# Design and development of embedded systems for the Internet of Things (IoT)

Fabio Angeletti
Fabrizio Gattuso

SAPIENZA
UNIVERSITÀ DI ROMA

W • SENSE
INTEGRATED CABLELESS SOLUTIONS

# FreeRTOS in details

In this lesson we will review some topics we studied during last lesson and we will explore the following FreeRTOS routines and topics:

- Datatypes and Coding Style
- Dynamic memory management (Heap System)
- Task
- Queue
- Interrupt
- Task notifications

**At the end of the class we will talk about the final project in details and the deadlines.**

# Datatypes and coding styles

- It's better to **specify** if a variable is **signed** or **unsigned**.

- **Variables** are prefixed with their type: ***'c'*** for **char**, ***'s'*** for **int16t** (short), ***'l'*** **int32t** (**long**), and ***'x'*** for **BaseType_t** and **any other non-standard types** (structures, task handles, queue handles, etc.).

- If a **variable** is **unsigned**, it is also prefixed with a ***'u'***. If a variable is a **pointer**, it is also prefixed with a ***'p'*.**

- **Functions** are **prefixed** with both the type they **return**, and the **file** they are **defined** within

***xQueueReceive()*** *returns a variable* ***BaseType_t*** *and is defined in* ***queue.c***

# Datatypes and coding styles (2)

- **BaseType_t**

  **The most efficient, natural, type for the architecture**. For example, on a 32-bit architecture BaseType_t will be defined to be a 32-bit type. On a 16-bit architecture BaseType_t will be defined to be a 16-bit type.

- **TickleType_t**

  If configUSE_16_BIT_TICKS is set to 1, then TickType_t is defined to be an unsigned 16-bit type. If configUSE_16_BIT_TICKS is set to 0, then TickType_t is defined to be an unsigned 32-bit type.

| Type | Value |
| --- | --- |
| pdTRUE | 1 |
| pdFALSE | 0 |
| pdPASS | 1 |
| pdFAIL | 0 |

# Task

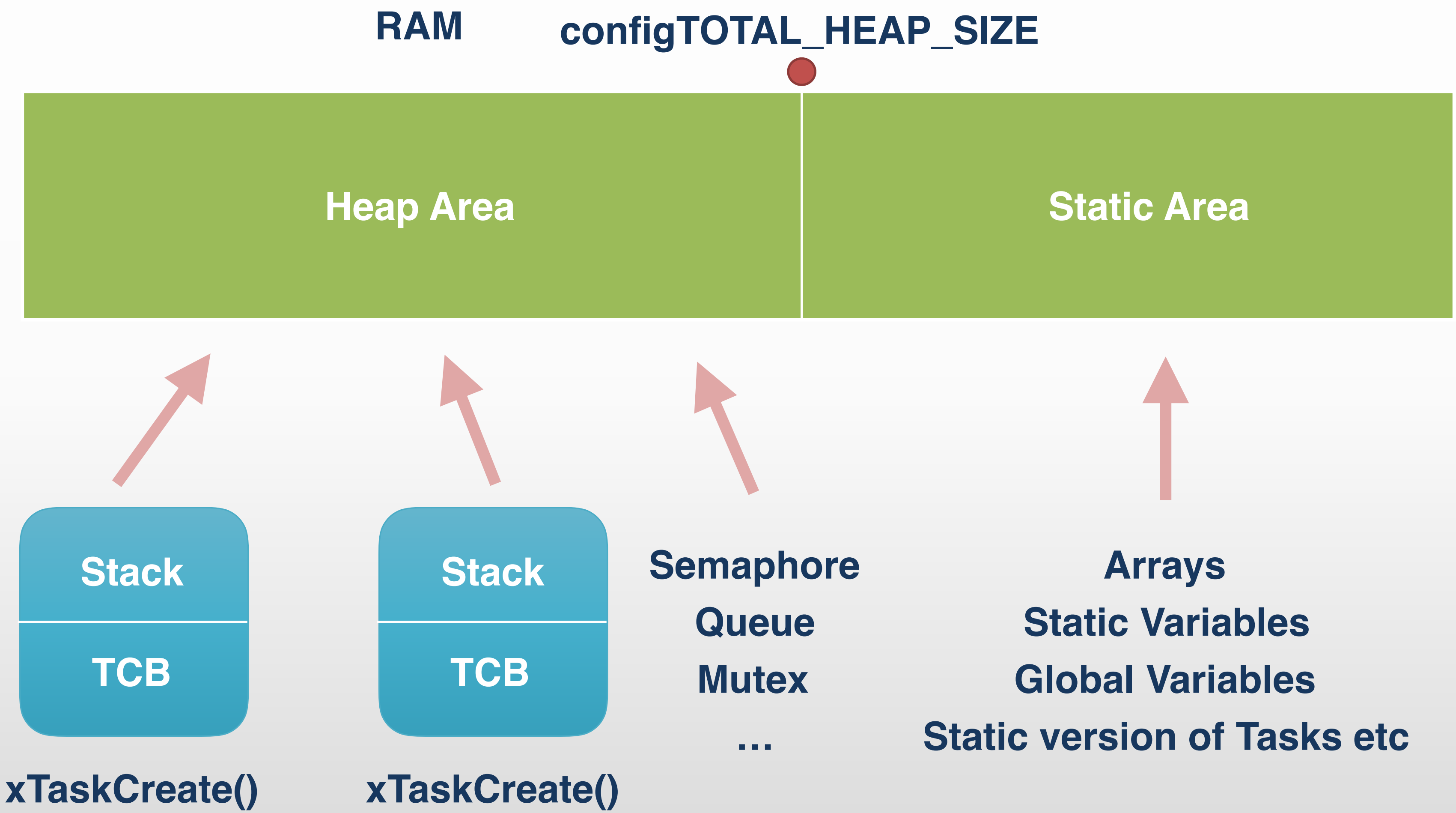
**Tasks are implemented as C functions. Return void and take a void pointer parameter, as shown here.**

```
void ATaskFunction( void const *pvParameters )  {

    /* Variables can be declared just as per a normal function. Each instance of
    a task created will have its own copy of the variable. This would not be true
    if the variable was declared static. */
    int32_t lVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; ) or while(1) {
        /* The code to implement the task functionality will go here. */
    }
}
```

# Memory management



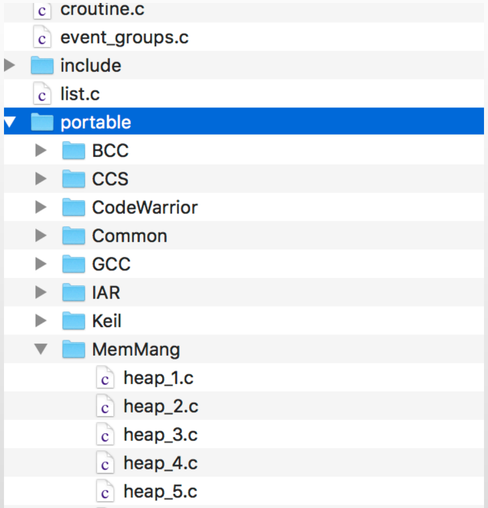
# Heap System

How we explained on the last lesson you can use static allocation or dynamic allocation.

