FreeRTOS

Fabrice Muller

Polytech Nice Sophia

Fabrice.Muller@univ-cotedazur.fr

Introduction

Course mainly based on document : *Mastering the FreeRTOS™ Real Time Kernel, A Hands-On Tutorial Guide, Richard Barry*

https://www.freertos.org/

FreeRTOS

- Portable
- Open source
- Royalty free
- (Mini) Real Time Operating System
- No Input/output libraries (driver)
  - USART, I2C, SPI, CAN …
- Dedicated for microcontroller systems
  - No graphical interface
  - No I/O hard disk (SATA, SCSI …)
  - No formatting management (FAT …)

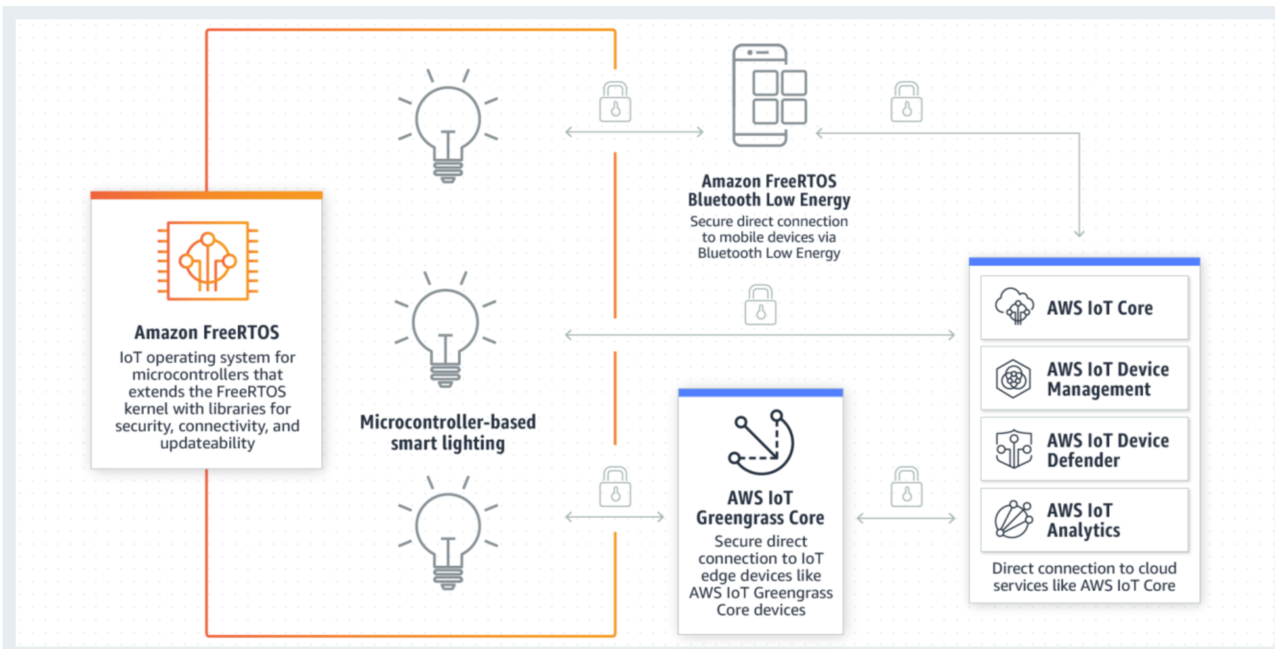
- https://www.freertos.org/

Amazon FreeRTOS

- Amazon FreeRTOS extends the FreeRTOS kernel
  - FreeRTOS
  - Amazon Web Services (AWS), additional libraries
- Easily program, deploy, and manage low-power connected devices
  - Includes libraries to configure devices to a local network (Wi-Fi or Ethernet), or connect to a mobile device using Bluetooth Low Energy
  - Includes an over-the-air (OTA) to remotely update devices
- Secure data and device connections
  - Includes support for data encryption and key management
  - Includes support for Transport Layer Security (TLS)
- Broad hardware and technology ecosystem
  - Builds IoT solutions on a variety of chipsets
  - Supports a variety of architectures
- https://aws.amazon.com/freertos/

Amazon FreeRTOS - How it works

- Connected microcontroller-based devices and collect data from them for IoT applications
- AWS cloud platform offers over 165 fully featured services
- https://aws.amazon.com/freertos/



Main functionnalities

- Real-Time (RT) : preemptive / cooperative scheduler
- Small kernel (4Kb to 9Kb)
- Easy to use with C language
- Illimited task number and level of priority
- Flexible management of priorities
- Communications (inter-tasks / tasks-interrupts)
  - Queues
  - Semaphore (Binary, Counting, recursive)
  - Mutex (Mutual Exclusion, priority inversion)
- Software timer
- Stack overflow checking
- Idle hook function
- Trace

Official Platforms supported

Combination of compiler and processor is considered to be a separate **FreeRTOS port**

- ARMv8-M
- Atmel
- Cadence
- Cortus
- Cypress
- Espressif ESP32
- Freescale
- Infineon
- Fujitsu
- Microchip
- Microsemi

Labs

- Nuvoton
- NXP
- Renevas
- SiFive
- Silicon Labs
- Spansion
- ST Microelectonics
- Texas Instrument
- Xilinx
- Intel/x86, Intel/FPGA (ex Altera)

https://www.freertos.org/RTOS_ports.html

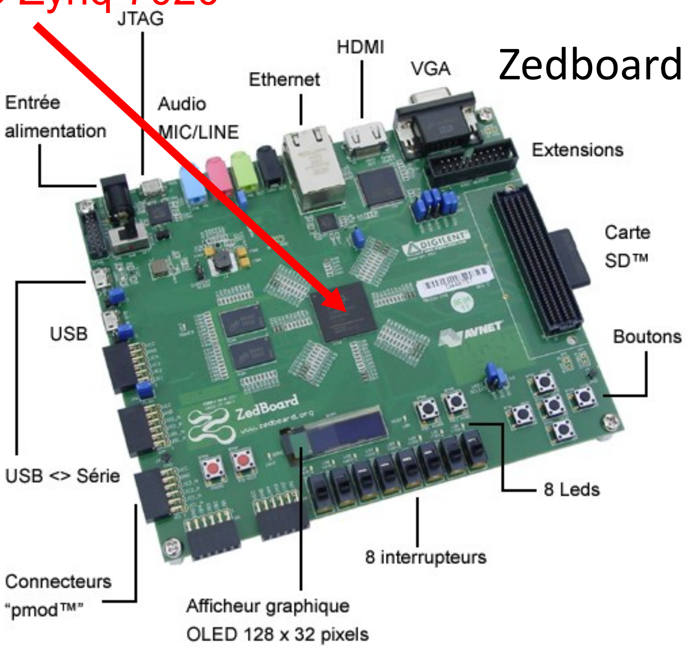
Intel/x86 - Windows simulator

- To be run in a Windows environment
- True real time behavior cannot be achieved
- Visual Studio projects / Eclipse with MingW (GCC)
- How to use it
  - https://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html

