The Abdus Salam
International Centre
for Theoretical Physics

***Advanced Workshop on modern FPGA-based technology for Scientific Computing***

# FreeRTOS Operating system for SoC


Fernando Rincón

*fernando.rincon@uclm.es*

# Contents

- Motivation for using FreeRTOS
- Some facts about FreeRTOS
- FreeRTOS in the Zynq
- FreeRTOS programming abstractions
  - Tasks
  - Queues
  - Other synchronization primitives

# Motivation

- Two main alternatives for firmware development for microcontrollers
  - Baremetal
  - Based on a O.S.
- The baremetal approach, based on a superloop:
  - forever loop that sequences the set of tasks
  - Polled or interrupt-based I/O
  - Typical in standalone implementations
  - Pros:
    - Simple
    - No OS overhead
  - Cons
    - Difficult to scale (low number of tasks)
    - Difficult to balance time and tasks priorities

```
int main() {
    init_system();
    …
    While(1) {
        do_a();
        do_b();
        do_c();
    }
    // You'l never get here
}
```

# Motivation

- Based on a O.S.
  - Multi-threaded: multiple threads spawn to carry out multiple tasks concurrently
  - Each task has different priority and timing requirements
  - The operating system provides some hardware abstraction layer
  - Extra services, such as a filesystem, network stack, ...
  - Pros:
    - More modular architecture
    - Tasks can be pre-empted. Avoid priority inversion
  - Cons:
    - More complex and extra overhead
    - Higher memory requirements
    - Thread execution is difficult to test
    - Deterministic??

# FreeRTOS

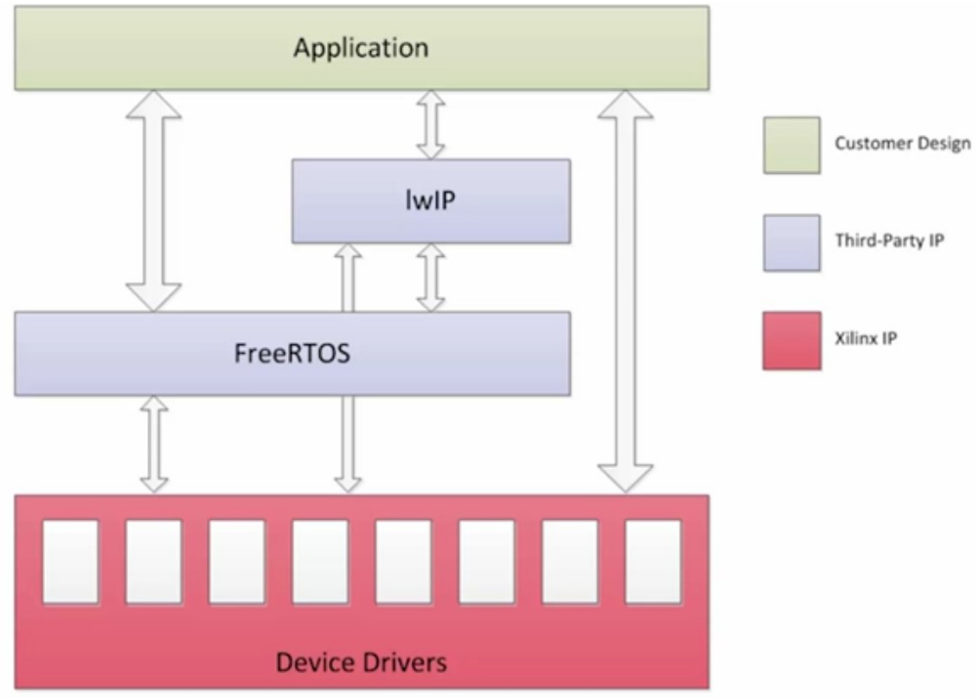
- Born in 2003 and initially conceived for microcontrollers
  - Really light
  - Really simple: the core of the O.S. are just 3 C files
  - Minimal processing overhead
    - FreeRTOS IRQ dispatch 10 cycles aprox.
    - Embedde Linux IRQ dispatch = 100 cycles aprox.
    - Ported to a large number of architectures
- Currently is Amazon the company that stewards the development of the O.S.
- Open Source MIT license
- More information at www.FreeRTOS.org