FreeRTOS support 5 dynamic allocation systems:

**FreeRTOS/Source/portable/MemMang**

- Heap 1
- Heap 2
- Heap 3
- Heap 4
- Heap 5

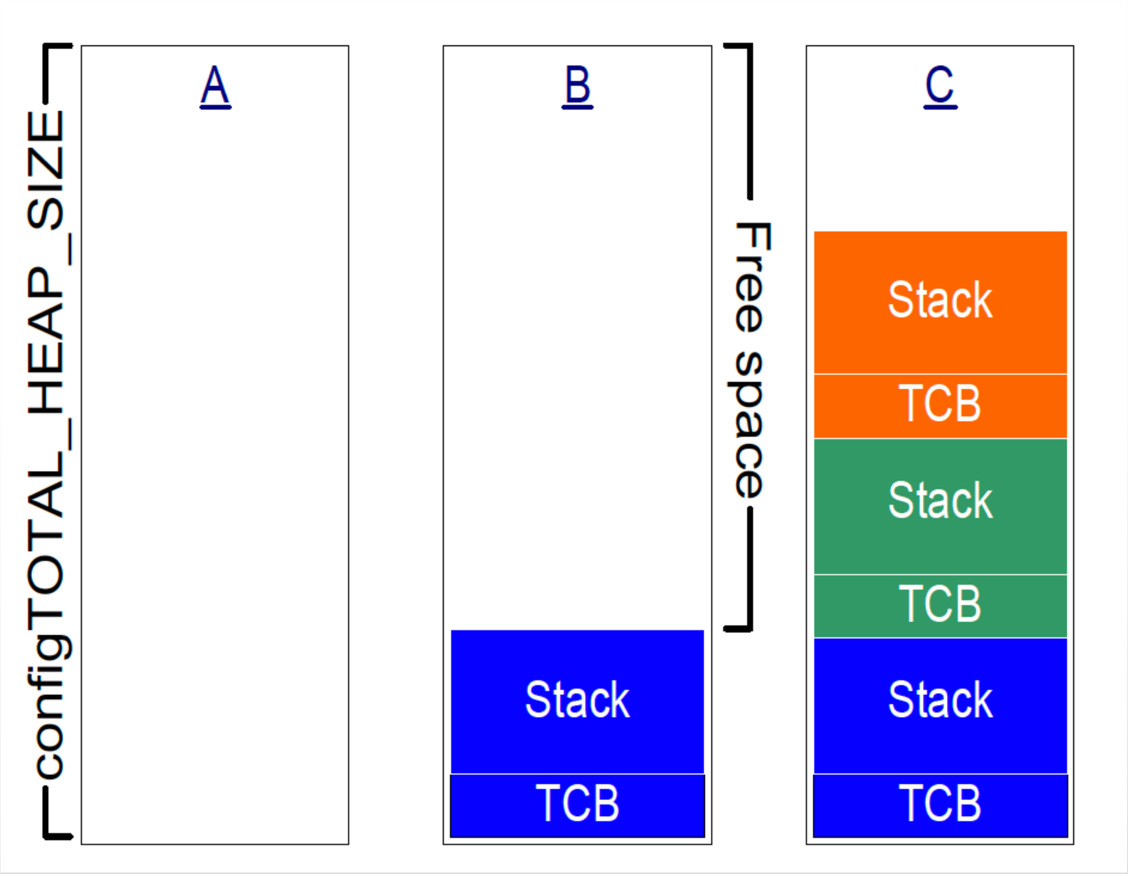


# Heap 1

It is the simplest scheme among all. **It does not permit memory to be freed once it has been allocated.**

The algorithm simply subdivides a single array into smaller blocks as requests for RAM are made. The total size of the array is set by the definition **configTOTAL_HEAP_SIZE** - which is defined in **FreeRTOSConfig.h**.

# Heap 1 (2)



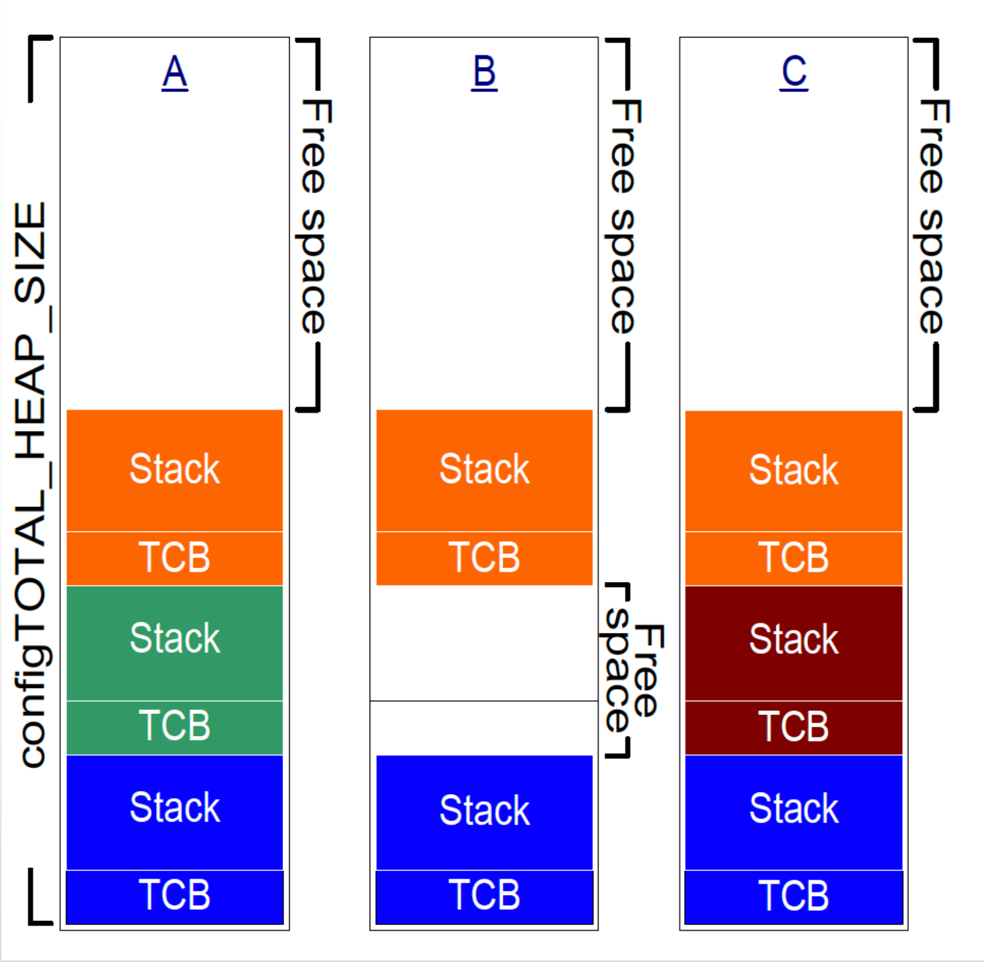
configTOTAL_HEAP_SIZE
A
B
Free space
C
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB

# Heap 2

This scheme uses a **best fit algorithm** and, unlike scheme 1, allows previously allocated blocks to be freed, however it does not combine adjacent free blocks into a single large block.

