
}
```



# 轮询变化

- Low priority tasks need to be slowed down

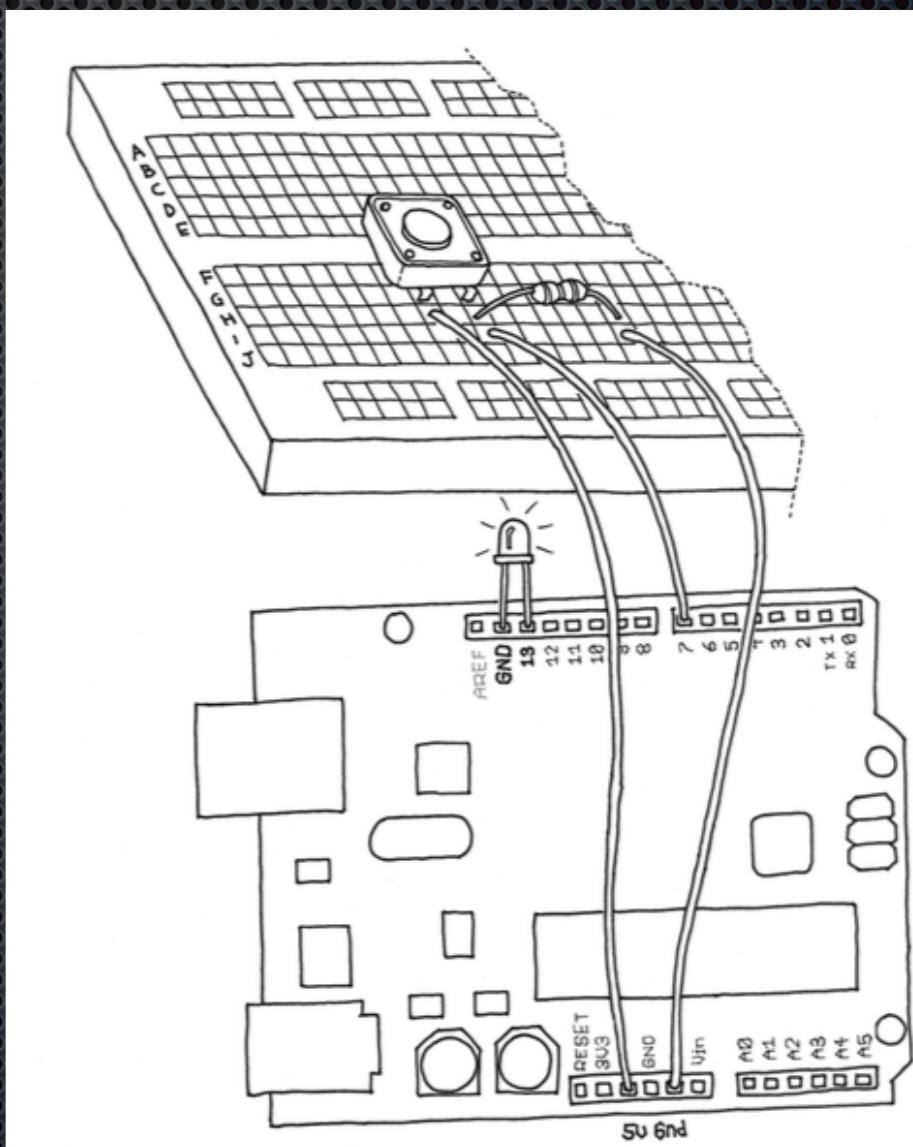
```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_A requires service)
            service device_A
        if (device_C requires service)
            service device_C
        if (device_A requires service)
            service device_A
        ...
        ...and so on, making sure high-priority device_A is always
        serviced on time
    }
}
```

# 轮询总结

- 优先级 - 无，一切按顺序运行
- 响应时间 —— 所有任务的总和
- 变化的影响 —— 显著，修改任务的执行时间或添加任务会影响所有其他任务
- 优点
  - 简单，没有共享数据问题
- 缺点
  - 浪费处理周期，在资源受限或低功耗系统中这些周期可能会显著增加
  - 在处理外设时可能会有很多抖动和延迟

# Arduino – 按下按钮时打开LED

```
const int LED = 13;      // the pin for the LED
const int BUTTON = 7; // the input pin where the
                     // pushbutton is connected
int val = 0; // val will be used to store the state
             // of the input pin
void setup() {
    pinMode(LED, OUTPUT); // tell Arduino LED is an output
    pinMode(BUTTON, INPUT); // and BUTTON is an input
}
void loop(){
    val = digitalRead(BUTTON); // read input value and store it
                               // check whether the input is HIGH (button pressed)
    if (val == HIGH) {
        digitalWrite(LED, HIGH); // turn LED ON
    }else{
        digitalWrite(LED, LOW);
    }
}
```



# Arduino – 看门狗Watchdog

```
#include <avr/wdt.h>
#define TIMEOUT WDTO_8S    // predefine time, refer avr/wdt.h
const int ledPin = 13;    // the number of the LED pin

void setup(){
    // disable the watchdog
    //wdt_disable();
    pinMode(ledPin,OUTPUT);
    // LED light once after start or if timeout
    digitalWrite(ledPin,HIGH);
    delay(1000);
    // enable the watchdog
    wdt_enable(TIMEOUT);
}

void loop(){
    // process runing
    digitalWrite(ledPin,LOW);
    delay(9000); //if timeout trig the reset
    //feed dog
    wdt_reset();
}
```

# 有限状态机

```
while(1) {  
    switch(state) {  
        case IDLE:  
            check_buttons();  
            LEDisplay_hex(NUM1);  
            if (BUTTON1 | BUTTON2 | BUTTON3)  
                state=SHOW;  
            break;  
        case SHOW:  
            NUM1=0;  
            if (BUTTON1) NUM1 += 0x0001;  
            if (BUTTON2) NUM1 += 0x0010;  
            if (BUTTON3) NUM1 += 0x0100;  
            state=IDLE;  
            break;  
    }  
}
```

# 有限状态机

- 类似于轮询，但只执行当前状态
- 每个状态决定下一个状态(非顺序执行)

# 有限状态机

- 优先级——每个状态决定下一个状态的优先级
- 响应时间——所有任务的总和
- 变化的影响——显著，修改任务执行时间或添加任务会影响所有其他任务
- 简单性——没有共享数据问题

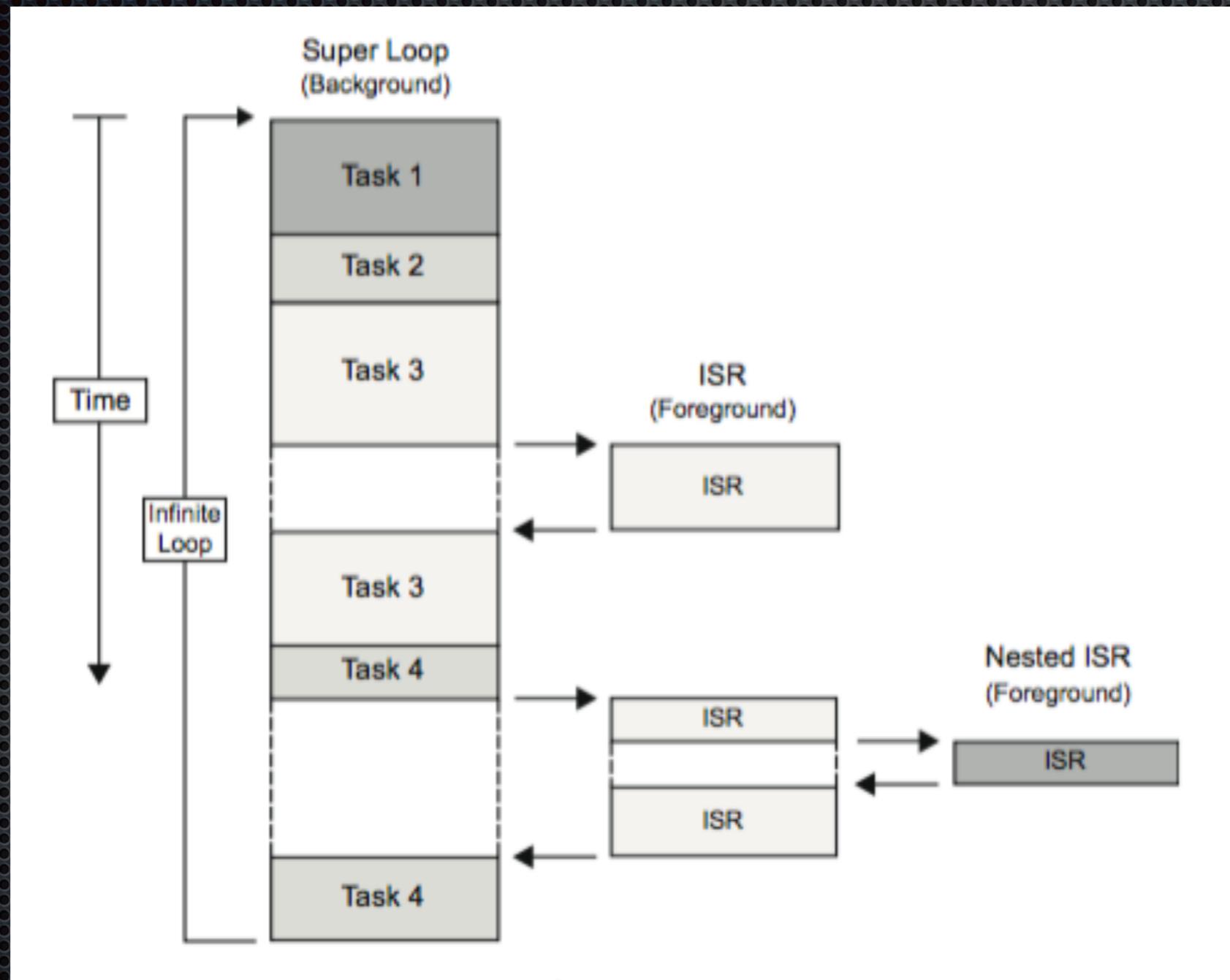
# 中断设计模式

- 中断应用程序的正常流程，以允许中断处理程序运行代码来处理系统中发生的事件
  - 例如，当数据可用、已接收或甚至已传输时，可能会触发外设的中断
- 当设计ISR时，我们希望中断尽可能快地运行(以最小化中断)
  - 避免内存分配操作，如声明非静态变量、操作堆栈或使用动态内存
  - 尽量减少函数调用，以避免时钟周期开销、不可重入函数或阻塞函数的问题

# 带有中断的轮询

```
BOOL flag_A = FALSE; /* Flag for device_A follow-up processing */
/* Interrupt Service Routine for high priority device_A */
ISR_A(void) {
    ... handle urgent requirements for device_A in the ISR,
    then set flag for follow-up processing in the main loop ...