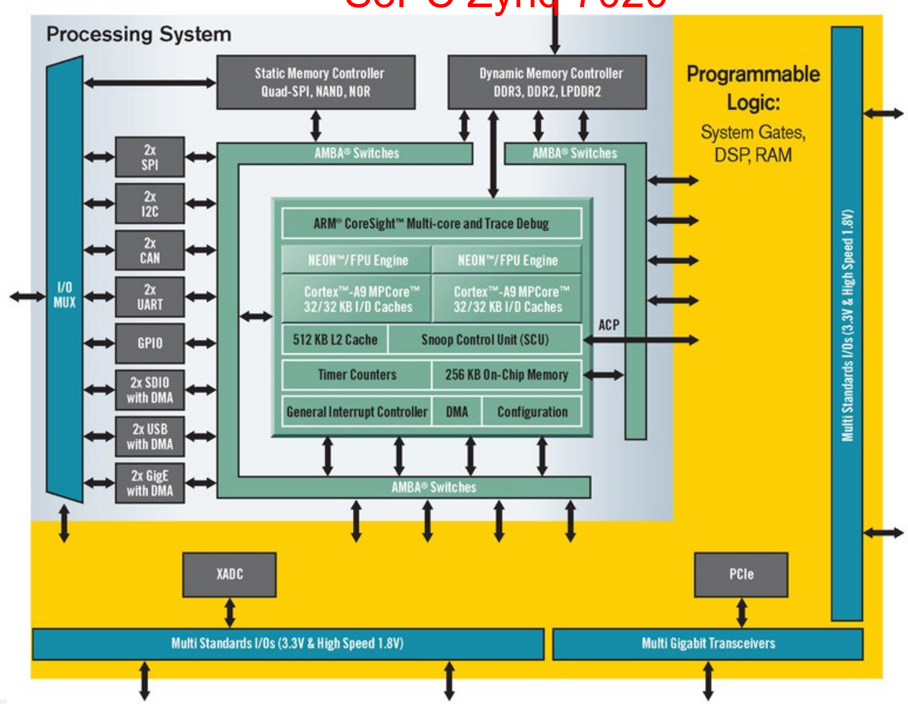
Xilinx – Zynq device

- using a FreeRTOS BSP on Xilinx SDK
- pre-configured FreeRTOS environment that does not require any source files
- How to use it
  - https://www.freertos.org/RTOS-Xilinx-SDK-BSP.html

SoPC Zynq 7020



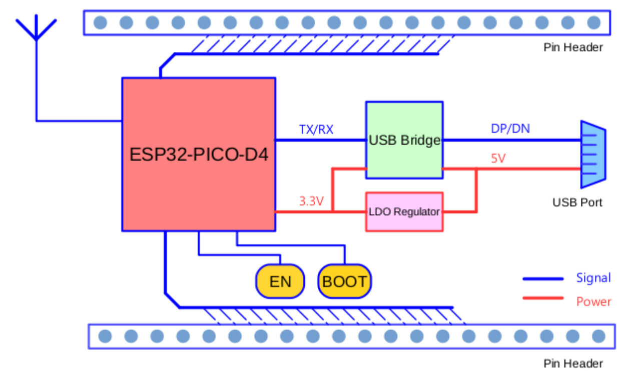
SoPC Zynq 7020

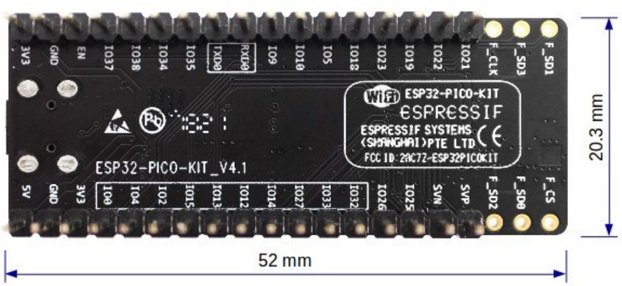


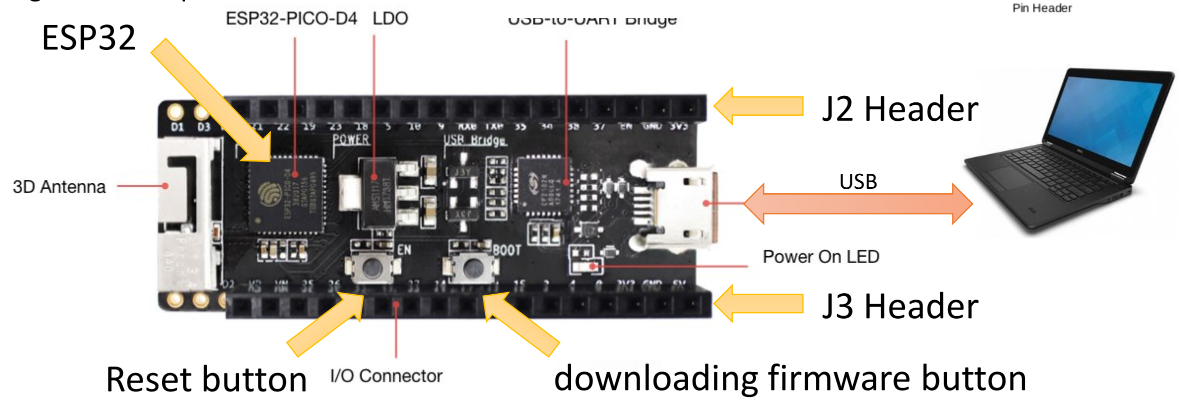
Espressif – ESP32-PICO-KIT Board

Useful for Labs

- System-in-Package (SiP) : ESP32-PICO-D4
- Including
  - 40 MHz crystal oscillator
  - 4 MiB flash
  - Filter capacitors and RF matching links in
- USB-UART bridge (up to 3 Mbps transfers rates)
- Buttons
  - BOOT : press for downloading firmware through the serial port.
  - EN: Reset





