EOS - LECTURE 1

Embedded Operating System

The basics of freeRTOS

Samuele Germiniani
samuele.germiniani@univr.it
Graziano Pravadelli
graziano.pravadelli@univr.it

Part1	break	Part2
60m +	10m +	60m

OUTLINE

1 - FreeRTOS introduction

1 - FreeRTOS introduction

What is FreeRTOS?

FreeRTOS is a real-time kernel on top of which embedded applications can be built to meet their hard real-time requirements.

- An application is organized as a collection of independent threads (task).
- The kernel decides which thread should be executing by examining the priorities
- The designer assigns the priorities to implement hard/soft real-time constraints

1 - FreeRTOS introduction

Why FreeRTOS?

For its value proposition!

- Supported
- Strictly quality controlled and professionally developed
- It is truly free to use in commercial applications without any requirement to expose your proprietary source code.

Why a real-time kernel?

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Idle time
- Power Management

FreeRTOS features

<u>FreeRTOS has the following standard features</u>

- **Pre-emptive or co-operative operation**
- **Very flexible task priority assignment**
- **Flexible, fast and light weight task notification mechanism**
- **Queues**
- **Binary semaphores**
- **Counting semaphores**
- **Mutexes**
- **Recursive Mutexes**
- **Software timers**

1 - FreeRTOS introduction

FreeRTOS licensing

The FreeRTOS open source license is designed to ensure:

1. *FreeRTOS can be used in commercial applications.*

2. *FreeRTOS itself remains freely available to everybody.*

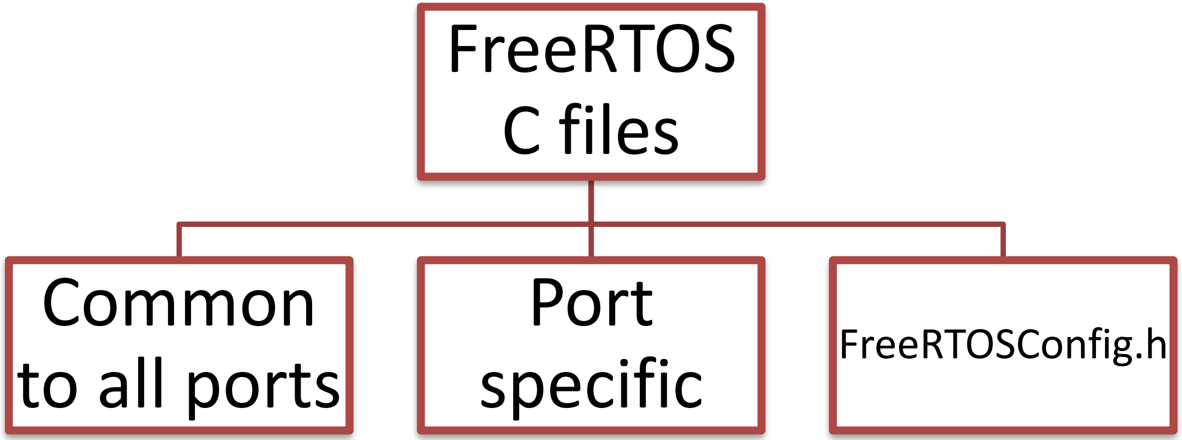
3. *FreeRTOS users retain ownership of their intellectual property.*

2 - The FreeRTOS Distribution

2.1 Organization of files

FreeRTOS supports over 20 compilers and 30 processors

Definition: *<u>FreeRTOS port</u>*
A supported combination of compiler and processor



2.1 Organization of files

Go to https://www.freertos.org/ and download the **FreeRTOS 202104.00** sources

1. wget https://github.com/FreeRTOS/FreeRTOS/releases/download/202104.00/FreeRTOSv202104.00.zip
2. unzip FreeRTOSv202104.00.zip && rm FreeRTOSv202104.00.zip
3. cd FreeRTOSv202104.00/ && ls