    flag_A = TRUE;
}
void main(void) {
    while(TRUE) {
        if (flag_A)
            flag_A = FALSE
            ... do follow-up processing with data from device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all high and low priority devices have been serviced
    }
}
```

# 前后台系统



<https://doc.micrium.com/display/osiidoc/Foreground-Background+Systems>

# 带有中断的轮询

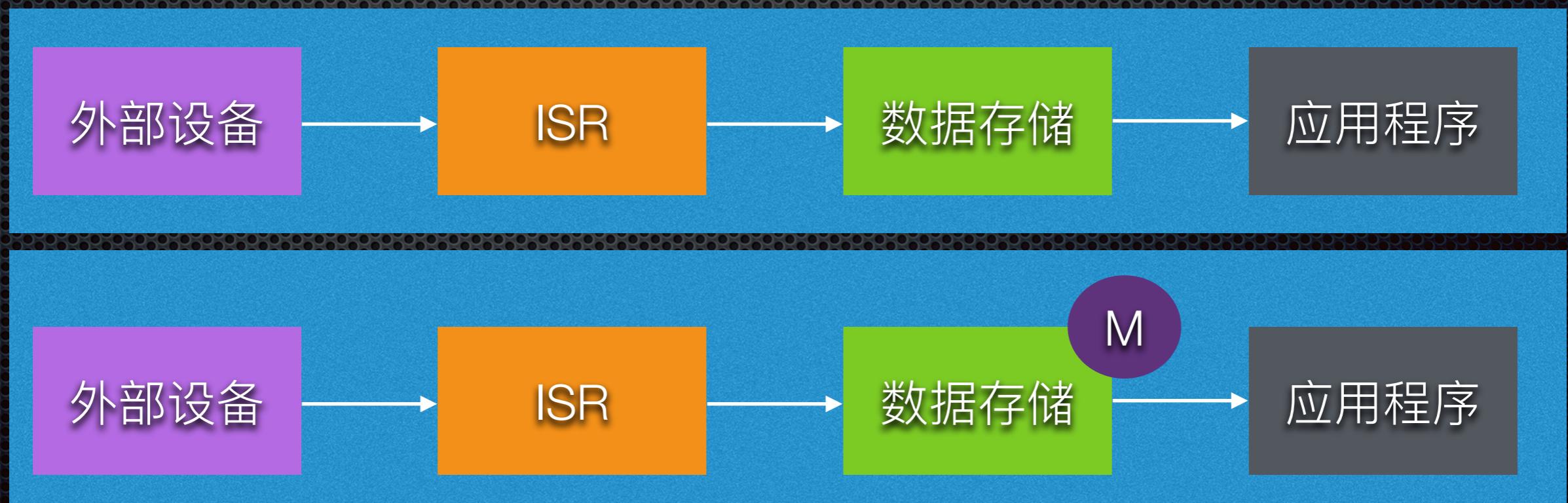
- 优先级-中断优先于主循环，中断的优先级
- 反应时间-所有任务的总和或中断执行时间
- 变更的影响-对中断服务例程的影响较小。与主循环的轮询相同。
- 共享数据——必须处理与中断服务例程共享的数据
- 优势：
  - 不需要浪费CPU周期来检查数据是否准备好
  - 获取数据的延迟是确定的
  - 抖动被最小化
- 缺点：
  - 中断的设置可能比较复杂
  - 必须小心不要使用频繁触发的中断
  - 当使用中断来接收数据时，开发人员必须仔细管理他们在ISR中所做的工作。开发人员经常需要使用ISR来处理所需的即时操作，然后将处理和非紧急的工作卸载给应用程序，从而增加了软件设计的复杂性。

# 数据获取/存储相关的中断设计模式

- 线性数据存储