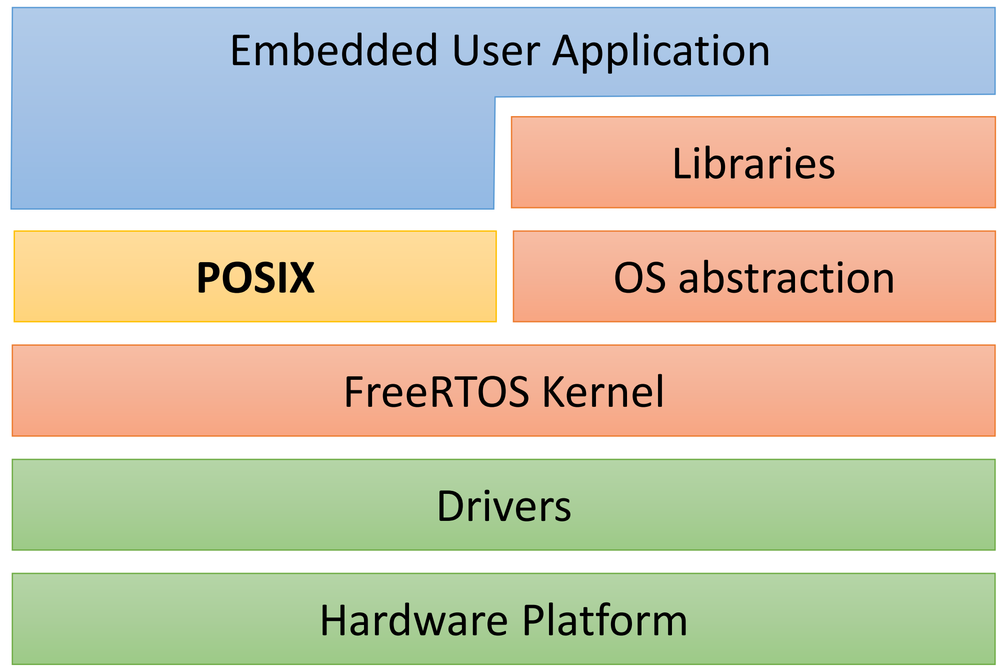


https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html

FreeRTOS & POSIX

- POSIX = Portable Operating System Interface
- Standard specified by the IEEE Computer Society for maintaining compatibility between operating systems
- Implementation of a subset of the POSIX threading API
- Subset of IEEE Std 1003.1-2017

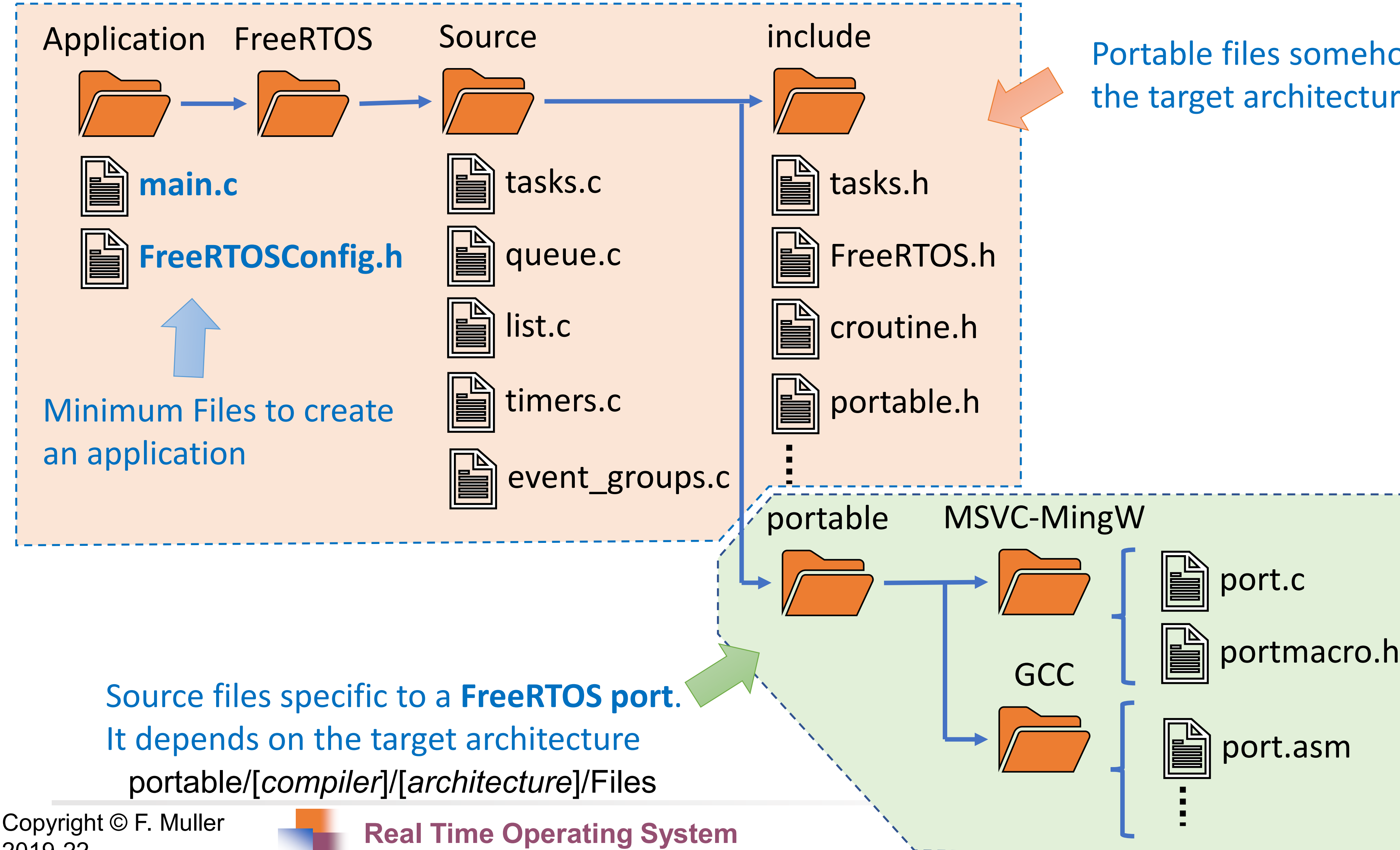


https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_POSIX/index.html

Organization of FreeRTOS

Version 10.4.3

Top Directories



Source files specific to a **FreeRTOS port**.
It depends on the target architecture
portable/[*compiler*]/[*architecture*]/Files