# FreeRTOS

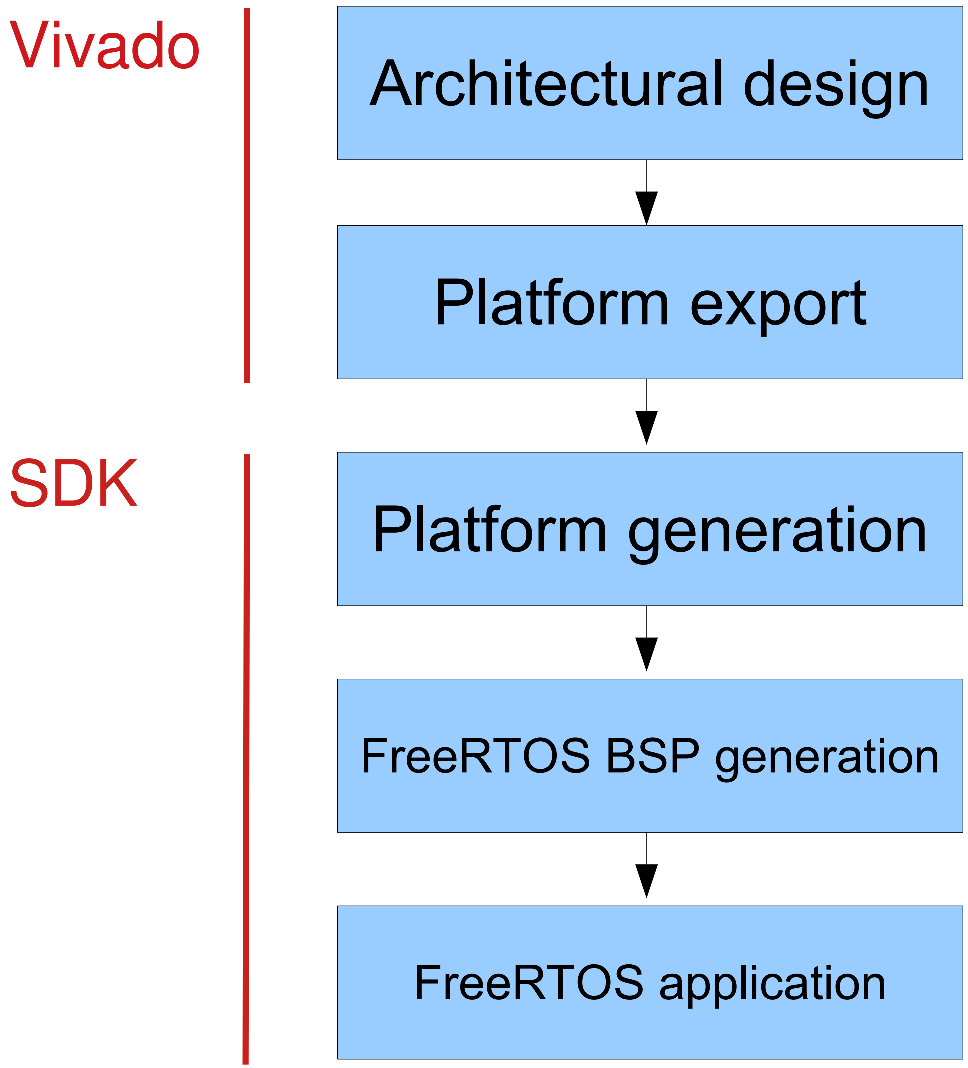
- An ecosystem of products:
  - Amazon FreeRTOS for IoT devices
  - Network communication stack
  - Command Line Interface
  - SSL and TLS security
  - FAT file system

# FreeRTOS & Zynq

- FreeRTOS completely integrated in Xilinx Software Development Flow
- Provided as a BSP:
  - Extension of the standalone BSP
    - All low level drivers can be directly used
  - Includes the O.S. runtime
  - Optional extensions:
    - Filesystem
    - Network
    - ...