- 乒乓缓冲/双缓冲
- 环形/循环缓冲区
- 带有信号量的循环缓冲区
- 带有事件标志的循环缓冲区
- 消息队列

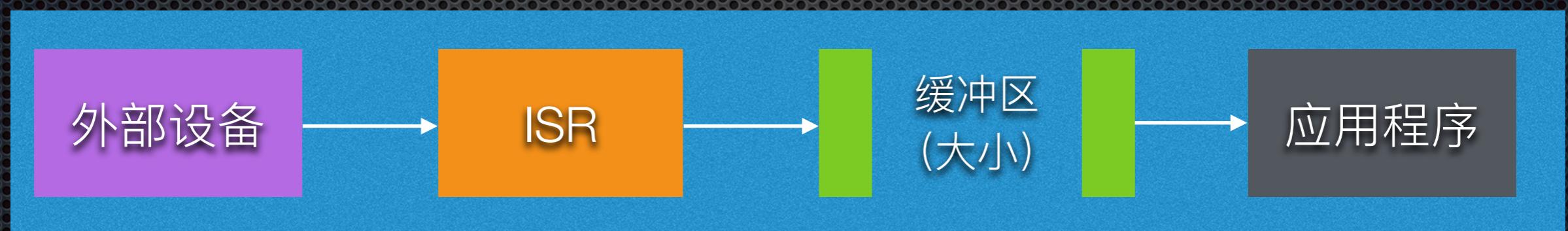
# 线性数据存储设计模式

- 中断服务程序可以直接访问的共享内存位置
- 线性数据存储可能是危险的：
  - 线性数据存储是经常遇到竞态条件的地方
  - 用于存储应用程序和ISR之间的数据的共享变量也需要声明为volatile，以防止编译器优化
- 数据存储必须由互斥锁保护，以防止竞态条件



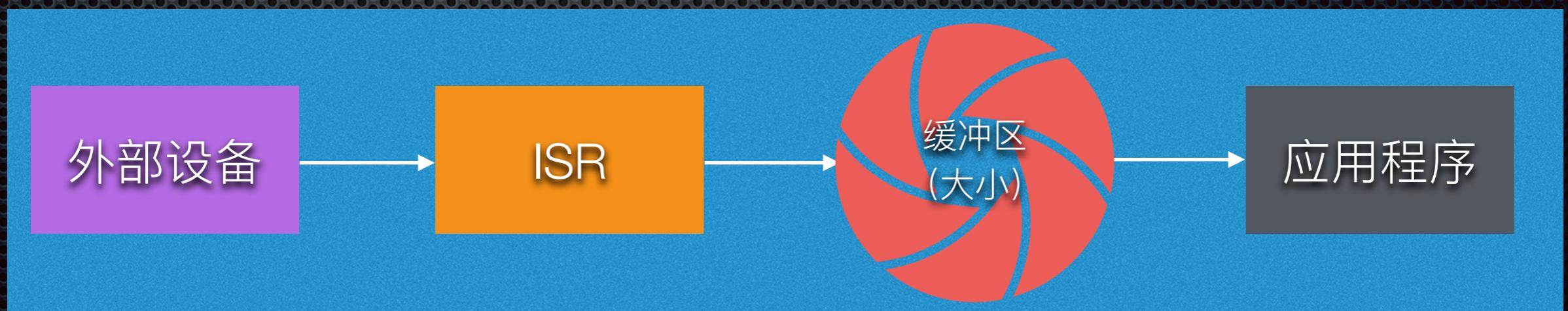
# 乒乓缓冲区/双缓冲区设计模式

- 旨在帮助缓解数据存储遇到的一些竞态条件问题



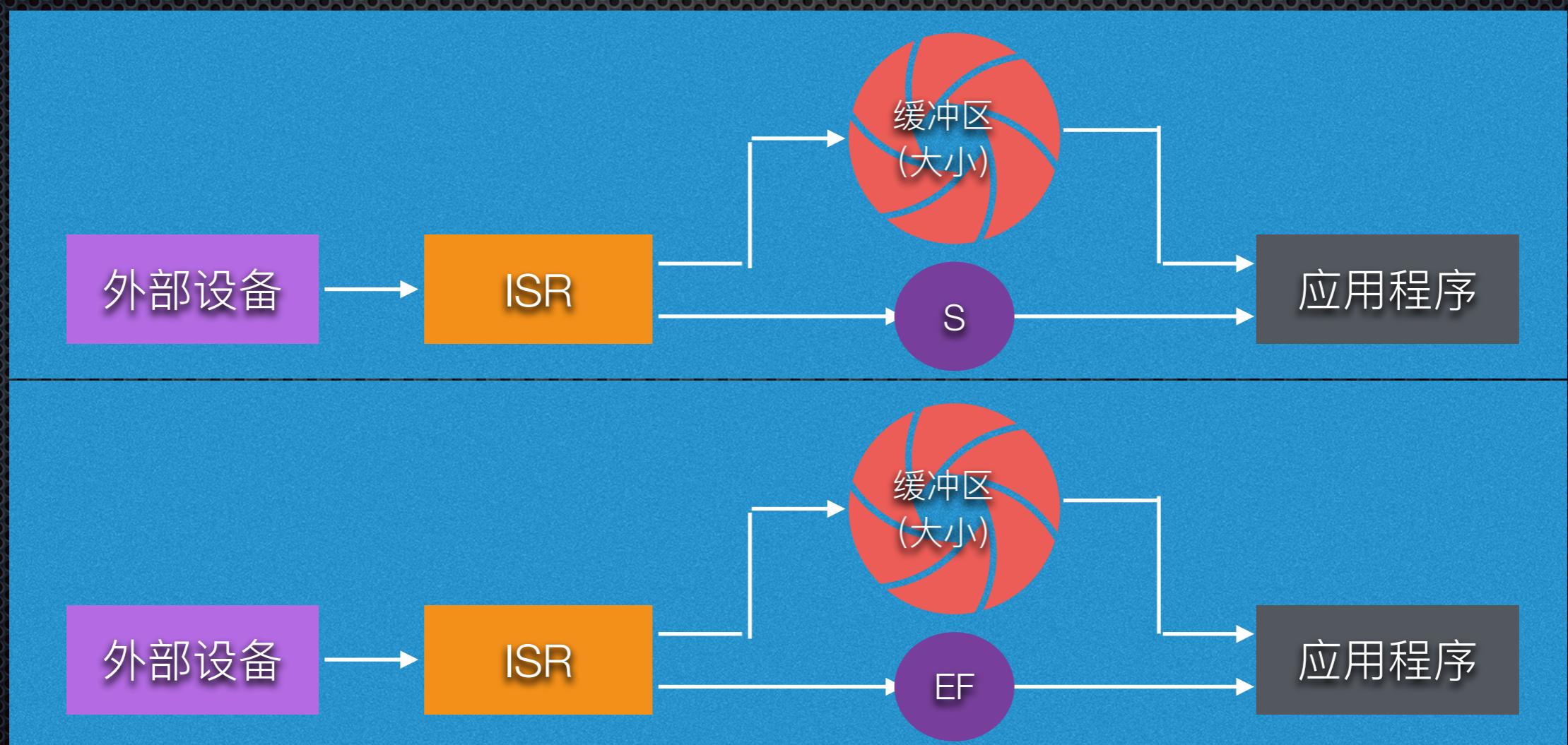
# 环形缓冲区设计模式

- 环形缓冲区（Circular Buffer），也被称为循环缓冲区（Cyclic Buffer）或者环形队列（Ring Buffer），是一种数据结构类型，它在内存中形成一个环形的存储空间
- 环形缓冲区的特点是其终点和起点是相连的，形成一个环状结构。这种数据结构在处理流数据和实现数据缓存等场景中具有广泛的应用
- 在中断中接收到的实时数据可以从外设中移除并存储在循环缓冲区中。因此，中断可以尽可能快地运行，同时允许应用程序代码自行处理循环缓冲区。使用循环缓冲区有助于确保数据不丢失，中断速度快，合理地处理数据



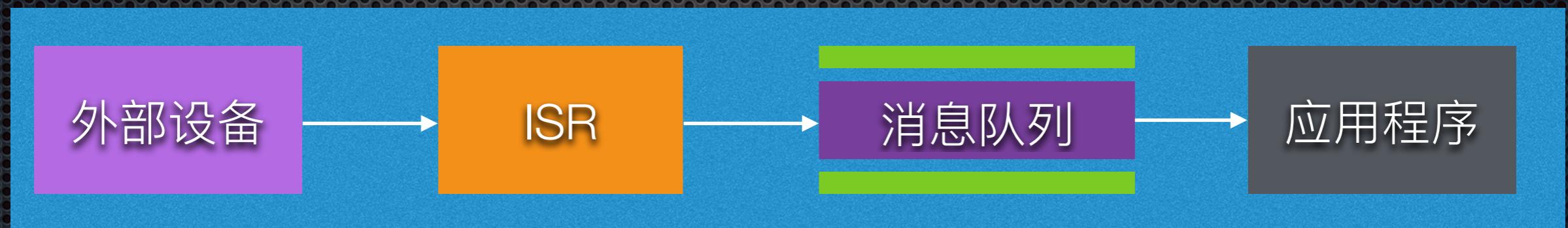
# 带有通知的循环缓冲区设计模式

- 应用程序需要轮询缓冲区以查看是否有新的可用数据