Again the total amount of available RAM is set by the definition **configTOTAL_HEAP_SIZE** - which is defined in **FreeRTOSConfig.h**.

# Heap 2 (2)



configTOTAL_HEAP_SIZE
A
Free space
Stack
TCB
Stack
TCB
Stack
TCB
B
Free space
Stack
TCB
Free space
Stack
TCB
C
Free space
Stack
TCB
Stack
TCB
Stack
TCB

# Heap 3

This scheme is just a wrapper for the standard **malloc()** and **free()** functions, making them thread-safe but still not deterministic.

This system require to increase the memory allocated to the kernel (OS).

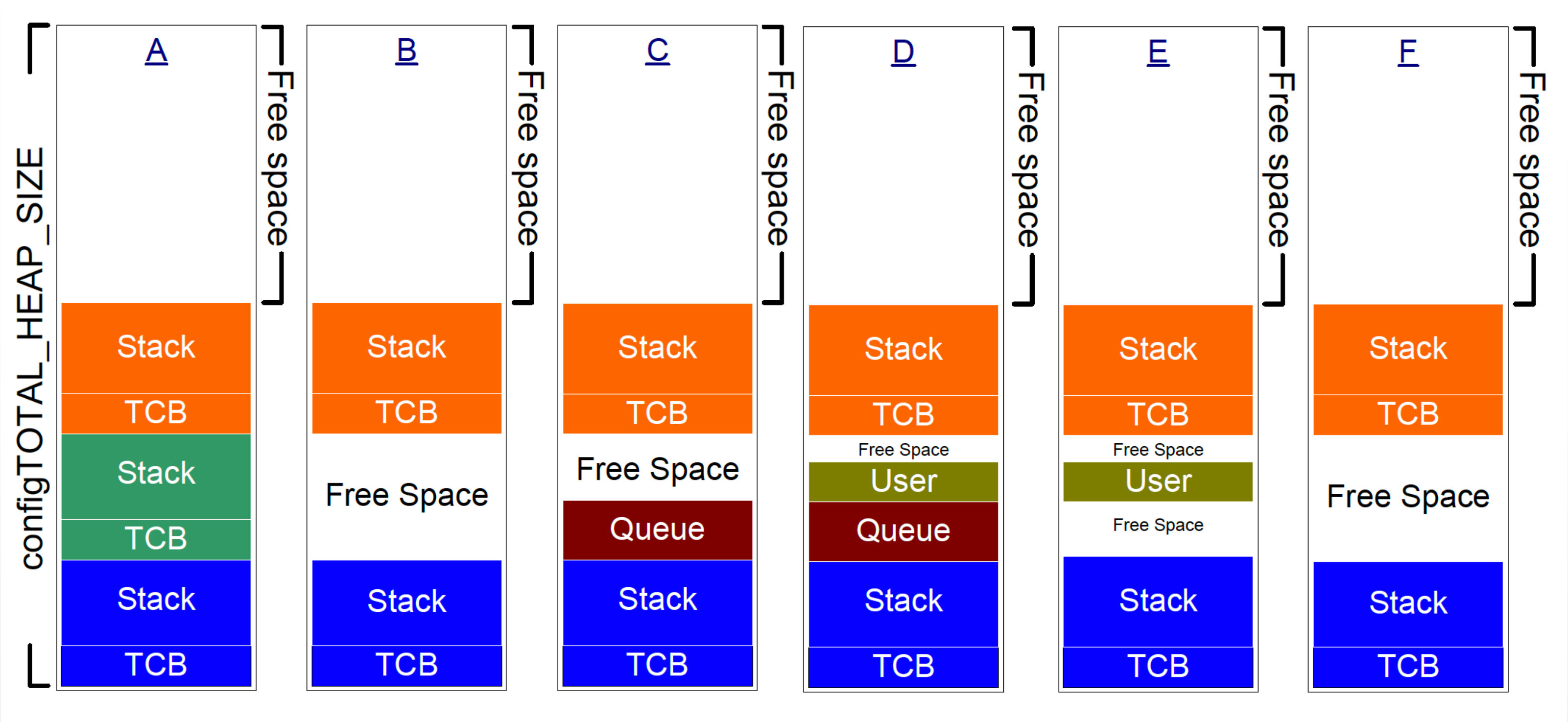
**Not suggested for real-time applications.**

# Heap 4

This scheme uses a **first fit algorithm** and, unlike scheme 2, it does **combine adjacent free memory blocks** into a single large block.

The **xPortGetFreeHeapSize()** API function returns the total amount of heap space that remains unallocated (allowing the configTOTAL_HEAP_SIZE setting to be optimized), but **does not provide information on how the unallocated memory is fragmented into smaller blocks**.

# Heap 4 (2)



configTOTAL_HEAP_SIZE
A
Free space
Stack
TCB
Stack
TCB
Stack
TCB
B
Free space
Stack
TCB
Free Space
Stack
TCB
C
Free space
Stack
TCB
Free Space
Queue
Stack
TCB
D
Free space
Stack
TCB
Free Space
User
Queue
Stack
TCB
E
Free space
Stack
TCB
Free Space
User
Free Space
Stack
TCB
F
Free space
Stack
TCB
Free Space
Stack
TCB

# Dynamic Memory Allocation API

You can use the standard C library mechanism to allocate the memory but it is:

**not deterministic, rarely thread-safe, not always available on small embedded systems**

malloc() -> **pvPortMalloc()**

free() -> **vPortFree()**

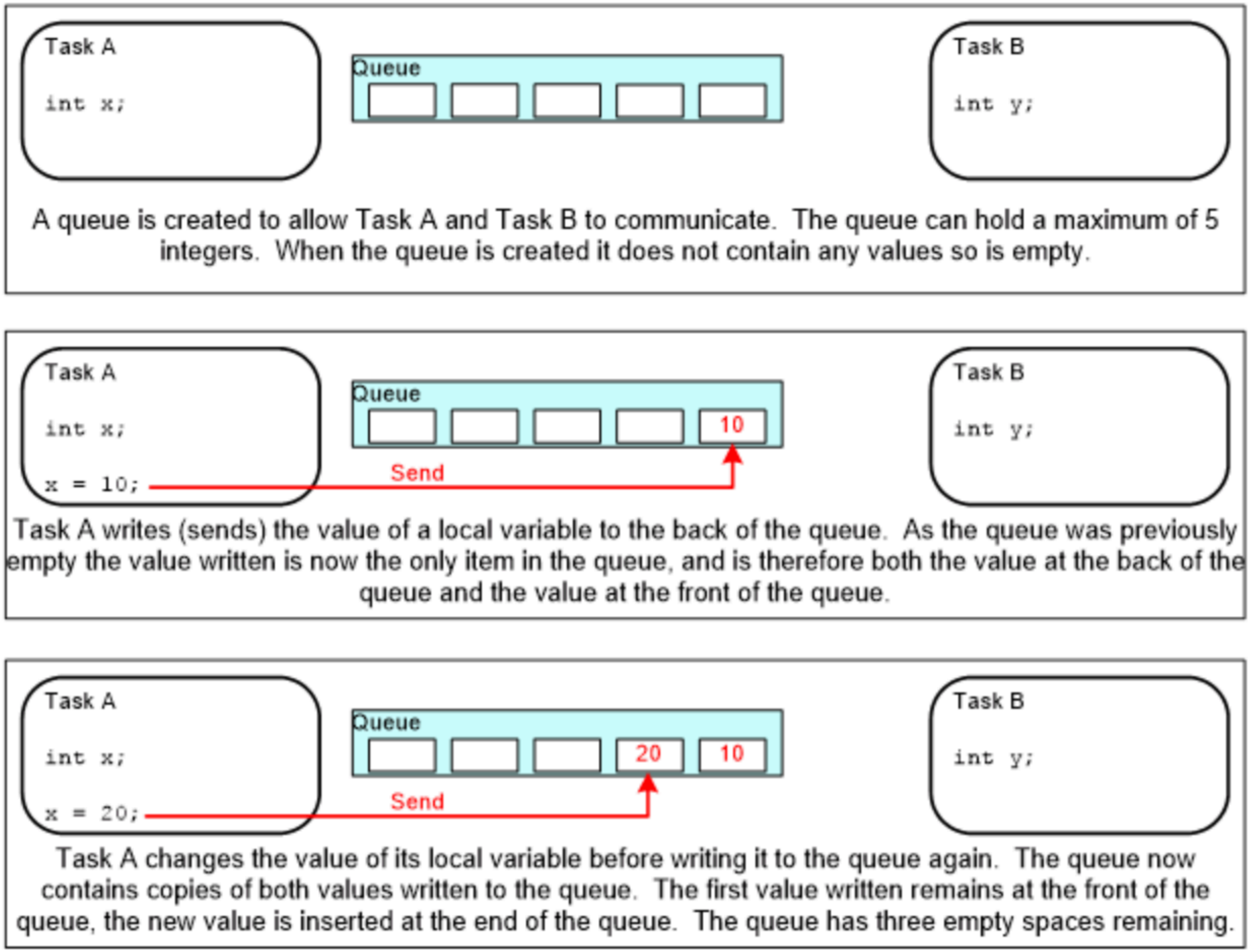
# Queue

Queues are use to communicate:

1. Task to task
2. Task to interrupt
3. Interrupt to task

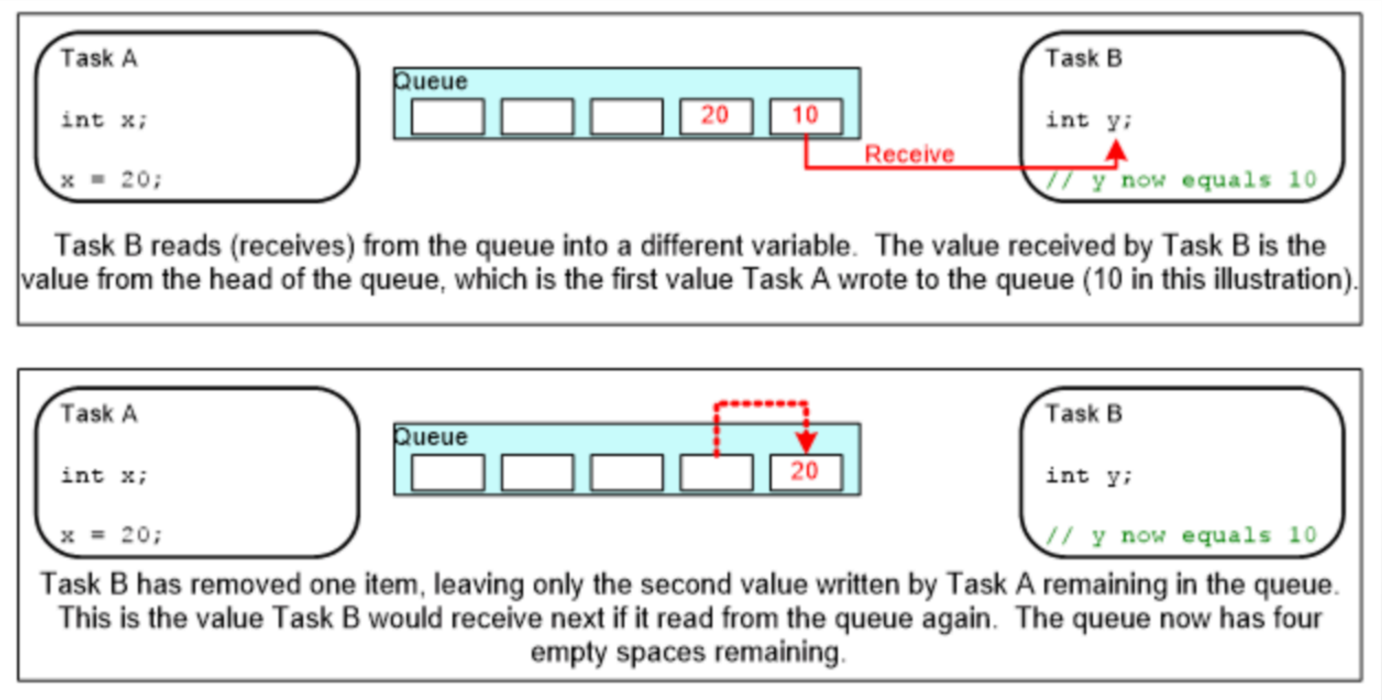
A queue can hold a **finite number of fixed-size data items**. The maximum **number of items** a queue can hold is called its **length**. Both the length and the size of each **data item are set when the queue is created**.

# Queue (2)



Task A
int x;
Queue
Task B
int y;
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.
Task A
int x;
x = 10;
Queue
10
Send
Task B
int y;
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.
Task A
int x;
x = 20;
Queue
20
10
Send
Task B
int y;
Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

# Queue (3)




The Queues work by **copying** an element into its memory (no references)

# Queue (4)

QueueHandle_t **xQueueCreate**( UBaseType_t **uxQueueLength**, UBaseType_t **uxItemSize** );

BaseType_t **xQueueSendToFront**( QueueHandle_t **xQueue**, const void * **pvItemToQueue**, TickType_t xTicksToWait );

BaseType_t **xQueueSendToBack**( QueueHandle_t **xQueue**, const void * **pvItemToQueue**, TickType_t xTicksToWait );

BaseType_t **xQueueReceive**( QueueHandle_t **xQueue**, void * const **pvBuffer**, TickType_t xTicksToWait );

# Queue with pointers

You can also use queue with pointers when the **data to store is large**.