# FreeRTOS Design Flow




This information will be used for the generation of the appropriate drivers for the peripherals

It includes the standalone drivers plus the extra libraries selected

Based on the FreeRTOS API plus the peripheral drivers

# FreeRTOS Configuration

- Through a header file: FreeRTOSConfig.h

```
#define configUSE_PREEMPTION 1

#define configUSE_MUTEXES 1

#define INCLUDE_xSemaphoreGetMutexHolder 1

#define configUSE_RECURSIVE_MUTEXES 1

#define configUSE_COUNTING_SEMAPHORES 1

#define configUSE_TIMERS 1

#define configUSE_IDLE_HOOK 0

#define configUSE_TICK_HOOK 0

#define configUSE_DAEMON_TASK_STARTUP_HOOK 0

#define configUSE_TICKLESS_IDLE 0
#define configTASK_RETURN_ADDRESS    NULL
#define INCLUDE_vTaskPrioritySet           1
#define INCLUDE_uxTaskPriorityGet          1
#define INCLUDE_vTaskDelete                1
#define INCLUDE_vTaskCleanUpResources      1
#define INCLUDE_vTaskSuspend               1
#define INCLUDE_vTaskDelayUntil            1
#define INCLUDE_vTaskDelay                 1
#define INCLUDE_eTaskGetState              1
#define INCLUDE_xTimerPendFunctionCall     1
#define INCLUDE_pcTaskGetTaskName          1
#define configMAX_API_CALL_INTERRUPT_PRIORITY (18)
```

Tasks can be interrupted by others with higher priority

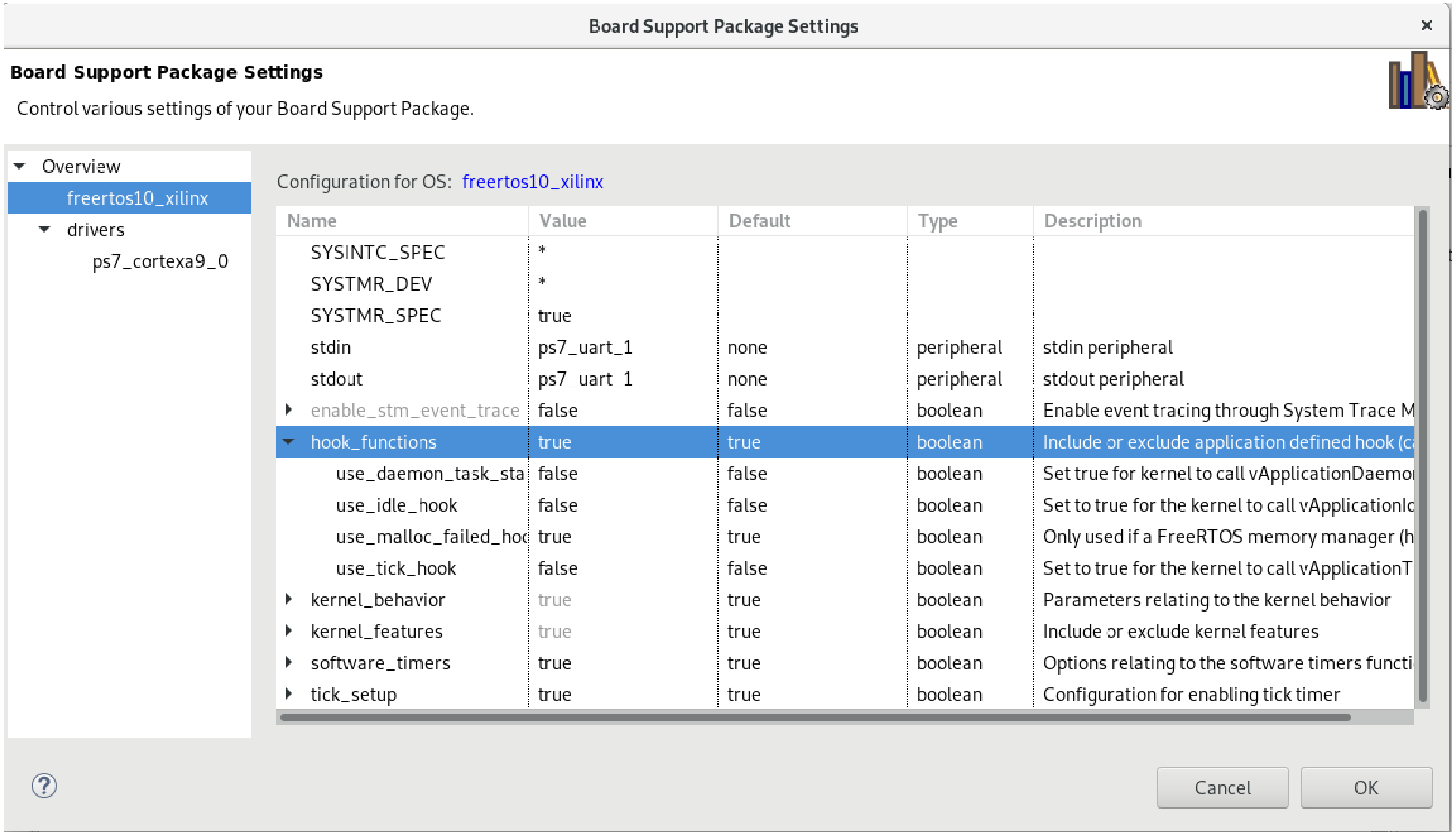
This will include a timer service task

Hooks are used to trigger the execution of functions upon the happening of certain events