  - 信号量和事件标志
- 在大多数实时操作系统中，使用事件标志比使用信号量更有效



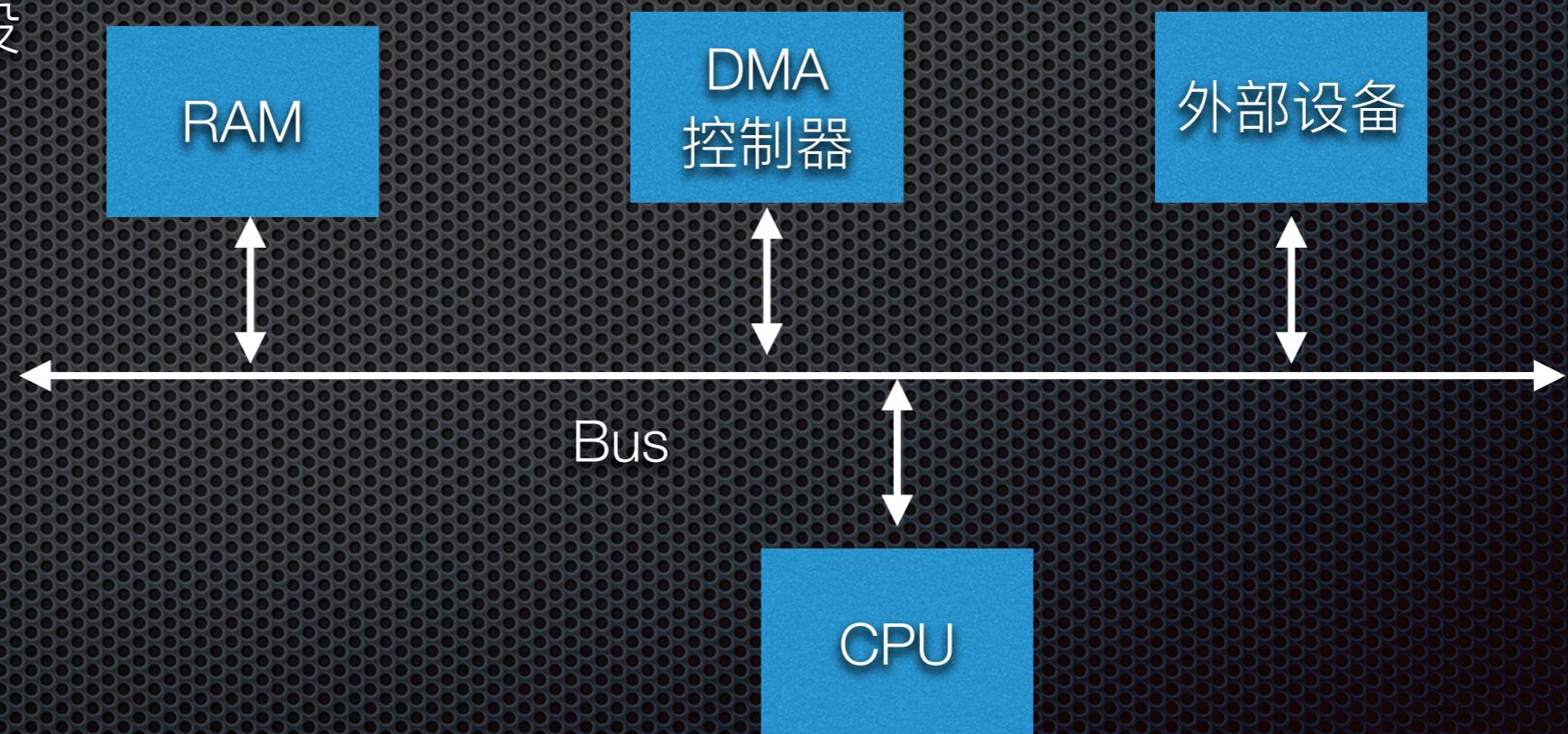
# 消息队列设计模式

- 类似于使用带有信号量的线性缓冲区
- 消息队列通常需要更多的RAM、ROM和处理能力



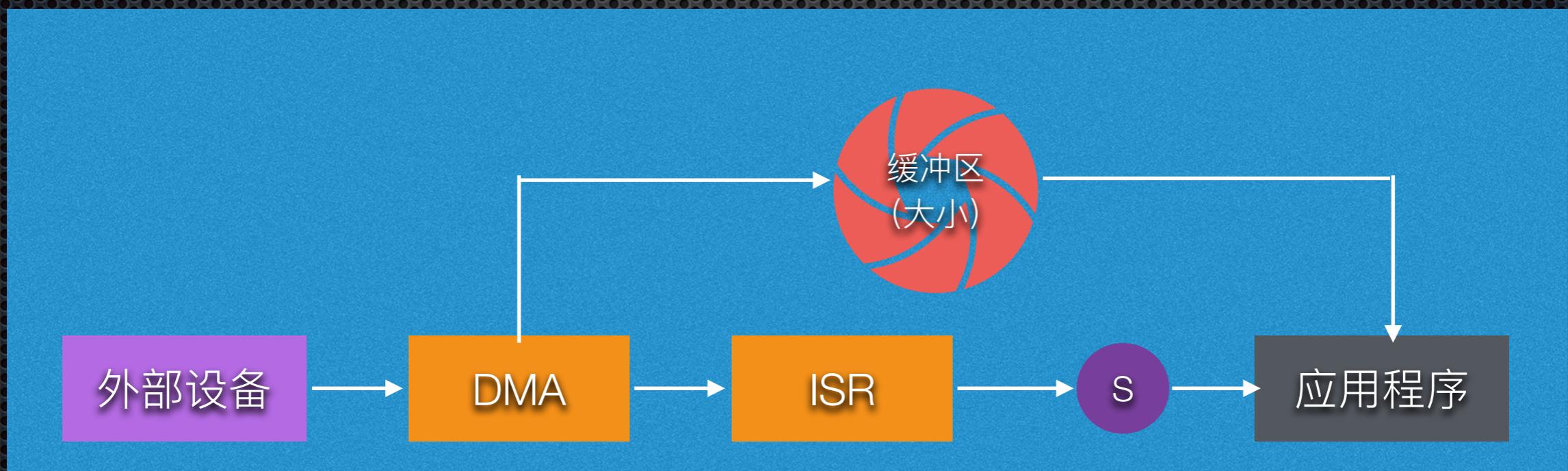
# Direct Memory Access (DMA)

- DMA，全称Direct Memory Access，即直接存储器访问
- 无需CPU的交互情况下在RAM和外设之间以及内部传输数据
  - RAM到RAM
  - 外设到RAM
  - 外设到外设



# DMA控制器将外设数据传输到循环缓冲区的设计模式

- DMA控制器可以显著提高外设和应用程序之间的数据吞吐量。此外，可以利用DMA控制器减轻CPU运行ISR来传输数据的负担，并最大限度地减少浪费的计算周期

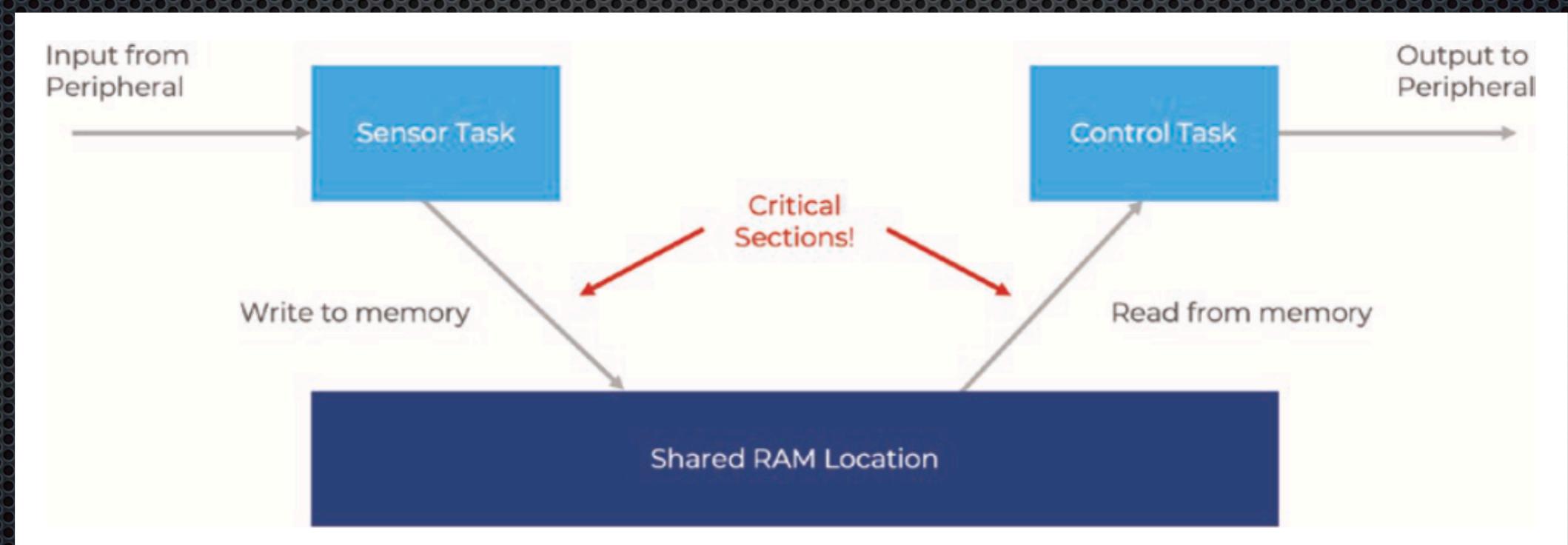


# RTOS应用程序设计模式

- 在RTOS应用程序中，通常有两种类型的同步
  - 资源同步：决定了对共享资源的访问是否安全
  - 活动同步：决定执行是否已达到特定状态
- 资源同步和活动同步设计模式有多种形式和大小。让我们更详细地探讨一些常见的同步模式。

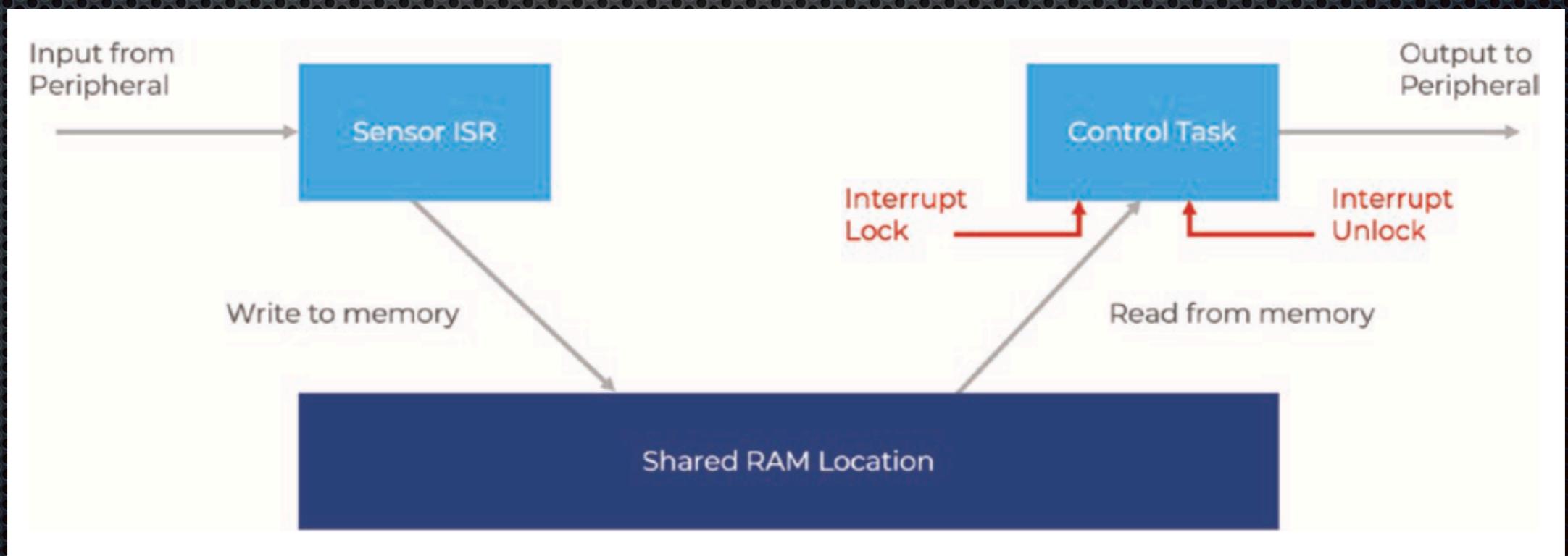
# 资源同步

- 确保需要访问资源(如内存位置)的多个任务或任务和中断以协调的方式进行，从而避免竞争条件和内存损坏。
- 可以通过三种方式处理资源同步