- When using a pointer to share memory between tasks, you must make sure that **both tasks do not modify the memory contents simultaneously,** as this could cause the **memory** contents to be **invalid** or **inconsistent.**
- If the memory was allocated dynamically or obtained from a pool of preallocated buffers, **one task should be responsible for freeing the memory**.
- You should **never use a pointer to access data that has been allocated on a task stack**. The data will not be valid after the stack frame has changed.

# Software timers

Software timers are used to **schedule** in the **future** or **periodically** with a fixed frequency.

The timers are not related to the hardware but they **do not use any processing time**.

You have to **include** the **<timer.c>** on your project and set **configUSE_TIMERS = 1** in **FreeRTOSConfig.h**

**Don't use block code inside a timer code**

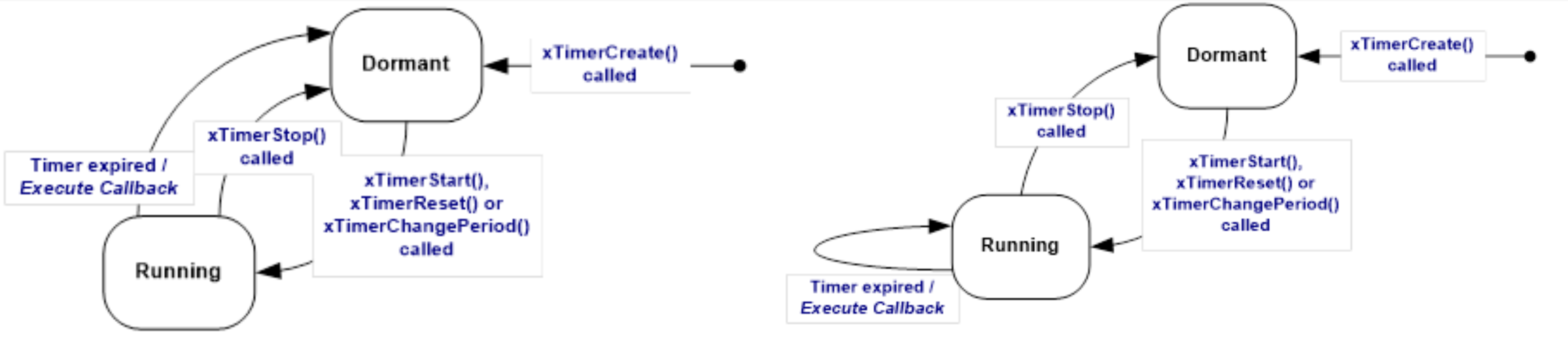
# Software timers (2)

- **One shot**

  a one-shot timer will execute its callback function **only once**. A one-shot timer can be **restarted manually**.

- **Auto-reloaded**

  an auto-reload timer will **restart itself each time it expires**, resulting in **periodic execution** of its callback function.




**One Shot** **Auto-reloaded**

# Software timers (3)

TimerHandle_t **xTimerCreate**( const char * const **pcTimerName**,
TickleType_t **xTimerPeriodInTicks**,
UBaseType_t **uxAutoreload**, void * **pvTimerID**,
TimerCallBackFunction_t **pxCallbackFunction**);

TimerHandle_t **xTimerStart**( TimerHandle_t **xTimer**, TickType_t **xTicksToWait** );

TimerHandle_t **xTimerReset**( TimerHandle_t **xTimer**, TickType_t **xTicksToWait** );

TimerHandle_t **xTimerStop**( TimerHandle_t **xTimer**, TickType_t **xTicksToWait** );

TimerHandle_t **xTimerChangePeriod**( TimerHandle_t **xTimer**,
TickType_t **xNewTimerPeriodInTicks**, TickType_t **xTicksToWait** );

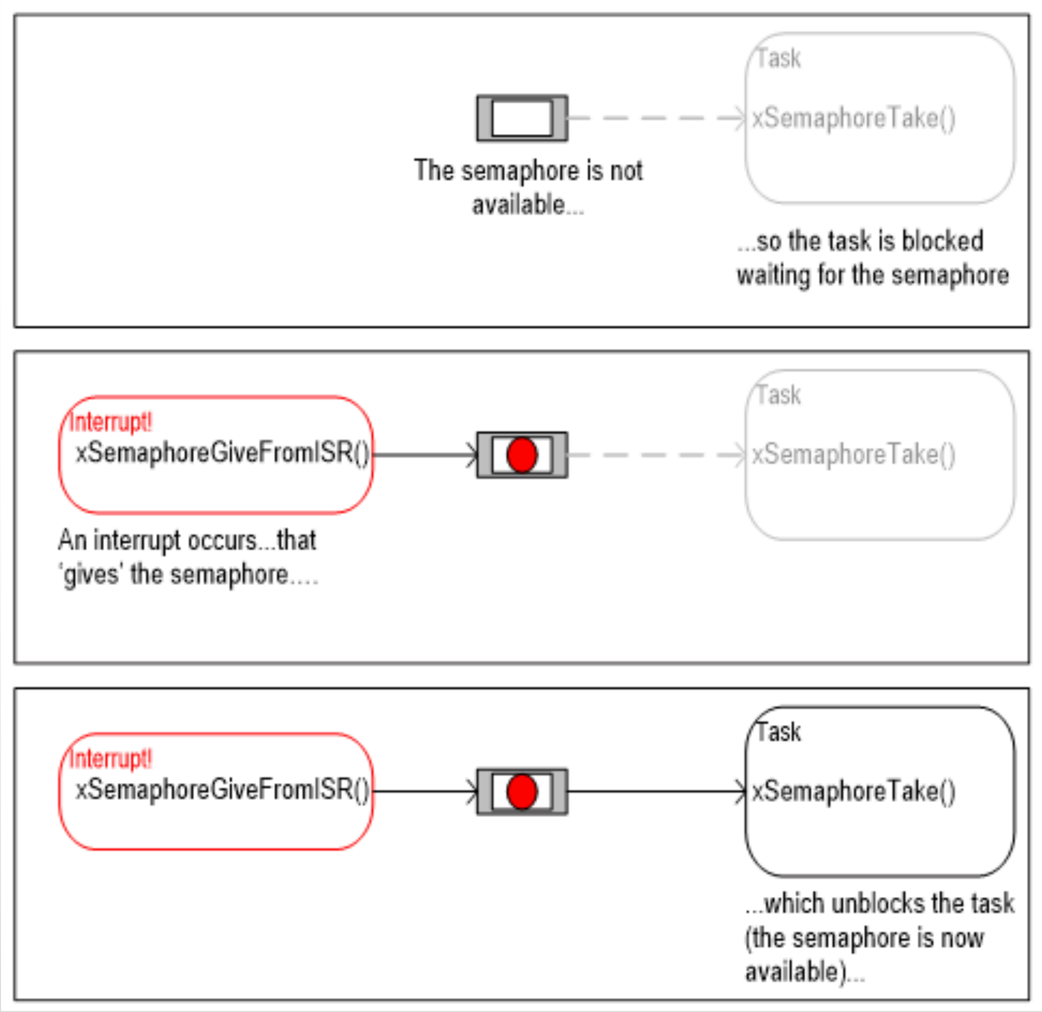
# Interrupt

An **interrupt service routine (ISR)** is a **hardware** feature because the hardware controls which interrupt service routine will run and when. FreeRTOS provides two versions of some API functions **(FROM_ISR).**

- From ISR you have to do non trivial operations.
- The interrupt processing is not deterministic.
- You have to delegate the main job to a correlated task.