Application - FreeRTOSConfig.h

- **FreeRTOSConfig.h** file is used to tailor FreeRTOS for use in a specific application
- Ensures the kernel configuration before compilation
- Configuration of the API and functionalities
  - #define configUSE_PREEMPTION 1
  - #define configUSE_MUTEXES 1
  - #define configUSE_COUNTING_SEMAPHORES 1
  - …
- All active macros (=1), footprint will be maximal on memory
- Every demo application contains a FreeRTOSConfig.h file
- Located in FreeRTOS/Demo/[*Target Architecture*]
  - Example: *FreeRTOS/Demo/CORTEX_A9_Zynq_ZC702*

Application – Main file

- Just create a *main.c* file

```c
int main( void ) {
   /* Perform any hardware setup necessary. */
   prvSetupHardware();

   /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

   /* Start the created tasks running. */
   vTaskStartScheduler();

   /* Execution will only reach here if there was insufficient heap to start the scheduler. */
   for( ;; );
   return 0;
}
```

Application – Main file with ESP-IDF

- Just create a *main.c* file
- *app_main()* is a task, not a function !

```
void app_main( void ) {

  /* --- APPLICATION TASKS CAN BE CREATED HERE --- */



  /* to ensure its exit is clean */
  vTaskDelete(NULL);
}
```

Compilers – Include Paths

- 3 directories to be included in the compiler's include path

- The path to the core FreeRTOS header files
  - FreeRTOS/Source/include
- The path to the source files that are specific to the FreeRTOS port
  - FreeRTOS/Source/portable/[compiler]/[architecture]
- A path to the FreeRTOSConfig.h header file