  - 中断锁定、抢占锁定和互斥锁



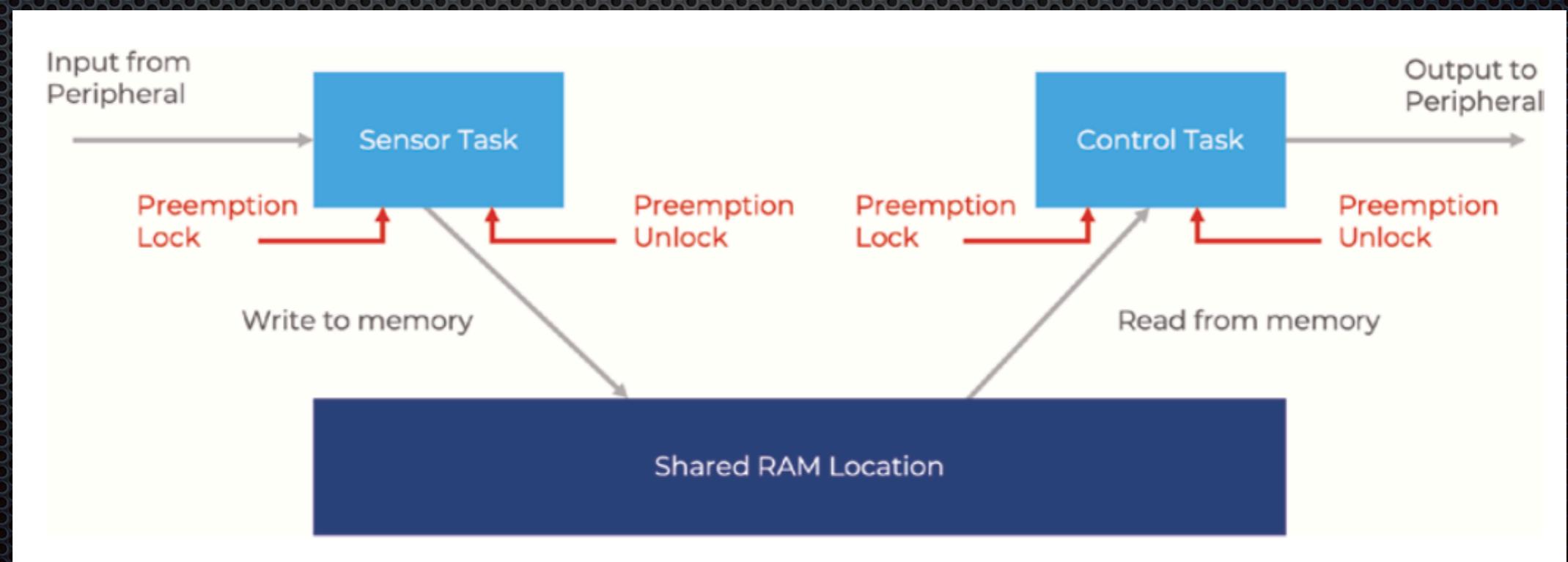
# 中断锁

- 当系统任务禁用中断以在任务和中断之间提供资源同步时，就会发生中断锁定。
- 中断锁定可能是有用的，但在实时嵌入式系统中可能会导致许多问题。例如，通过在读取操作期间禁用中断，中断可能会被错过，或者在最好的情况下，延迟执行。延迟执行会给系统增加不必要的延迟甚至抖动。



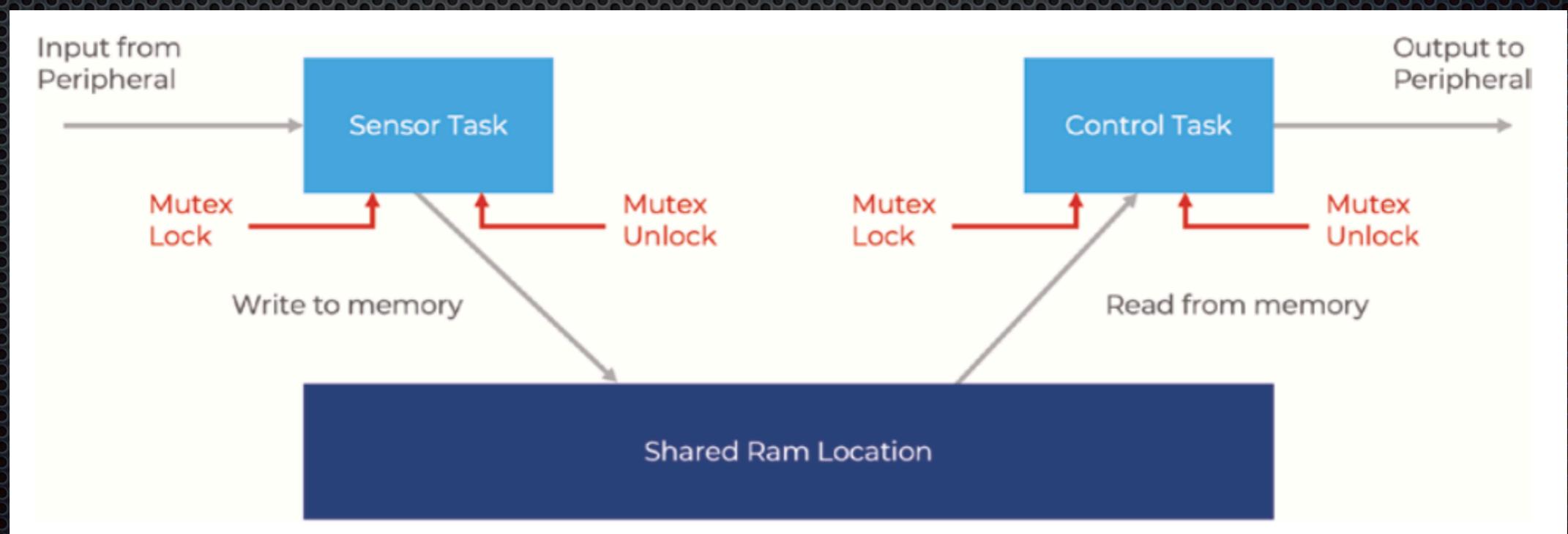
# 抢占锁

- 抢占锁可用于确保任务在执行临界区期间不间断，抢占锁在临界区期间临时禁用RTOS内核抢占调度程序
- 抢占锁是一种比中断锁更好的技术，因为关键的系统中断仍然允许运行；但是，更高优先级的任务在执行时可能会延迟。抢占锁还会给系统带来额外的延迟和抖动。还有一种可能性是，由于内核的抢占式调度器被禁用，高优先级任务的执行可能会延迟。



# 互斥锁

- 保护共享资源的最安全、最推荐的方法之一是使用互斥锁。互斥锁通过创建一个对象来保护临界区，该对象的状态可以被检查，以确定是否可以安全访问共享资源，其唯一目的是为共享资源提供互斥
- 互斥锁不会禁用中断，它不会禁用内核的抢占调度程序
- 保护共享资源的互斥锁的一个潜在问题是，开发人员需要知道它的存在！

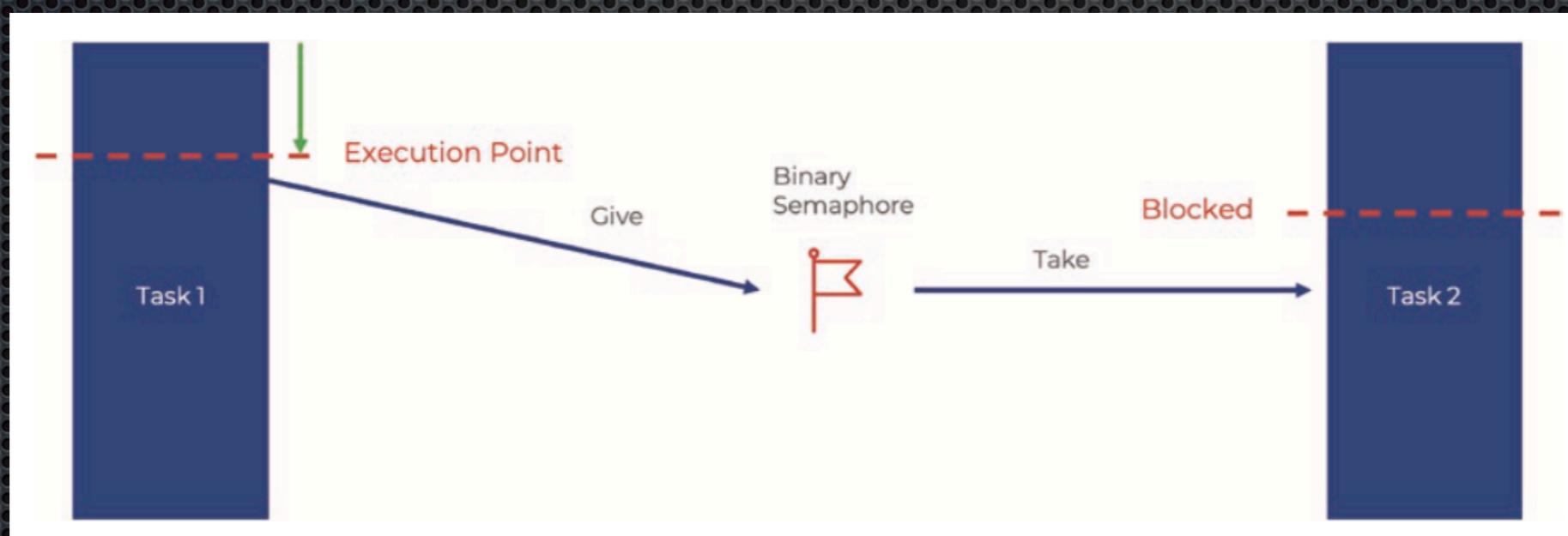


# 活动同步

- 活动同步是关于协调任务执行的。
  - 例如，假设我们正在开发一个获取传感器数据并使用传感器值驱动电机的系统。我们很可能想要向运动任务发出信号，告诉它有新的数据可用，这样任务就不会对陈旧的数据采取行动。