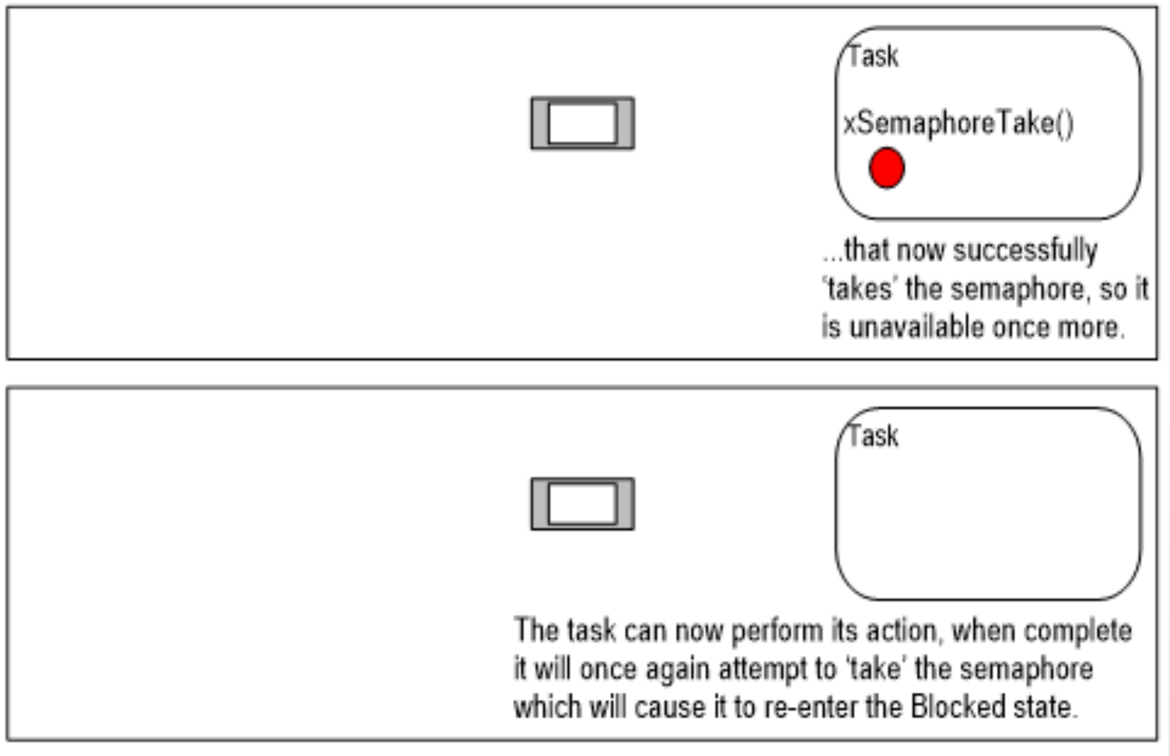
**A semaphore or a mutex is used as a synchronization method**

# Interrupt (2)



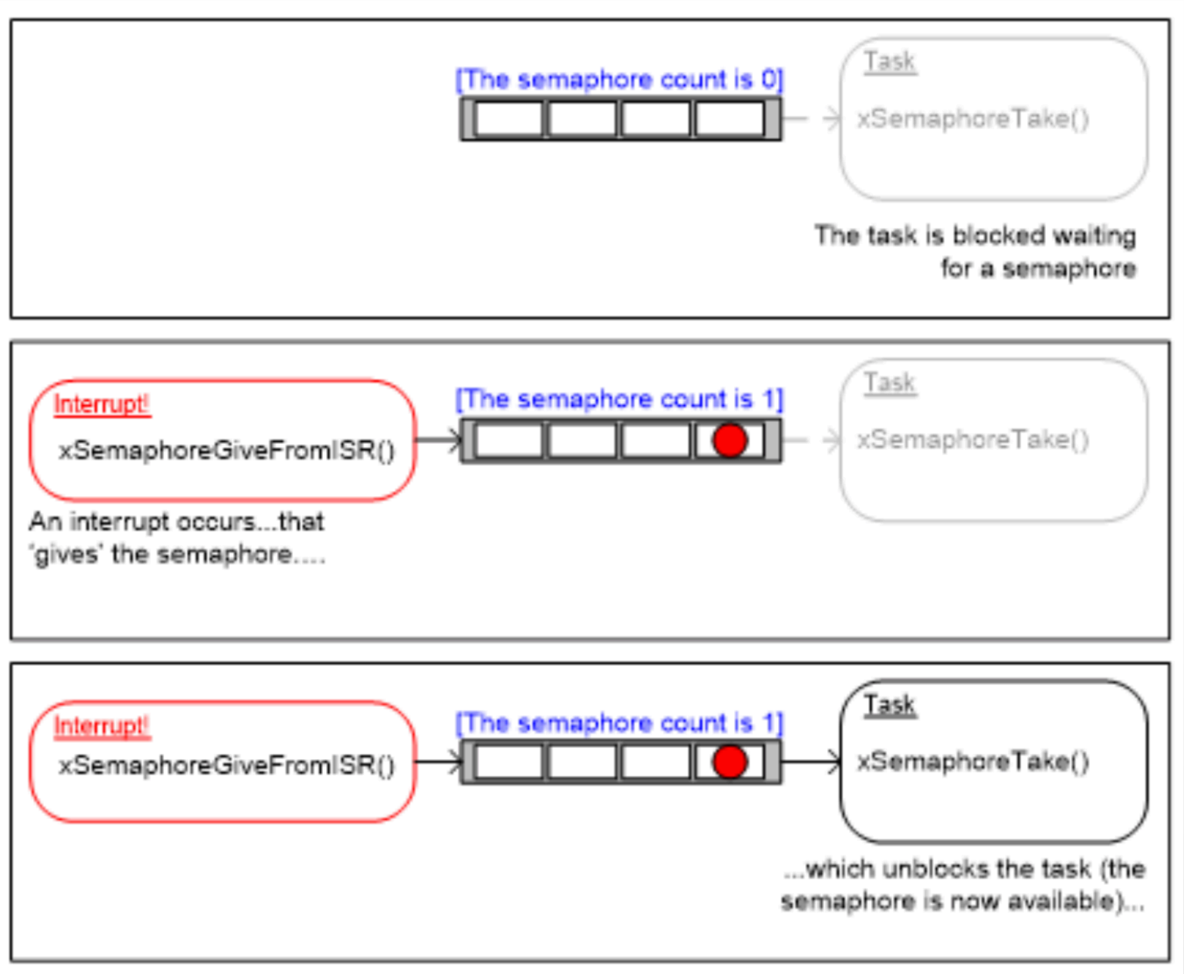
Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...