Development tool ESP32

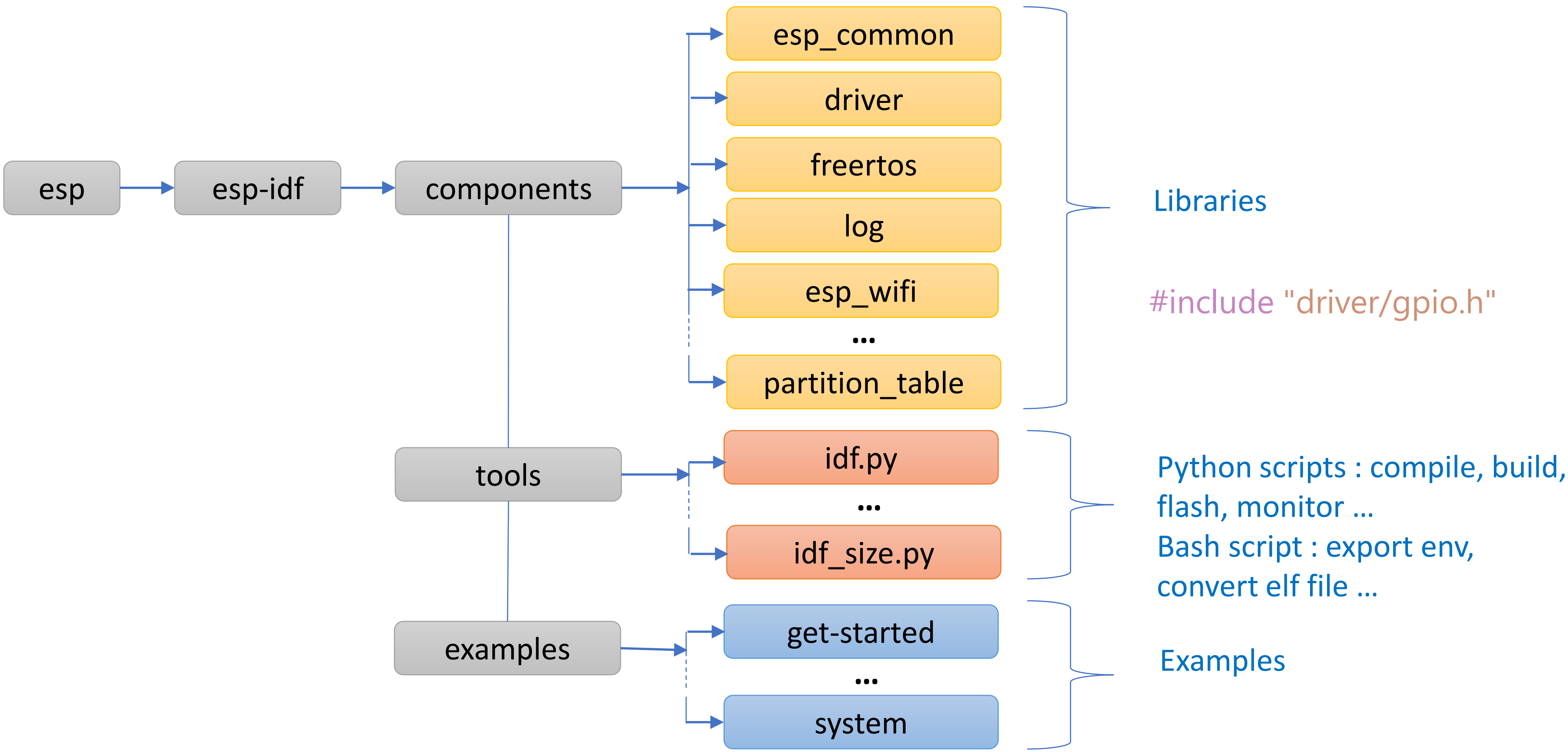
Espressif IoT Development Framework

ESPRESSIF

Espressif IoT Development Framework

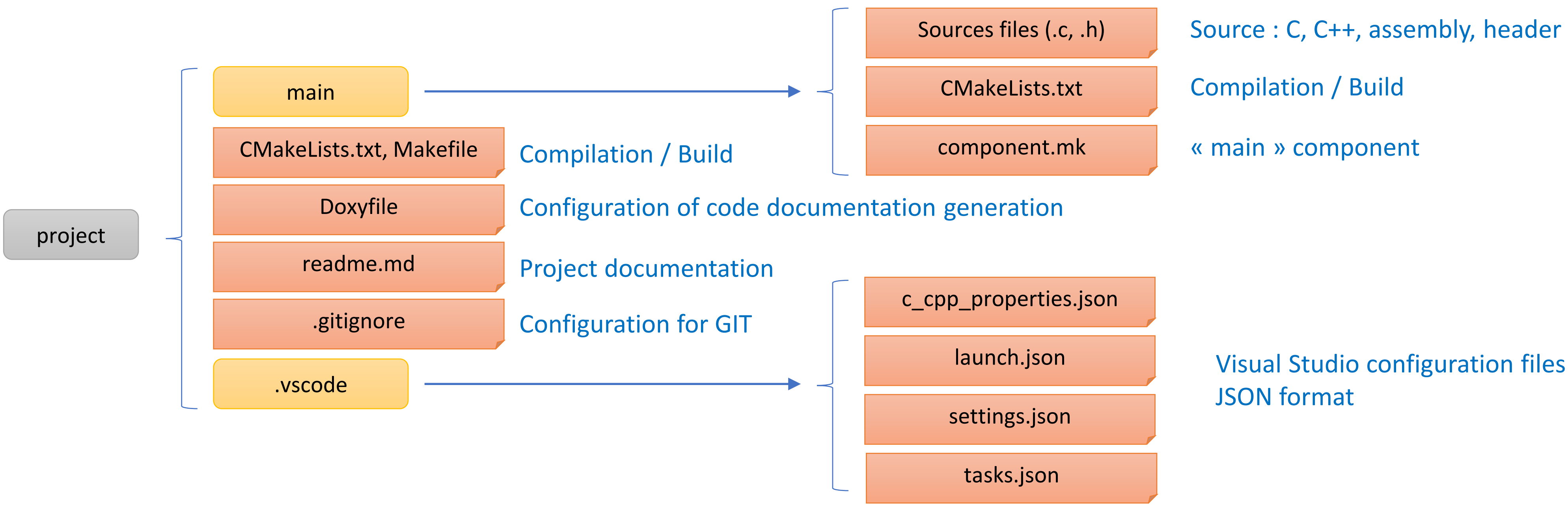
- **Esp**ressif **I**oT **D**evelopment **F**ramework = **ESP-II** ESPRESSIF
- Included
  - Libraries
  - Tools
  - Examples
- ESP-IDF Programming Guide
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/
- Labs: ESP-IDF v4.4 with FreeRTOS v10.4.3

ESP-IDF folder structure



ESP32 project template

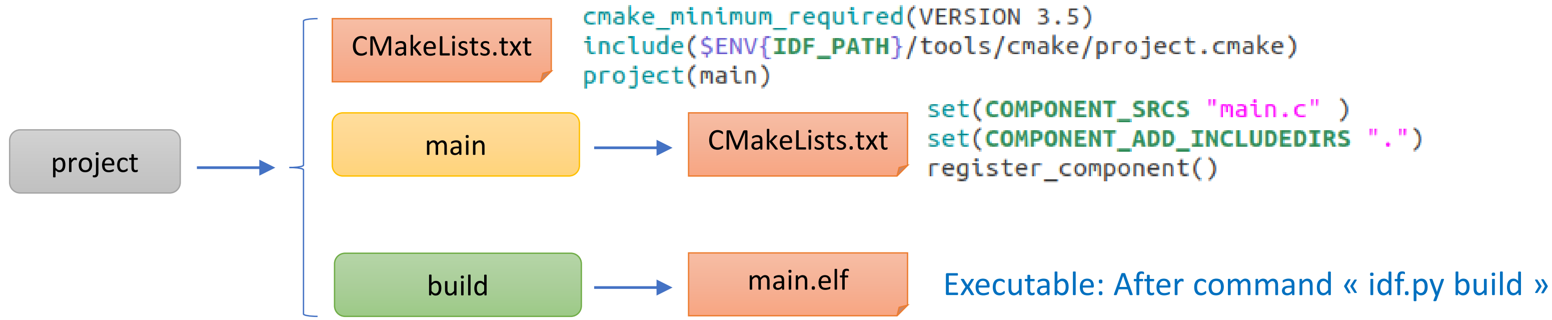
- For Visual Studio Code
- Located in « esp32-vscode-project-template » project
  - https://github.com/fmuller-pns/esp32-vscode-project-template



CMakeLists.txt & CMake

CMake

- CMake (cmake.org)
  - Cross-platform family of tools
  - Designed to build, test and package software
  - Used to control the software compilation process using simple platform and compiler independent configuration files
  - Generate native makefiles
  - Open-source
- File configuration : *CMakeLists.txt*
- ESP32 guide
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html#project-cmakelists-file
- idf.py (Python script) is a wrapper around CMake
  - idf.py build