- 在使用实时操作系统时，可以使用许多活动同步模式来协调任务执行。

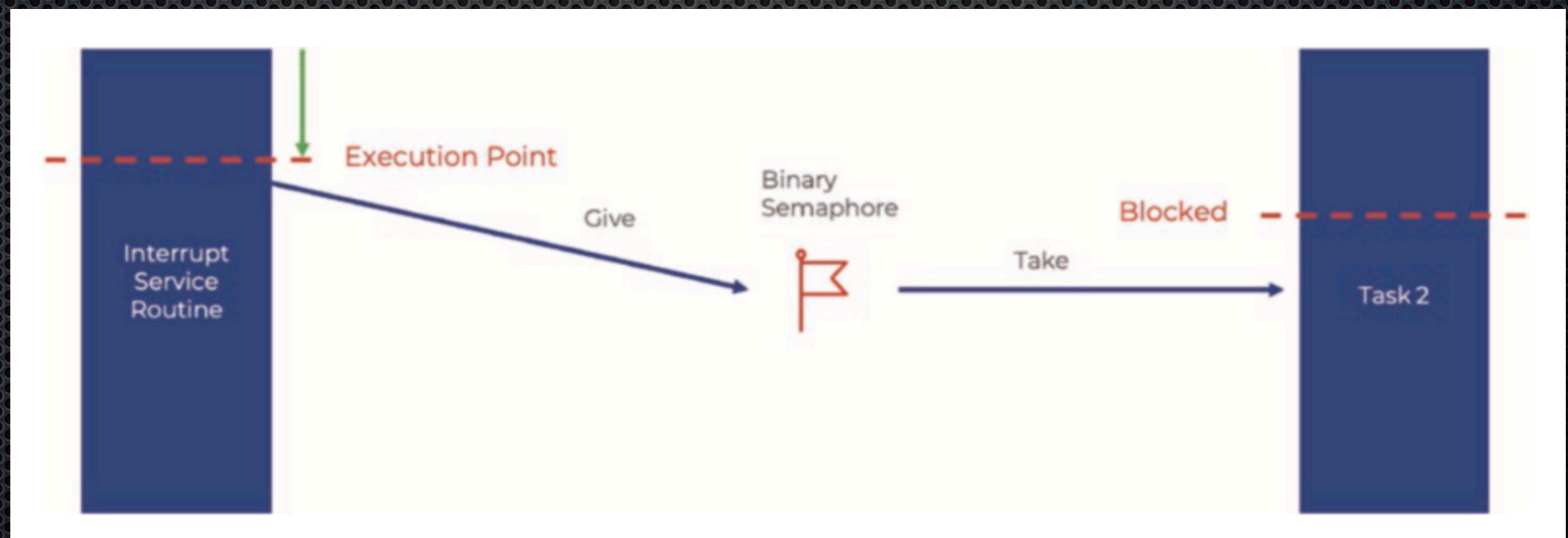
# 单向同步(任务对任务)

- 单向同步使用二值信号量或事件标志来同步任务



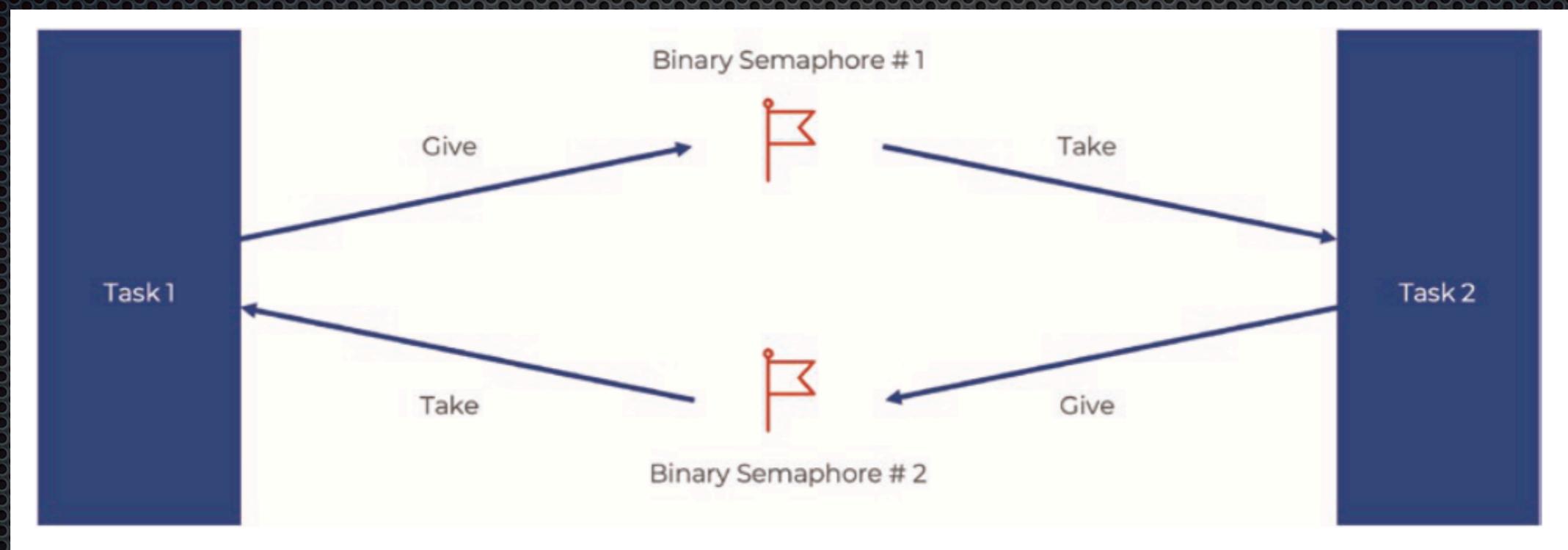
# 单向同步(中断到任务)

- 单向同步还可以在中断和任务之间同步和协调任务执行，不同之处在于，在ISR给出信号量或事件标志之后，ISR将继续执行，直到完成为止。
- 单向同步也可以与计数信号量一起使用



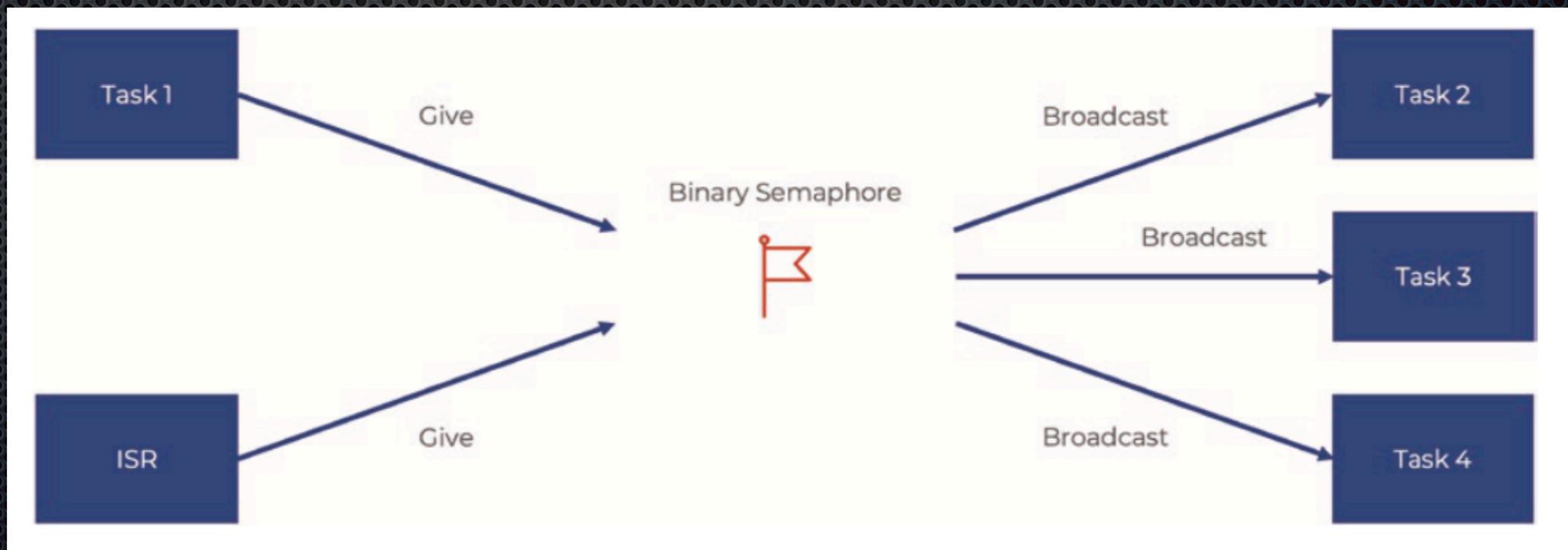
# 双向同步

- 两个任务在它们之间的两个方向上进行协调



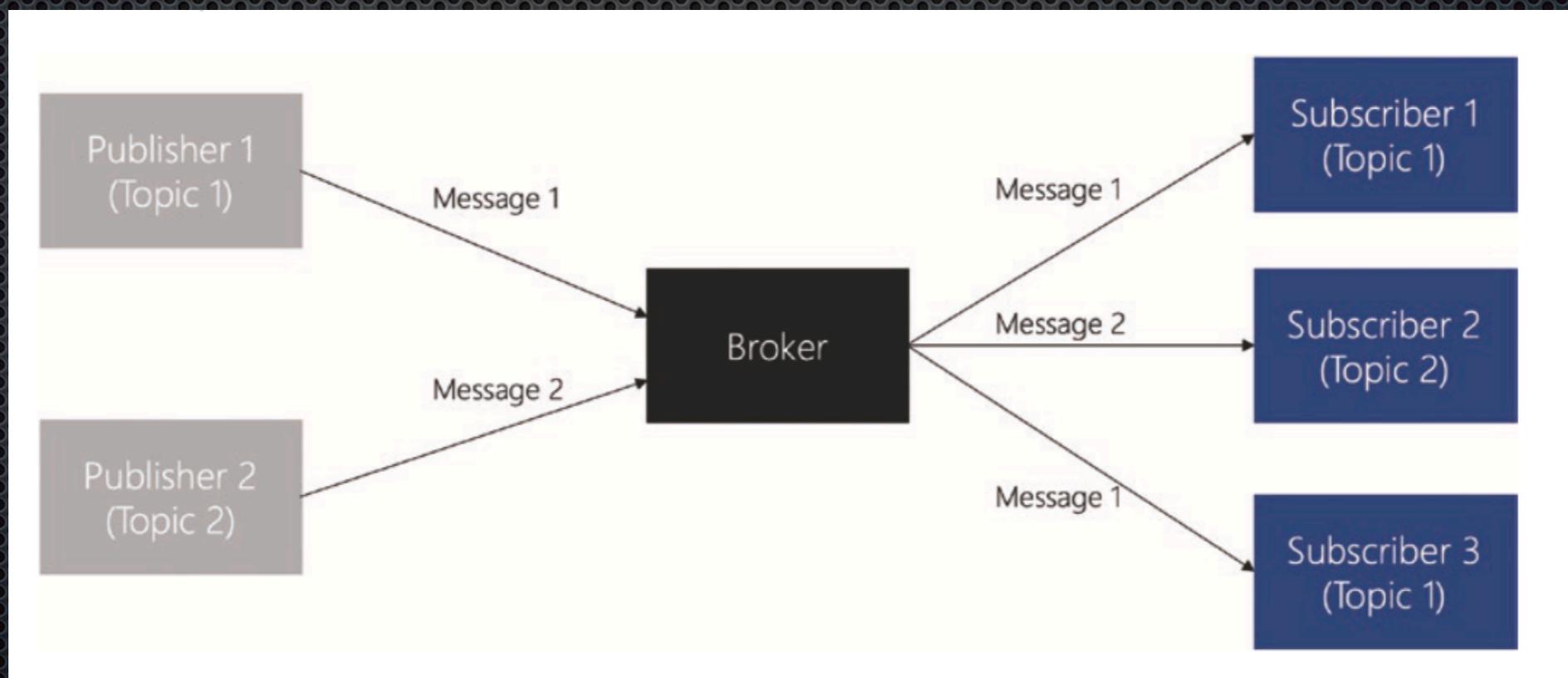
# 同步多个任务之广播设计模式

- 广播设计模式允许多个任务阻塞，直到给定信号量、出现事件标志，甚至将消息放入消息队列。
- 任务或ISR可以提供由多个任务使用的二值信号量广播。广播可能无法在所有实时操作系统中实现，必须检查RTOS是否支持。
- 如果不支持广播，则可以创建由任务或中断给出的多个信号量。从设计的角度来看，使用多个信号量并不优雅，从实现的角度来看也不高效，但有时在软件中就是这样。



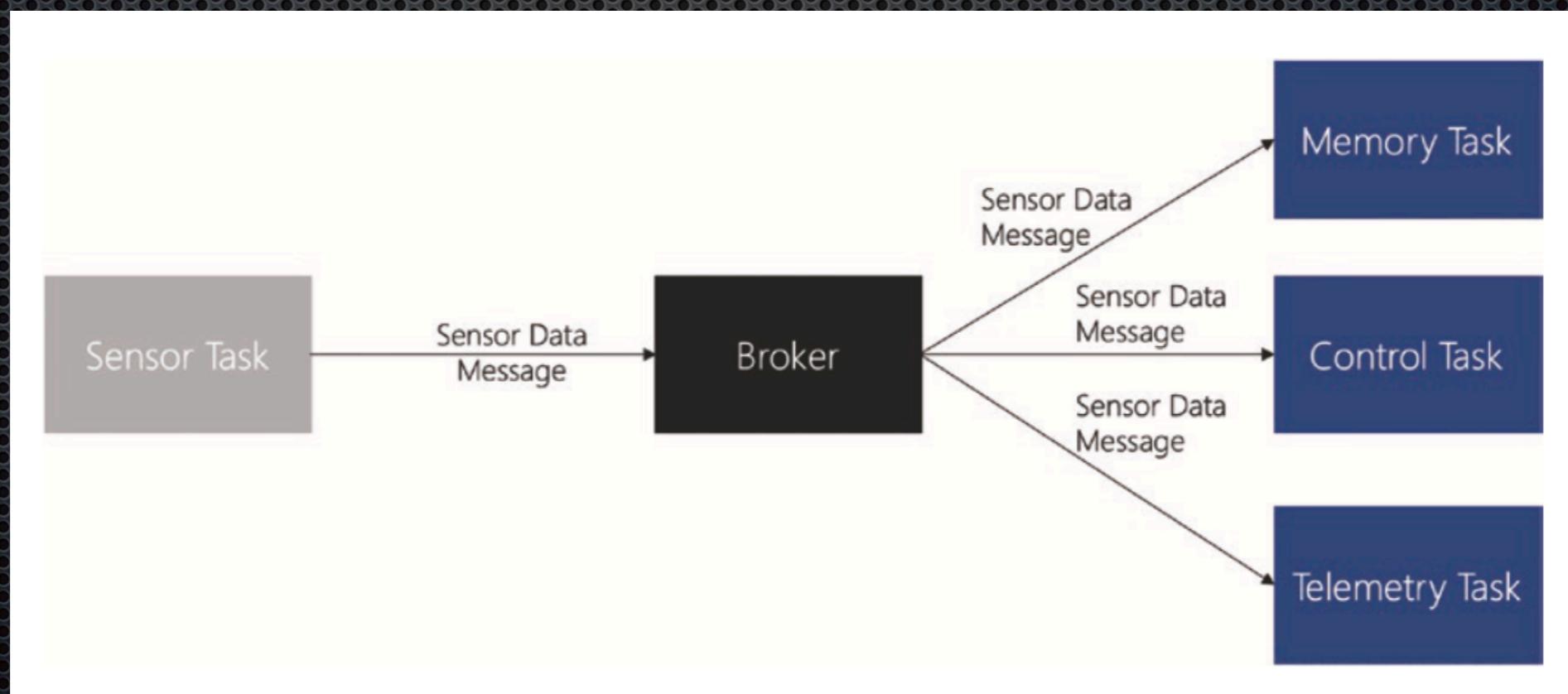
# 发布和订阅模型

- 在物联网领域广泛使用
  - 在许多情况下，物联网设备将启动电源，连接到云，然后订阅它想要接收的消息主题。该设备甚至还可以发布特定的主题。
- ROS使用
- 会导致应用程序占用更大的内存