# Interrupt (3)

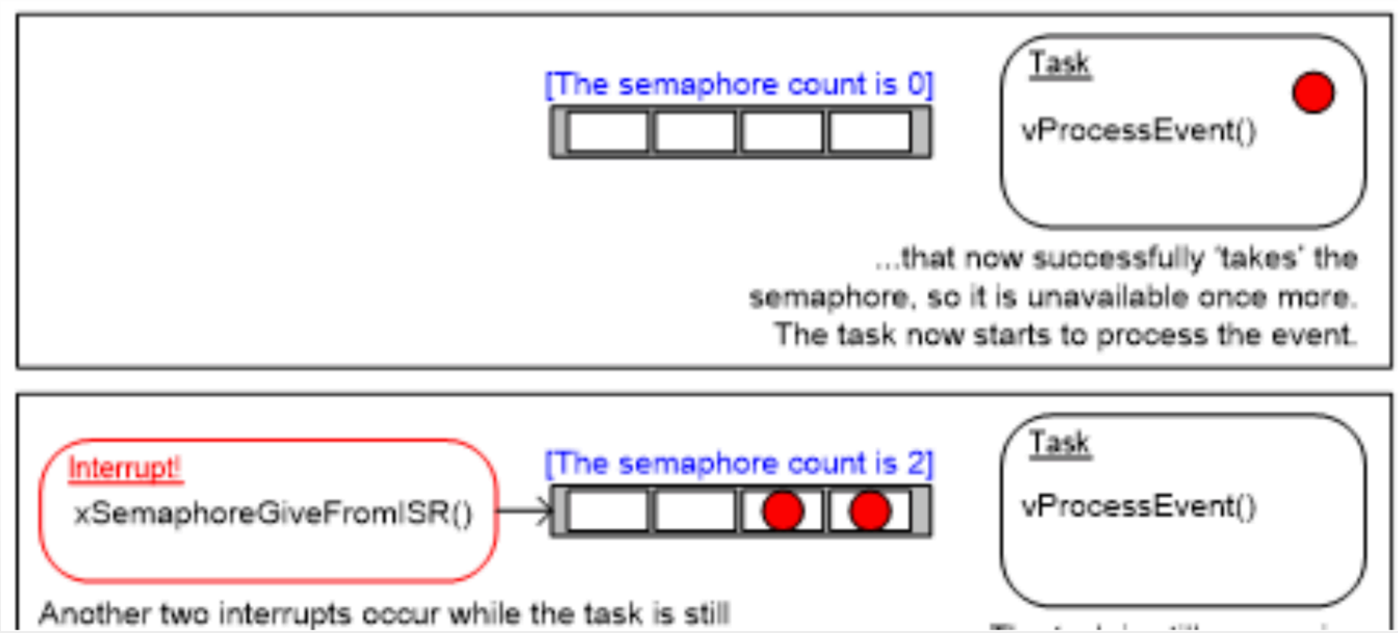



What's the problem here? Can we do better?

# Interrupt (4)


[The semaphore count is 0]
Task
xSemaphoreTake()
The task is blocked waiting for a semaphore
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...

# Interrupt (5)




We studied at least one other system we can use here…

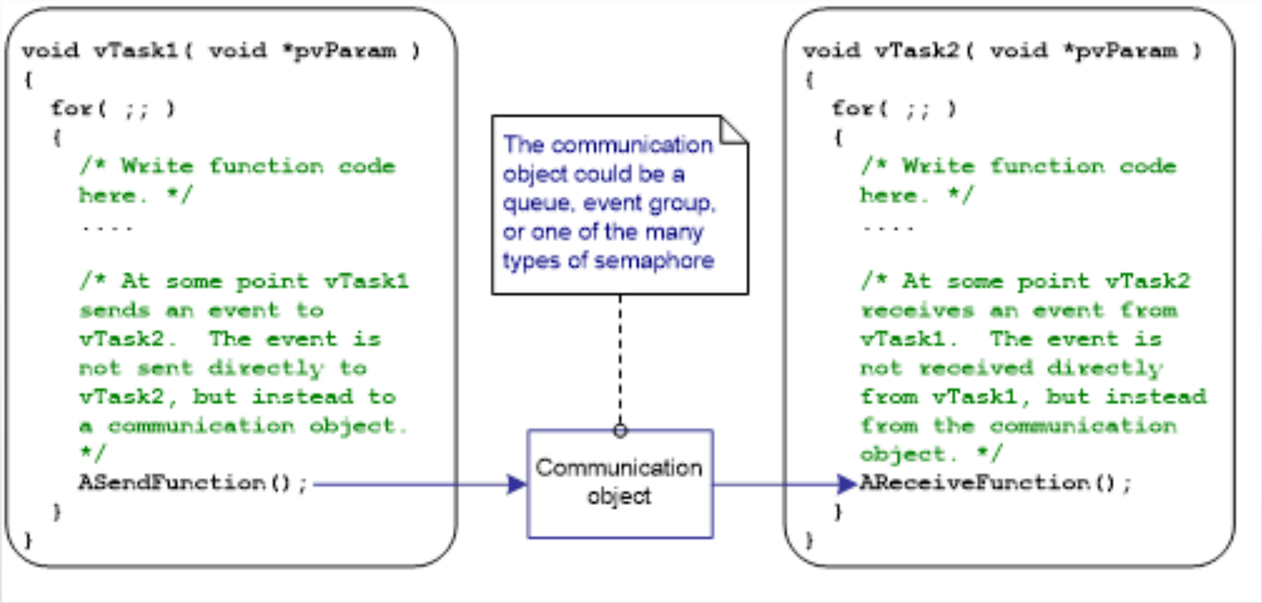
# Interrupt (6)

We can synchronize an ISR with a task with different methods:

- Binary Semaphore
- Counting Semaphore
- Mutex
- Queue

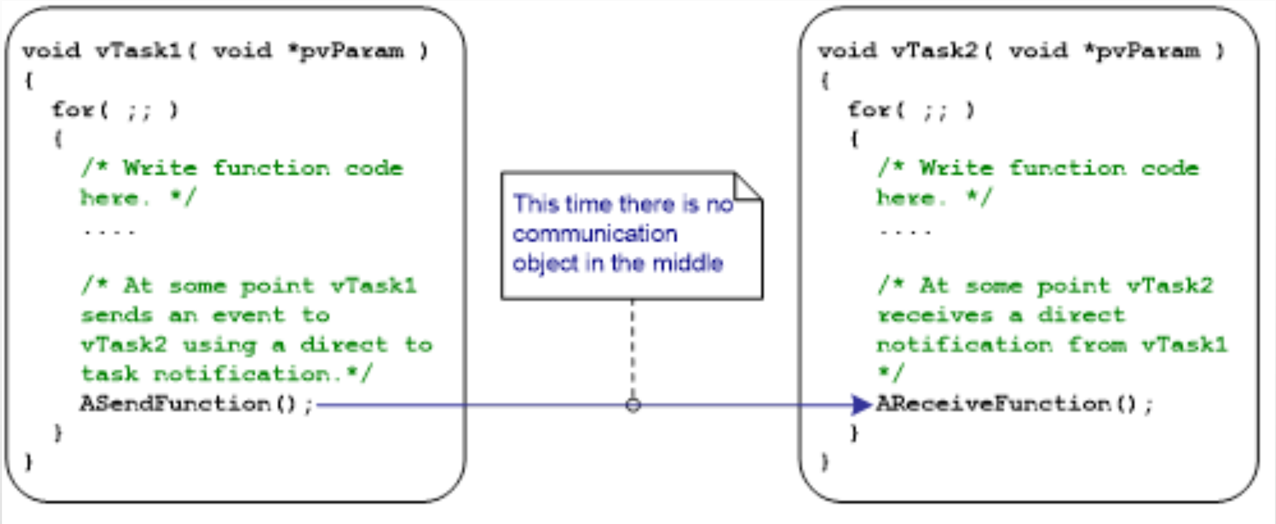
# Task notifications

Two tasks can talk to each other between intermediary objects.