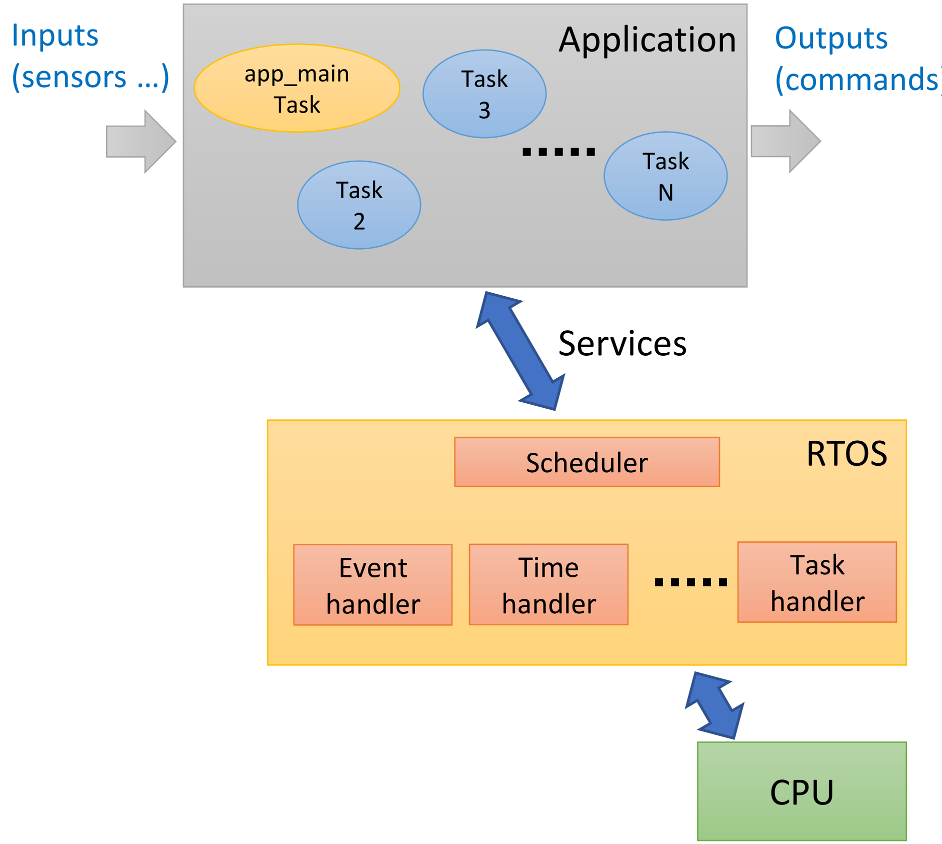
Visual Studio Code

- *.vscode* folder including configuration
- JSON format (JavaScript Object Notati
- Environment
  - IDF_TOOLS
  - IDF_PATH
- Configuration : esp32
  - name, browse
  - **includePath: important for components**
- Miscellaneous
  - cStandard : c11 (ISO/IEC 9899:2011)
  - cppStandard : c++17 (ISO/IEC 14882)

```
{
    "env": {
        "IDF_TOOLS": "~/.espressif/tools",
        "IDF_PATH": "~/esp/esp-idf"
    },
    "configurations": [
        {
            "name": "esp32",
            "browse": {
                "path": [
                    "${workspaceFolder}",
                    "${IDF_PATH}",
                    "${IDF_TOOLS}"
                ],
                "limitSymbolsToIncludedHeaders": true
            },
            "includePath": [
                "${workspaceFolder}",
                "${workspaceFolder}/build/config",
                "${workspaceFolder}/build/bootloader/config",
                "${IDF_TOOLS}/xtensa-esp32-elf/esp-2019r2-8.2.
                "${IDF_TOOLS}/xtensa-esp32-elf/esp-2019r2-8.2.
                "${IDF_TOOLS}/xtensa-esp32-elf/esp-2019r2-8.2.
                "${IDF_TOOLS}/xtensa-esp32-elf/esp-2019r2-8.2.
                "${IDF_PATH}/components/newlib/include",
                "${IDF_PATH}/components/esp32/include",
                "${IDF_PATH}/components/soc/esp32/include",
    ],
    "defines": [],
    "cStandard": "c11",
    "cppStandard": "c++17",
    "intelliSenseMode": "clang-x64"
        }
    ],
    "version": 4
```

Using FreeRTOS on ESP32 boards

- RTOS = Real Time Operating System
- Starting point
  - *app_main()* task
- Input/output management
  - Input/output handler
  - Interrupt handler
- Task scheduling
  - Organization of functioning in tasks
  - Scheduling policy
  - Time handler
- Inter task communications
  - Synchronization (event)
  - Communication (data)
  - Access to a shared resource (data)
  - Time (counter, watchdog)



Coding Style

Base Types

- Define in *portmacro.h* header file
- Most efficient data type for the architecture
  - UBaseType_t, BaseType_t
  - 32-bit type on a 32 bit architecture, 16-bit type on a 16 bit architecture …
- Specific types
  - portCHAR, portLONG, portSHORT
  - portFLOAT, portDOUBLE
  - portBASE_TYPE
- Useful Constants
  - pdTRUE, pdFALSE
  - pdPASS, pdFAIL

Variable prefix names

Base prefix names

- c : char
- s : short
- l : long
- x : portBASE_TYPE

Other prefix names

- p : pointer
- u : unsigned
- v : void

```
BaseType_t uxMyBoolVar = pdTRUE;             // u: unsigned, x: Base Type
portSHORT *psMyVar1;                          // p: pointer, s: short
const portCHAR *pcTaskName = "Task 1\r\n";// p: pointer, c: char
```

Function prefix names

Like variable name

- c, s, l, x
- p, u, v

\+

File name where it defined

- Task : task.c
- Semaphore : semphr.h
- Queue : queue.c
- Timer : timers.c
- …

```
// Function, v: function returns void
void vInit(UBaseType_t uxPriority) {    // u: unsigned, x: base type

  vTaskPrioritySet(vTask1, uxPriority); // return (v: void), Task: task.c
  BaseType_t xVar = xQueueReceive(...); // return (x: base type), Queue: queue.c
  void *pvId = pvTimerGetTimerID(...);  // return (p: pointer, v: void), Timer: timer.c
  vSemaphoreCreateBinary(...);          // return (v: void), Semaphore: semph.c
  ...
}
```