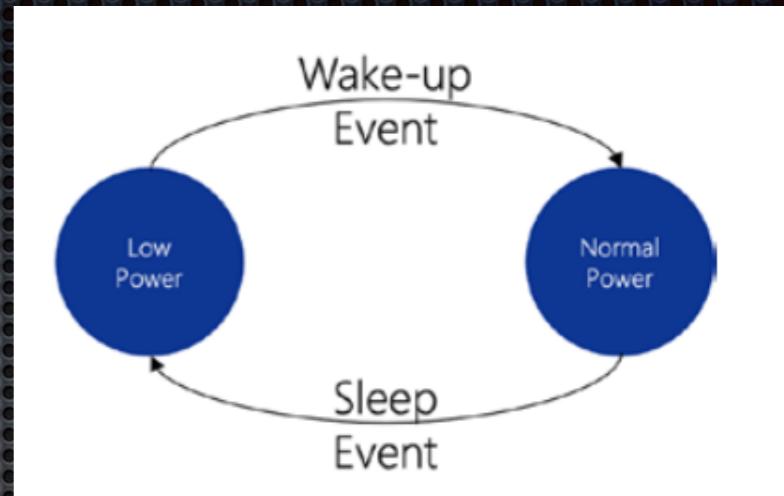


# 例子—IOT

- 打破了传感器任务和任何需要使用数据的任务之间的依赖关系
- 系统完全可扩展
  - 如果突然需要使用传感器数据进行故障处理，不必回头重新构建系统或添加另一个依赖项。相反，创建一个订阅传感器数据的故障任务。该任务将获得操作和检测故障所需的数据



# 低功耗应用程序设计模式



- 关于低功耗设计模式，主要模式是尽可能地保持设备关闭
  - 事件驱动中，事件之间没有实际工作要做时，应该将微控制器置于适当的低功耗状态，并关闭任何非必要的电子设备
  - 除非发生唤醒事件，否则系统处于低功耗状态
  - 实际工作完成后，系统恢复到低功耗状态
- 建议
  - 使用内置tickless模式的RTOS，或者可以扩展系统tick以使微控制器休眠更长  
时间的RTOS
    - 例如，FreeRTOS有一个可以启用的tickless模式。开发人员可以添加自定义代码来管理低功耗状态。当系统进入睡眠状态时，系统节拍被自动缩放，因此发生的系统节拍更少！

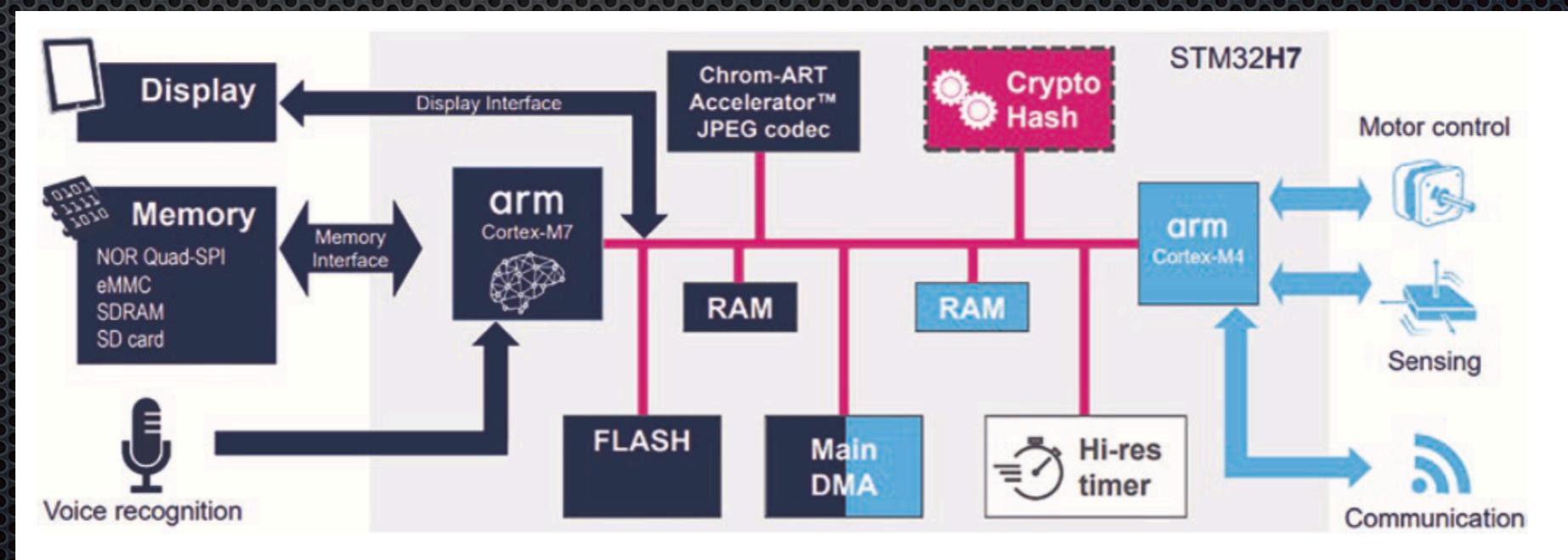
# 多核架构

- 同构多核，对称多核处理具有相同处理器架构的两个核
- 异构多核架构，每个处理核心都有不同的底层架构
  - 例如，Cypress PSoC 64具有用于用户应用程序的Arm Cortex-M4和作为安全处理器的Arm Cortex-M0+。这两个核心也不必以相同的时钟速度运行。例如，Arm Cortex-M4的工作频率为150mhz，而Arm Cortex-M0+的工作频率为100mhz。