Some functionality can be optionally included/excluded from the core of the O.S.

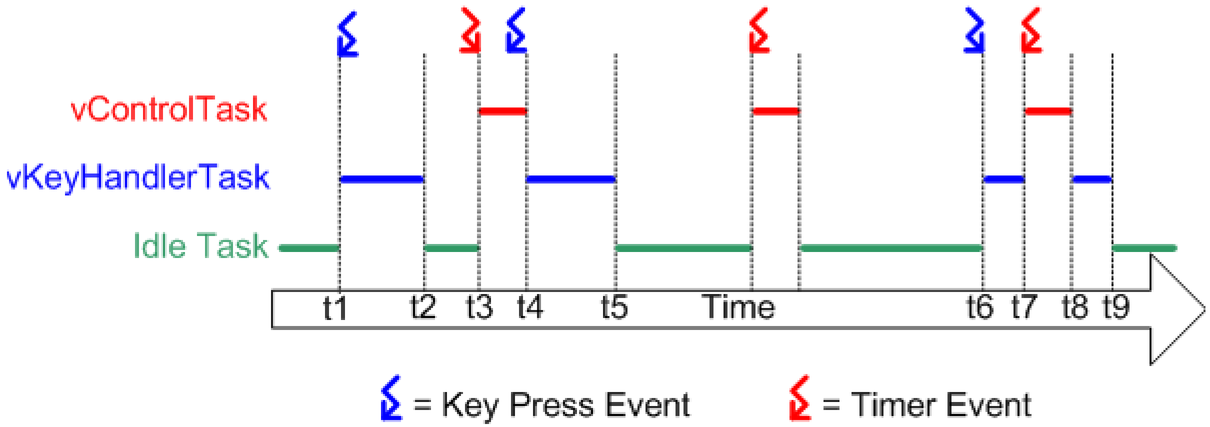
# FreeRTOS Configuration

- The Xilinx way to handle configuration is through the mss file in the FreeRTOS BSP generated in the SDK



# FreeRTOS tasks

- Every thread of execution is a task
- Tasks are independent between them. They have their own execution context (memory)
- Tasks are never called from the program
- Tasks are executed by the FreeRTOS scheduler **depending on their priorities and as a response to events**
- Only one task active at the same time
- Tasks never return
- There's a special IDLE task
  - No need to create it



# FreeRTOS tasks

- A typical FreeRTOS application will look like this

```
void main()
{
  xTaskCreate (Task_A, ….);
  xTaskCreate (Task_A, ….);
  xTaskCreate (Task_A, ….);
  xTaskStartScheduler ();
}
```