2.1 Organization of files

The Top Directories in the FreeRTOS Distribution

```
FreeRTOS
 │    │
 │    ├─Source    Directory containing the FreeRTOS source files
 │    │
 │    └─Demo      Directory containing pre-configured and port specific FreeRTOS demo projects
 │
FreeRTOS-Plus
      │
      ├─Source    Directory containing source code for some FreeRTOS+ ecosystem components
      │
      └─Demo      Directory containing demo projects for FreeRTOS+ ecosystem components
```

2.1 Organization of files

FreeRTOS Source Files Common to All Ports

```
FreeRTOS
    │
    └─Source
        │
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

2.1 Organization of files

FreeRTOS Source Files Specific to a Port

```
FreeRTOS
  │
  └─Source
     │
     └─portable Directory containing all port specific source files
        │
        ├─MemMang Directory containing the 5 alternative heap allocation source files
        │
        ├─[compiler 1] Directory containing port files specific to compiler 1
        │  │
        │  ├─[architecture 1] Contains files for the compiler 1 architecture 1 port
        │  ├─[architecture 2] Contains files for the compiler 1 architecture 2 port
        │  └─[architecture 3] Contains files for the compiler 1 architecture 3 port
        │
        └─[compiler 2] Directory containing port files specific to compiler 2
           │
           ├─[architecture 1] Contains files for the compiler 2 architecture 1 port
           ├─[architecture 2] Contains files for the compiler 2 architecture 2 port
           └─[etc.]
```

2.1 Organization of files

Include paths

- FreeRTOS/Source/include
- FreeRTOS/Source/portable/[compiler]/[architecture]
- A path to the FreeRTOSConfig.h header file

It contains the application's configuration

2.2 Creating a FreeRTOS Project

How to start a new project

FreeRTOS provides a demo for each compiler/architecture combination

1. Find your compiler + target architecture (we are using the POSIX simulator)

```
cd FreeRTOS/Demo/Posix_GCC/
```

2. Check if it compiles correctly

```
make
```

3. Start the demo

```
./build/posix_demo
```

4. ~~Remove all the demo files~~ Download a ready empty project

```
cd .. && rm -rf Posix_GCC
git clone https://gitlab.com/SamueleGerminiani/posix_eos1 && cd posix_eos1
```

2.3 Data Types and Coding Style

Variable Names

- Variables are prefixed with their type: **c** for char, **s** for int16_t (short),… **x** for non-standard types

Function Names

- Functions are prefixed with both the type they return, and the file they are defined within:
- **vTaskPrioritySet**() returns a **void** and is defined within **task.c**

(File scope (private) functions are prefixed with prv.)

2.3 Data Types and Coding Style

Macro Names

Upper case, and prefixed with lower case letters that indicate where the macro is defined

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

3. Heap Memory Management

3. Heap Memory Management

Dynamic memory allocation

- FreeRTOS comes with five example implementations of both **pvPortMalloc()** and **vPortFree()**

- The five examples are defined in the heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c source files respectively, all of which are located in the FreeRTOS/Source/portable/MemMang directory.

- *Compile* your project with the *preferred implementation*

3. Heap Memory Management

Dynamic memory allocation

What memory implementation are we using?

4 - Task management

4.1 The basics of a task

What is a task function?

Tasks are implemented as C functions.
The only thing special about them is their prototype.