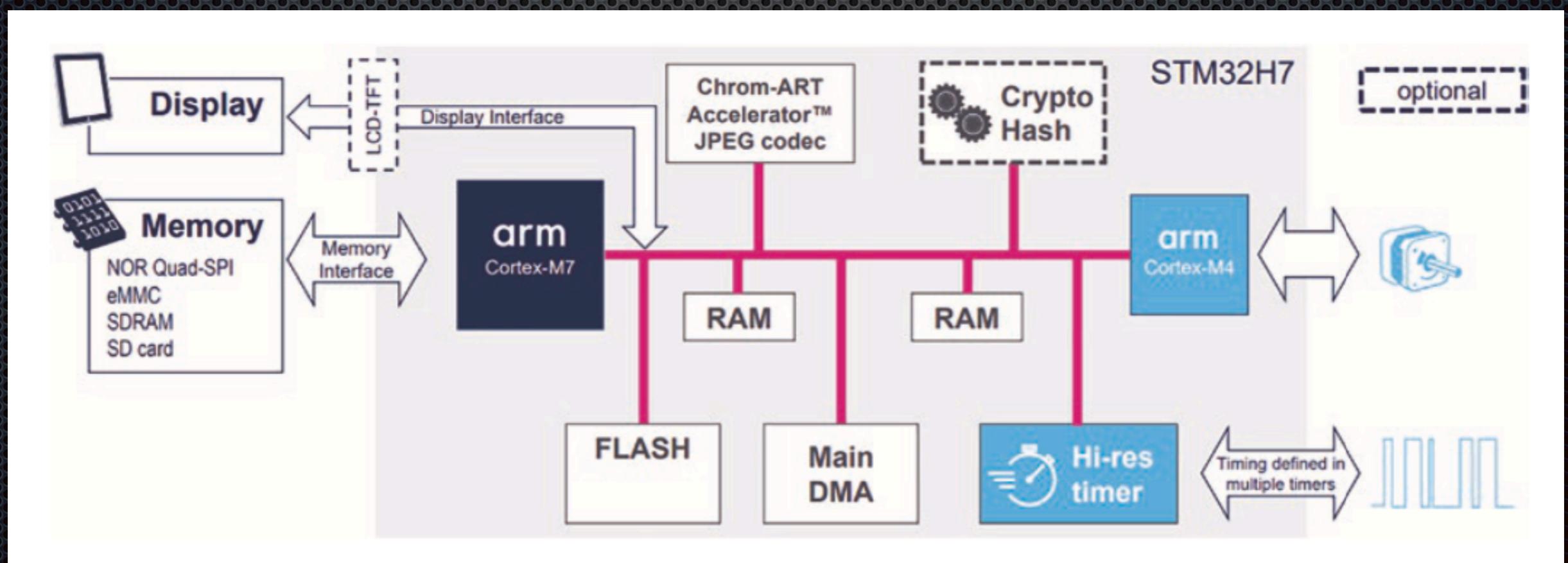
# 人工智能用例

- 在微控制器上运行机器学习推理无疑是可以做到的，但它是计算周期密集的。其理念是一个核心用于运行机器学习推理，而另一个核心用于实时控制。
- 例如，在STM32H7系列中，多核部件具有Arm Cortex-M7和Arm Cortex-M4。M7运行机器学习推理，而M4做标准的实时工作，如电机控制、传感器采集和通信。M4内核可以向M7提供运行AI算法所需的数据，M7可以向M4提供结果。



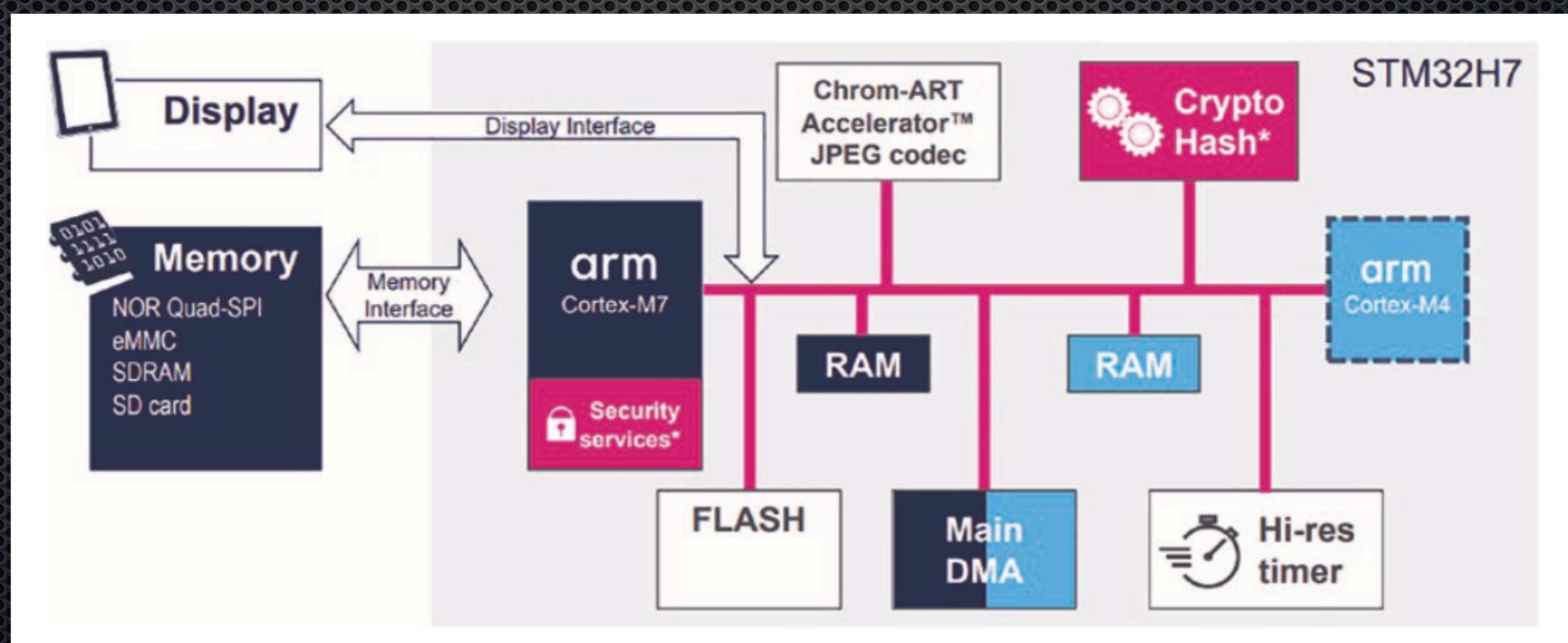
# 实时控制用例

- 多核微控制器系统中经常使用的另一个用例是让每个核心管理实时控制能力。
  - 例如，一个呼吸机系统使用多核微控制器STM32H7，M7内核用于驱动LCD和管理触摸输入。M4 用于运行控制泵和阀门所需的各种算法。



# 安全解决方案用例

- 多核的另一个流行用例是用应用程序管理安全解决方案。
  - 例如，开发人员可以使用内置在多核中的硬件隔离，将其中一个作为安全处理器，处理安全操作和信任根。另一个核心为正常应用空间。数据可以通过共享内存存在内核之间共享，但内核之间只能通过处理器间通信(IPC)请求进行交互。



# 参考文献

- Jacob Beningo, Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques. Apress, 2022. Chapter2 and5 (<https://link.springer.com/book/10.1007/978-1-4842-8279-3>).