# Task notifications (2)

Task notifications allow tasks to interact with other tasks and to synchronize with ISRs without the need for a separate communication object



# Task notifications (3)

You have to set **configUSE_TASK_NOTIFICATIONS = 1** in **FreeRTOSConfig.h**

When you active the task notification each task has a notification state, which can be pending or not pending. When a task receives a notification, its notification state is set to pending.

Using a task notification to send an event or data to a task is **significantly faster** than using a queue or semaphore.

- Task notifications can be used to send events and data from an **ISR to a task** or **task to task.**
- Task notifications are sent directly to the receiving task, so can be **processed only by the task to which the notification is sent.**
- A task's notification value can hold **only one value at a time**.

# Task notifications (4)

BaseType_t **xTaskNotifyGive**( TaskHandle_t **xTaskToNotify**);

BaseType_t **xTaskNotifyGiveFromISR**( TaskHandle_t **xTaskToNotify,**
BaseType_t *pxHigherPriorityTaskWoken);

uint32_t **ulTaskNotifyTake**( BaseType_t **xClearCountOnExit**,
TickType_t xTicksToWait );

There are also **xTaskNotify**, **xTaskNotifyFromISR**, **xTaskNotifyWait** advanced version of the previous functions.

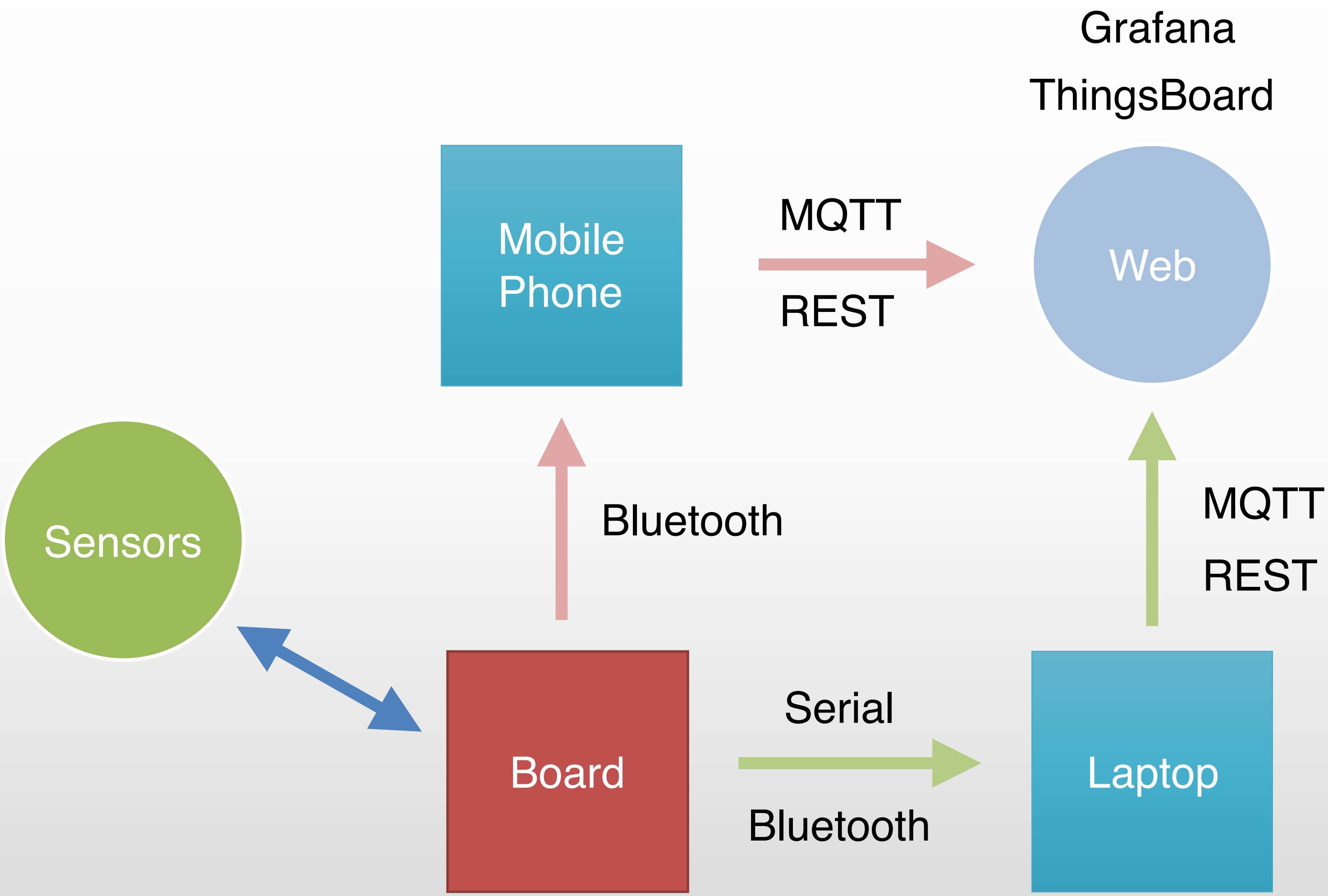
# The final project

You have to present a final project to pass the class.

The requirements:

- Work on **FreeRTOS** on your board
- Use at least **one external sensor**
- Interact with at least **one external system** (**serial** bus, **bluetooth**)
- Visualize data or statistics with **Thingsboard** or **Grafana**

# The final project (2)



# The final project (3)

You can also work on networks problems like **indoor localization** or **time synchronization** but that can be trivial.

It's **highly suggested** to work on group but it's accepted to work alone.

**DEADLINES**

**By the 21st April you have to submit your group details and the hardware you need.**

**By the 1st May you have to choose your final project.**

**By the 8th June you have to upload your project on GitHub. The presentation will be on the 12th June.**

# Next lessons

We have six lessons more. These are the next topics:

- IoT Network Technologies and protocols
- How to read and understand a data-sheet
- Low power techniques
- IoT Security
- IoT on Cloud and web data visualization
- WSense IoT Real Examples