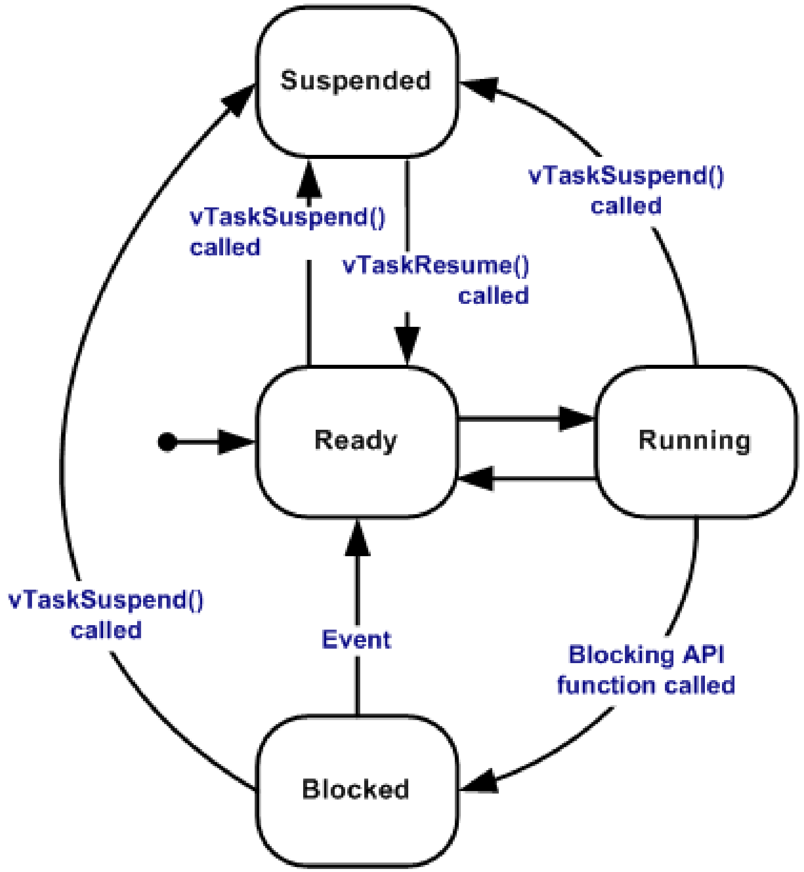
```
void Task_A ()
{
  Init_A();
  while (1)
  {
     do_A();
  }
}
```

```
void Task_B ()
{
  Init_B();
  while (1)
  {
    do_B();
  }
}
```

```
void Task_C ()
{
  Init_C();
  while (1)
  {
    do_C();
  }
}
```

# FreeRTOS task model

- Tasks can be in different states of execution
  - **Ready**
    - When the task can be selected for execution, but is kept waiting since the CPU is busy with another task (depends on priority – next slide)
  - **Running**
    - Really executing the code
  - **Blocked**
    - Waiting for something:
      - An event. (e.g. a message has been received in a queue)
      - vTaskDelay() has been called so a certain time must pass.
  - **Suspended**
    - After calling vTaskSuspend()
    - Can later be resumed using xTaskResume()



# FreeRTOS priorities

- Tasks have priorities, used to the scheduler to select the most urgent one
- The range of different priorities is configurable in *FreeRTOSConfig.h*
  - configMAX_PRIORITIES
- Tasks can change their own priority, as well as the priority of other tasks.
- The IDLE task is the one with the lowest priority
  - tskIDLE_PRIORITY = 0
- The FreeRTOS scheduler is preemptive:
  - If a task with a higher priority that the actual one is READY, then the RUNNING one will be evicted and moved to the READY state, while the former will start the execution

# FreeRTOS tasks creation

- Tasks are modelled after normal C functions

```
static void prvTxTask( void *pvParameters )
```

  - void return:
    - And remember in fact they should never return
  - void pointer for arguments. Can be later casted to the right type
- Since not called, they must be registered (created) into the scheduler
  - The IDLE task is created automatically (special case)
- Can also be destroyed at run-time
- Some related functions:
  - xtaskCreate()
  - xtaskDelete()

# FreeRTOS Tasks

- In order to create a Task:

```
BaseType_t xTaskCreate(TaskFunction_t pxTaskCode,
                       const char * const pcName,
                       const configSTACK_DEPTH_TYPE usStackDepth,
                       void * const pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t * const pxCreatedTask
```

  - **pxTaskCode**: pointer to the function that really implements the task
  - **pcName:** name assigned, mainly used for debug purposes
  - **usStackDepth:** refers to the local memory assigned to the task
    - The **configMINIMAL_STACK_SIZE** parameter set in the FreeRTOSConfig.h configuration file
  - **pvParameters**: since no parameters are sent to the task
  - **uxPriority**: priority assigned to the task.
    - This constant is defined as the minimum possible priority
    - The lowest the number, the lowest the priority
  - **pxCreatedTask**: task handler
    - Previously declared as:

```
static TaskHandle_t xTxTask;
```

```
xTaskCreate(    prvTxTask,
                ( const char * ) "Tx",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY,
                &xTxTask );
```