Macro Names

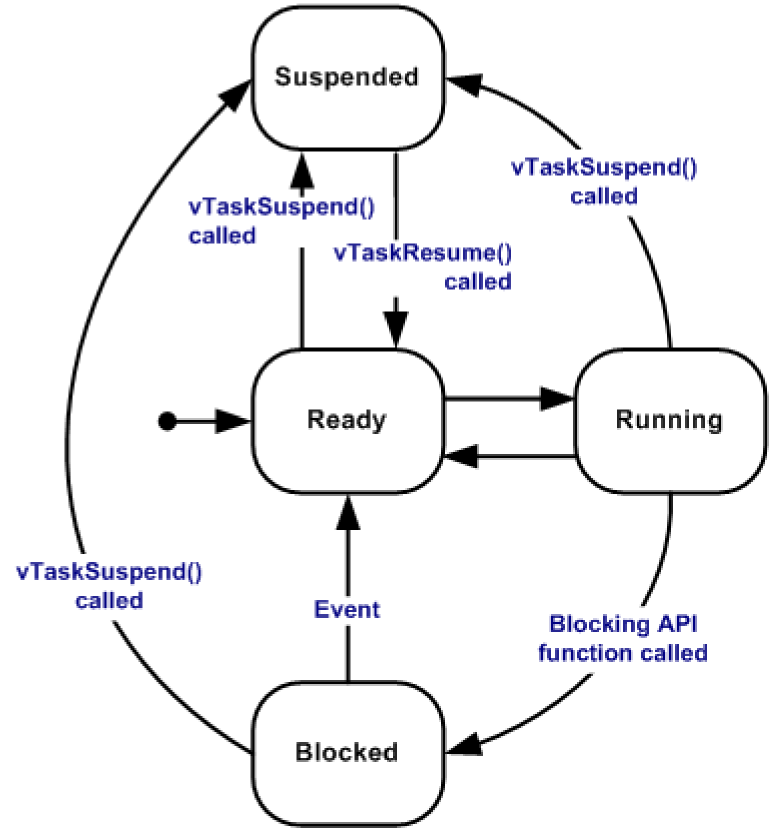
- Most macros
  - Written in upper case
  - Prefixed with lower case letters

Prefix	Location	Example
port	portable.h / portmacro.h	portMAX_DELAY, portDOUBLE, portINLINE
task	task.h	taskENTER_CRITICAL(), taskENABLE_INTERRUPTS()
pd	projdefs.h	pdFALSE, pdMS_TO_TICKS
config	FreeRTOSConfig.h	configUSE_PREEMPTION, configUSE_IDLE_HOOK
err	projdef.h	errQUEUE_BLOCKED, errQUEUE_FULL, errQUEUE_EMPTY
…	…	…

Task Management

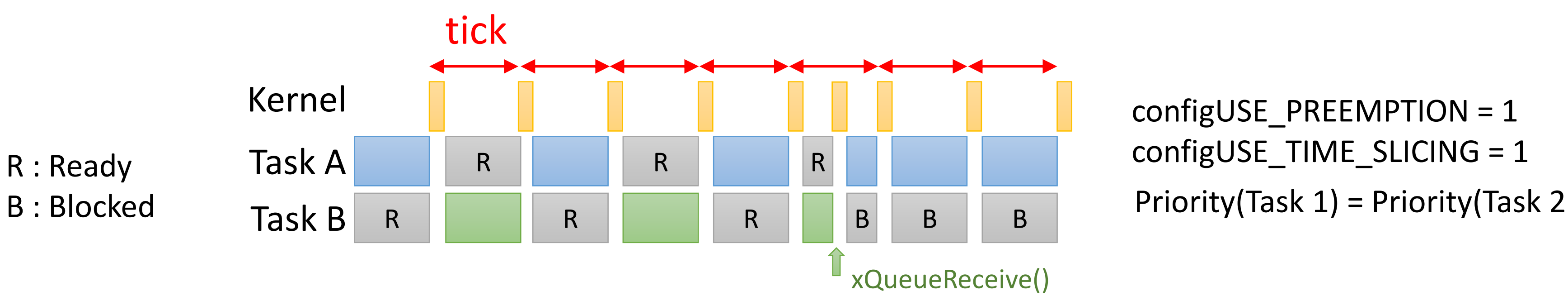
Task States

- Running
  - Task is actually executing
- Ready
  - Tasks are those that are able to execute (Ready list)
- Blocked
  - Tasks are currently waiting for either a temporal or external event (delay, queue, semaphore …)
  - Tasks normally have a timeout period and be unblocked
- Suspended
  - Tasks only enter or exit this state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume()
  - Tasks do not have a time out



Configuration for task scheduling

- Preemption
  - *configUSE_PREEMPTION* macro
  - Cooperative RTOS scheduler (0) : If a low priority T1 task is ahead of a higher priority T2 task, the T1 task continues to execute. **taskYIELD()** function must be call by the task to give the hand to kernel if function API is not used by the task.
  - Preemptive RTOS scheduler (1) : can change of task execution at each tick period
- Time Slicing
  - *configUSE_TIME_SLICING* macro
  - No switch between Tasks (0) : Runs the highest priority task that is in the Ready state, but will not switch between tasks of equal priority even if a tick interrupt has occurred
  - Round robin scheduling (1) : Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme



Task implementation – infinite loop

Declaration

Application code inside the infinite loop

```
void vMyTask(void *pvParameters)
{
  const char *pcTaskName = "Task is running\r\n";
  volatile uint32_t ul;

  for (;; ) {

    DISPLAY("%s",pcTaskName);
    /* Simulate a cpu usage */
    for (ul = 0; ul < 0xffffff; ul++);

  }
}
```

- When a task is in blocked, suspended or ready state, the context of the task (variables …) is saved in the TCB (Task Control Block)

Task implementation – task exit

- The application code comes out of the infinite loop
- Must delete the task properly

Declaration

Application code inside the infinite loop

Delete

```c
void vMyTask(void *pvParameters)
{
  const char *pcTaskName = "Task is running\r\n";
  volatile uint32_t ul;
  int counter = 50;

  for (;; ) {
     DISPLAY("%s",pcTaskName);
    /* Simulate a cpu usage */
    for (ul = 0; ul < 0xffffff; ul++);
    /* Exit ? */
    if (counter-- == 0) break;
  }
  /* to ensure its exit is clean */
  vTaskDelete(NULL);
}
```

Task creation

- xTaskCreate() function
- Return pdFAIL or pdPASS

```
BaseType_t xTaskCreate(
       TaskFunction_t pvTaskCode,     /* Pointer to the function that implements the task. */
       const char * const pcName,     /* Text name for the task. */
       uint16_t usStackDepth,         /* Stack depth */
       void *pvParameters,            /* Task parameter. */
       UBaseType_t uxPriority,        /* Task priority. */
       TaskHandle_t *pxCreatedTask);  /* Task handle to reference the task in API calls */
```