```
void ATaskFunction( void *pvParameters );
```

N.B FreeRTOS tasks must not be allowed to return from their implementing function in any way

4.1 The basics of a task

Create a task function

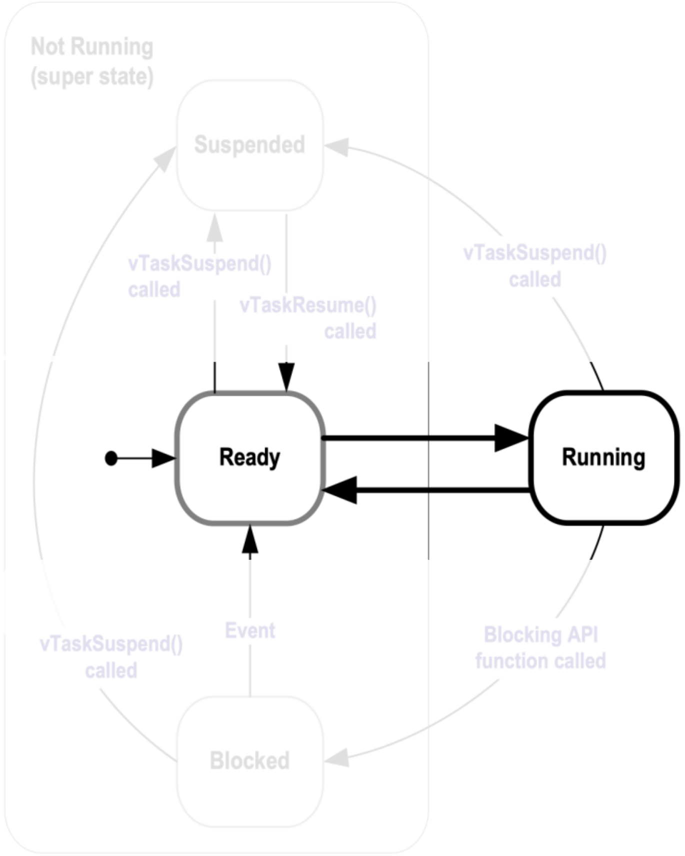
```
void vTask1(void *pvParameters) {
  const char *pcTaskName = "Task 1 is running\r\n";
  volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */
  /* As per most tasks, this task is implemented in an infinite loop. */
  for (;;) {
    /* Print out the name of this task. */
    console_print("%s\n",pcTaskName);
    /* Delay for a period. */
    for (ul = 0; ul < 100000000; ul++) {
      /* This loop is just a very crude delay implementation.
      There is nothing to do in here. Later examples will replace this crude
      loop with a proper delay/sleep function. */
    }
  }
}
```

--> Put this function in main.c

4.1 The basics of a task

Task States

- **Ready** : The task is waiting to enter the running state
- **Running**: The task is using the CPU (we assume a uniprocessor system)
- The scheduler selects always the **Task** with the highest priority in the ready state



4.2 Creating Tasks

The xTaskCreate() API Function

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

A pointer to the function that implements the task

A descriptive name for the task. This is not used by FreeRTOS in any way.

Tells the kernel how large to make the stack for this task.

Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task.

Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1),

Used to reference the task from other entities

4.2 Creating Tasks

The xTaskCreate() API Function

Use the API function to create tasks

1. In file main.c, define task function vTask2 (identical to vTask1 except that it prints a different message)
2. In file main.c, function "main" (before starting the scheduler), use the API function to create two runnable tasks (one for function vTask1 and one for vTask2). Give the same priority to both tasks

Example:
xTaskCreate(taskFunction, "Inconsequential string", 1000, NULL, 1, NULL);