Task creation example

# FreeRTOS hello world

**3.** Once the scheduler is started, functions will be executed depending on the schedul-ing policy

```
#define TASK_NAME "HelloTask"
#define TASK_STACKDEPTH 1000
#define TASK_PRIORITY 1
#define TASK_PARAMETER NULL
#define TASK_HANDLE NULL


void sayHello( void *pvParameters )
{
  while (1) {
    printf("hello\n");
    vTaskDelay(1000 / portTICK_RATE_MS);
  }
}

int main( void )
{
  xTaskCreate( sayHello, TASK_NAME, TASK_STACKDEPTH,
               TASK_PARAMETER, TASK_PRIORITY, TASK_HANDLE );
  vTaskStartScheduler();

  printf("Something wrong\n");
  return 0;
}
```

**1. main** function is normally used to create at least one task

**2.** The scheduler is a never-ending loop, so the program should never get to this point

# FreeRTOS hello world

- sayHello task activation:
  - Once the scheduler is started, the task becomes ready
  - Since it’s the only task apart from the IDLE one (always present) it will be scheduled to RUN.
  - There are no other tasks but the IDLE one, with lower priority, so the task is always chosen to RUN.
  - But when the task executes vTaskDelay to force a wating time, it becomes BLOCKED, waiting for the time to pass
  - Once the time has passed,
    - The task will be moved to the READY state
    - The IDLE task (priority 0) will be evicted
    - The sayHello task will move to RUN

# FreeRTOS Task Communication

- Two mechanisms:
  - Global variables which can be read from all tasks
  - Queues as the main mechanism for inter-task communication
- Queues:
  - Asynchronous model of communication based on a FIFO
  - Data can written to both the head and tail of the queue
  - Of arbitrary size and depth, but defined at compile time
  - Items are passed by value $\rightarrow$ not zero copy
  - Access can be blocking or non-blocking

# Global variables and their risks

- The global variable is shared by all tasks
- Access control should be managed by the programmer
  - Since processes can be evicted, the state can be inconsistent
- E.g.:
  - One process writes and another reads: Ok
  - Two processes write
    - You may assume wrong states
    - Need for explicit synchronization mechanisms such as locks

# FreeRTOS queues

- Queue creation:

```
xQueueHandle xQueueCreate (unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize)
```

- Queue data insertion at the back of the queue:

```
portBASE_TYPE xQueueSendToBack (xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait)
```

  - If *xTicksToWait* is 0 it will return immediately if full otherwise it will wait

- Data insertion at the front of the queue:

```
portBASE_TYPE xQueueSendToFront (xQueueHandle xQueue,
                                 const void * pvItemToQueue,
                                 portTickType xTicksToWait)
```

- Data extraction:

```
portBASE_TYPE xQueueReceive (xQueueHandle xQueue,
                             void * pvBuffer,
                             portTickType xTicksToWait)
```

# FreeRTOS queues

- The producer-consumer example

```
xQueueHandle queue;

void producer( void *pvParameters ) {
  int value = 0;

  while (1) {
    xQueueSendToBack(queue, &value, 0);
    value++;
    vTaskDelay (1000 / portTICK_RATE_MS);
  }
}

void consumer( void *pvParameters ){
  int value;

  while (1) {
    xQueueReceive(queue, &value, portMAX_DELAY);
    printf("value received: %d\n", value);
    vTaskDelay (1000 / portTICK_RATE_MS);
  }
}


int main( void ) {
  queue = xQueueCreate(100, sizeof(int));

  xTaskCreate( producer, P_TASK_NAME, TASK_STACKDEPTH, TASK_PARAMETER,
            TASK_PRIORITY, TASK_HANDLE );
  xTaskCreate( consumer, C_TASK_NAME, TASK_STACKDEPTH, TASK_PARAMETER,
            TASK_PRIORITY, TASK_HANDLE );
  vTaskStartScheduler();
}
```

Queue declaration

If the queue is full, it will return immediately

Blocking read

Queue creation with limited size

Be careful with priorities

# FreeRTOS synchronization

- Queues can also be used as a synchronization primitive
- But FreeRTOS includes some other types:
  - Binary semaphores
    - SemaphoreHandle_t xSemaphoreCreateBinary( void );
    - Used to prevent concurrent access
    - Typically used in Interrupt Service Routines (ISR)
  - Counting semaphores can be used in two scenarios
    - Counting events:
      - An event generator gives a semaphore for each event
      - Another task will take the event
      - The count value is the difference
    - Resource management
      - The semaphore tells the number of available instances of a resource
  - Mutexes
    - SemaphoreHandle_t xSemaphoreCreateMutex( void )
    - Similar to binary semaphores but the task taking the semaphore inherits the priority