N.B. put NULL when you don't use a parameter

4.2 Creating Tasks

The xTaskCreate() API Function

Activate pre-emption

3. In file FreeRTOSConfig.h, set configUSE_PREEMPTION to activate pre-emption

4.2 Creating Tasks

The xTaskCreate() API Function

Using the task parameter

Use the task parameter to pass the string that should be printed

1. Remove vTask2
2. Make vTask1 generic (vTask1 --> vTask)
3. In function main, add the string as a parameter of xTaskCreate() (remember to cast to void*)

4. Modify vTask to retrieve the string from pvParameters (remember to cast to char*).

4.3 Time Measurement and the Tick Interrupt

Tick interrupt

- ❖ A timed interrupted whos execution frequency is determined by configTICK_RATE_HZ

- ❖ <u>In FreeRTOS, time is measured after the tick interrupt</u>

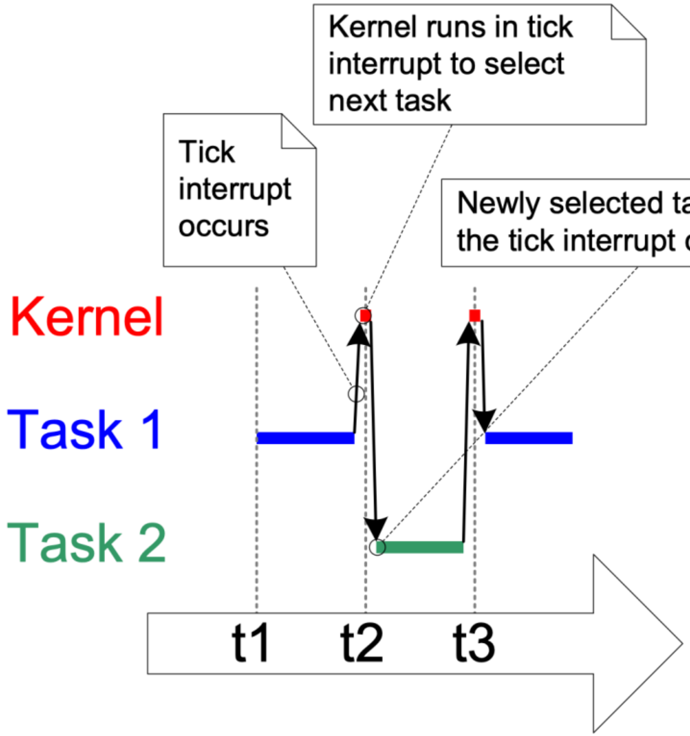
The time between two tick interrupts is called the "tick period".
One time slice equals one tick period.

Example:
if configTICK_RATE_HZ is set to 100 (Hz), then the time slice will be 10 milliseconds

4.3 Time Measurement and the Tick Interrupt

Tick interrupt

- The tick interrupt implements time slicing


Kernel runs in tick interrupt to select next task
Tick interrupt occurs
Newly selected task runs when the tick interrupt completes
Kernel
Task 1
Task 2
t1
t2
t3

4.3 Time Measurement and the Tick Interrupt

Tick interrupt

1. Find configTICK_RATE_HZ and increase it to 1000.

4.3 Time Measurement and the Tick Interrupt

Experimenting with priorities

1. Change the priority of "Task 1" to 10

Answer the following question
What happens to Task 2? Why?

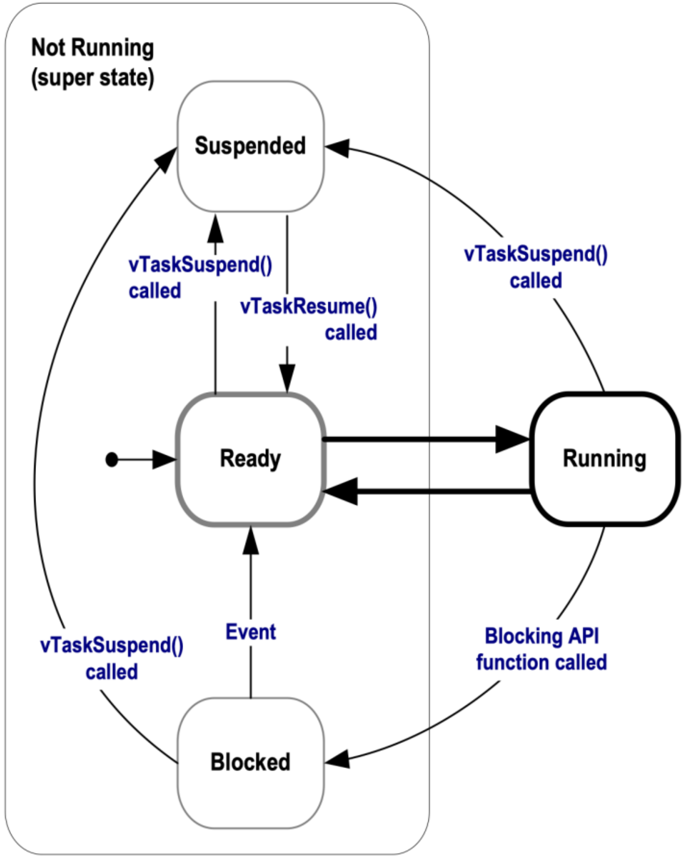
4.4 Blocked and Suspended

The Suspended State

- Task suspended using the vTaskSuspend() API function

The Blocked State

A task waiting for an event:
- Temporal events
- Synchronization events: queues, semaphores



4.4 Blocked and Suspended

Using the Blocked state to create a delay

1. Modify vTask to make it wait for 100ms using vTaskDelay() instead of the current wasteful "for" cycle

```
void vTaskDelay( TickType_t xTicksToDelay );
```

The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks

4.4 Blocked and Suspended

vTaskDelay() vs vTaskDelayUntil()

- Time in which Task leaves the blocked state is relative to the time at which vTaskDelay() was called.
- vTaskDelayUntil() specifies the exact tick count value at which the calling task should be woken.

4.4 Blocked and Suspended

How to use vTaskDelayUntil()

```
// Perform an action every 10 ticks.
void vTaskFunction(void *pvParameters) {
  TickType_t xLastWakeTime;
  const TickType_t xFrequency = 10;
  // Initialise the xLastWakeTime variable with the current time.
  xLastWakeTime = xTaskGetTickCount();
  for (;;) {
    // Wait for the next cycle.
    vTaskDelayUntil(&xLastWakeTime, xFrequency);
    // Perform action here.
  }
}
```

The function xTaskGetTickCount() can be used to get the current "time" in tick counts.

Gets automatically updated after every delay (that is why we have to give a pointer)

```
void vTaskDelayUntil( TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement );
```

4.4 Blocked and Suspended

Using the Blocked state to create a delay (vTaskDelayUntil)

1. Modify vTask to make it wait using vTaskDelayUntil() instead of vTaskDelay()

4.4 Blocked and Suspended

Suspending/resuming processes

1. Create a task "vBlocker" that keeps suspending or resuming **Task 2** every 500 ms

N.B. you will need to store the handler (use global variables) of Task 2, see xTaskCreate API.

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );

void vTaskResume( TaskHandle_t xTaskToResume );
```

4.5 The Idle Task and the Idle Task Hook

The idle Task

- There must always be at least one task that can enter the Running state
- The idle Task executes when no other tasks can
- It runs at the lowest priority (0)

The idle Task Hook

- The idle task can be configured through a Hook function
- Used to execute low priority, background, or continuous processing functionalities.

```
void vApplicationIdleHook( void );
```

n.b. An Idle task hook function must never attempt to block or suspend.

4.5 The Idle Task and the Idle Task Hook

Using the idle task

1. In file, **FreeRTOSHooks.c** , implement the idle task hook to increment a counter each time it is executed
2. Task1 must print the counter every second

4.6 Changing the Priority of a Task

The vTaskPrioritySet()/uxTaskPriorityGet() API Function

The vTaskPrioritySet() API function can be used to change the priority of any task after the scheduler has been started.

priority must be < configMAX_PRIORITIES, where configMAX_PRIORITIES is a compile time constant set in the FreeRTOSConfig.h header file.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

4.6 Changing the Priority of a Task

Changing priorities

1. Create an arbiter task that swaps the priority of Task1 and Task2 every 2 seconds.
Task1 and Task2 must print their name and their priority every 200 ms.
(make sure that Task1 and Task2 are created with different priorities)

n.b passing NULL to an API requiring a TaskHandle_t will cause the API to use the caller's task handle instead

4.7 Deleting a Task

The vTaskDelete() API Function

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

1. Create an "eraser" task that deletes Task1, Task2 and arbiter after 5 second. After that, eraser suspends himself.

5 - Semaphores

5 - Semaphores

Mutex

- A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks.

How to use a mutex

- Create a Mutex

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

- Take a mutex when accessing shared resources

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                TickType_t xTicksToWait );
```

Time to wait for the mutex to be freed.
Set it to "portMAX_DELAY", to wait indefinitely

- Free the mutex

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

5 - Semaphores

Modifying the console_print function

The "standard out" is a shared resource.
Use a mutex to make it mutually exclusive.

1. In file console.c, declare a mutex variable.
2. In file console.c, function console_init, initialise the mutex (create)
3. In file console.c, function console_print, take the mutex before accessing the standard out (take)
4. In file console.c, function console_print, free the mutex after accessing the standard out (give)

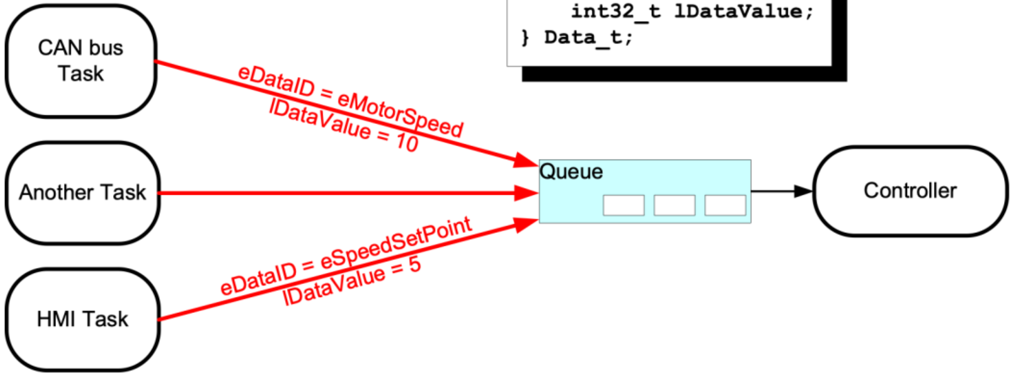
6 - Queue

6 - Queue

What is a queue?

queues are a type of container designed to operate in a FIFO context, where elements are inserted into one end of the container and extracted from the other

```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;
```



6 - Queue

How to use a generic queue in FreeRTOS

- <u>Create a Queue</u>

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

The returned value should be stored as the handle to the created queue

The maximum number of items that the queue being created can hold at any one time

The size in bytes of each data item that can be stored in the queue.

6 - Queue

How to use a generic queue in FreeRTOS

- Add element

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

The handle to the queue on which the item is to be posted

A pointer to the item that is to be placed on the queue

The maximum amount of time the task should block waiting for space to become available on the queue.
Set it to "portMAX_DELAY", to wait indefinitely

6 - Queue

How to use a generic queue in FreeRTOS

- Remove element

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

The handle to the queue on which the item is to be posted

Pointer to the buffer into which the received item will be copied

The maximum amount of time the task should block waiting for an item to receive.
Set it to "portMAX_DELAY", to wait indefinitely

6 - Queue

Using queues

1. In file main.c, declare a queue handle variable.
2. In file main.c, function main, initialize the queue (max 5 elements)
3. Write a "sender" task that adds a random integer every 100ms on a queue
4. Write a "receiver" task that removes the integer from the same queue every 100ms.

7 - Task notification

6 - Task notification

- Task notifications are used to simplify communication between tasks
- A direct task notification is an event sent directly to a task that can unblock the receiving task

Send a task notification

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

Always use (pdTRUE, portMAX_DELAY)

Wait for a notification

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

6 - Task notification

Using task notifications

1. Create 2 tasks. Each task prints a string every 100 ms.
 Each task must wait for a notification before printing the string.
 Each task must send a notification after printing the string.
 The first task prints the first string without waiting for a notification.