Simple Task instance

- Without parameter (NULL)
- Without Task handle (NULL)

```
int main( void ) {

  /* Create task with No parameter, No task handle */
  xTaskCreate(vMyTask, "My Task", 1000, NULL, 1, NULL);

  /* Start the scheduler to start the tasks executing. */
  vTaskStartScheduler();

  for (;; );

  return 0;
}
```

Task instance with parameter

- Parameters is a pointer of void type (void *)

```c
void vMyTask(void *pvParameters) {
  char *pcTaskName;
  volatile uint32_t ul;

  pcTaskName = (char *)pvParameters;

  for (;; )
  {
     DISPLAY("%s",pcTaskName);

    for (ul = 0; ul < 0xffffff; ul++);
  }
}
```

Cast

```c
const char *pcMyTaskName = "MyTask is running\r\n";

int main(void) {
  /* Create task without task handle */
  xTaskCreate(vMyTask, "My Task",
              1000,
              (void*)pcMyTaskName,
              1,
              NULL);

  /* Start the scheduler. */
  vTaskStartScheduler();

  for (;; );
  return 0;
}
```

Cast

Task instance with task handler

- Task handler is used to access on the API
- Useful to change parameters dynamically (priority …)

```
TaskHandle_t xHandleMyTask = NULL;

int main( void ) {
  /* Create task. */
  xTaskCreate(vMyTask, "My Task",
              1000,
              NULL,
              1,
              &xHandleMyTask);

  vTaskStartScheduler();

  for (;; );
  return 0;
}
```

```
void vMyTask(void *pvParameters)
{
  volatile uint32_t ul;
  int count;

  for (;; ) {
    /* Simulate a cpu usage */
    for (ul = 0; ul < 0xffffff; ul++);

    /* Change priority from handler */
    if (count++ > 50)
      vTaskPrioritySet(&xHandleMyTask, 4);
    else
      vTaskPrioritySet(&xHandleMyTask, 1);
  }
}
```

Change priority dynamically

Multiple Instances of a same task

- Each instance
  - Independent (1 TCB & 1 stack per instance)
  - Own local variables
- If they are declared *static*, the variable is shared between the different instances of the task

```c
int main( void ) {
  /* Create 2 task instances of vMyTask() */
  xTaskCreate(vMyTask, "My Task1",
              1000, NULL, 1, NULL);
  xTaskCreate(vMyTask, "My Task2",
              1000, NULL, 1, NULL);

  vTaskStartScheduler();
  for (;; );
  return 0;
}
```

```c
void vMyTask(void *pvParameters) {
  volatile uint32_t ul;
  static int count;

  for (;; ) {
    /* Simulate a cpu usage */
    for (ul = 0; ul < 0xffffff; ul++);

    /* count is a shared variable of the task */
    count++;
    DISPLAY("Count = %d", count);
  }
}
```

ul is a local variable

count is shared by 2 instances

Idle Task

- To ensure there is always at least one task that is able to run
- Idle task is created automatically with the lowest possible priority (tskIDLE_PRIORITY = 0)
- Idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted
- Idle task hook (callback)
  - Idle task hook is a function that is called during each cycle of the idle task
  - Does not call any API functions that might cause the idle task to block
  - Set *configUSE_IDLE_HOOK = 1* to use it

```
void vApplicationIdleHook(void) {
    ...
}
```

Approximated Periodic task

- *vTaskDelay(TickNumber)* to blocked task during TickNumber ticks
- ***pdMS_TO_TICKS*** macro converts time to tick number
- Period depends on execution time of the task
  - Keep the blocked state is relative to the time at which vTaskDelay() was called

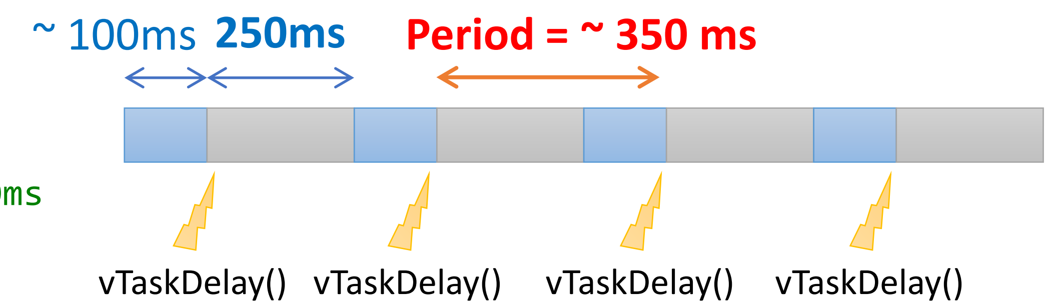
Convert 250ms to tick number

```
void vMyTask(void *pvParameters) {
  char *pcTaskName;
  const TickType_t xDelay250ms = pdMS_TO_TICKS(250UL);

  /* parameter : Task name */
  pcTaskName = (char *)pvParameters;

  for (;; ) {
    DISPLAY("%s",pcTaskName);
    calculationFct();    // duration: 100ms
    vTaskDelay(xDelay250ms);
  }
}
```

100 ms

Task blocked for 250 ms

Period = 100 ms + **250 ms** = **~ 350 ms**



Exactly Periodic task

- Should be used when a fixed execution period is required
- *vTaskDelayUntil(LastTickNumber, TickNumber)* to blocked task during TickNumber ticks relative to last call of *vTaskDelayUntil()*
- Use *xTaskGetTickCount()* function to initialize *LastTickNumber* variable

```
void vMyTask(void *pvParameters) {
  char *pcTaskName;
  TickType_t xLastWakeTime;
  const TickType_t xDelay250ms = pdMS_TO_TICKS(250UL);
  volatile uint32_t ul;

  pcTaskName = (char *)pvParameters;
  xLastWakeTime = xTaskGetTickCount();

  for (;; ) {
    DISPLAY("%s",pcTaskName);
    calculationFct();    // duration: 100ms
    vTaskDelayUntil(&xLastWakeTime, xDelay250ms);
  }
}
```

100 ms

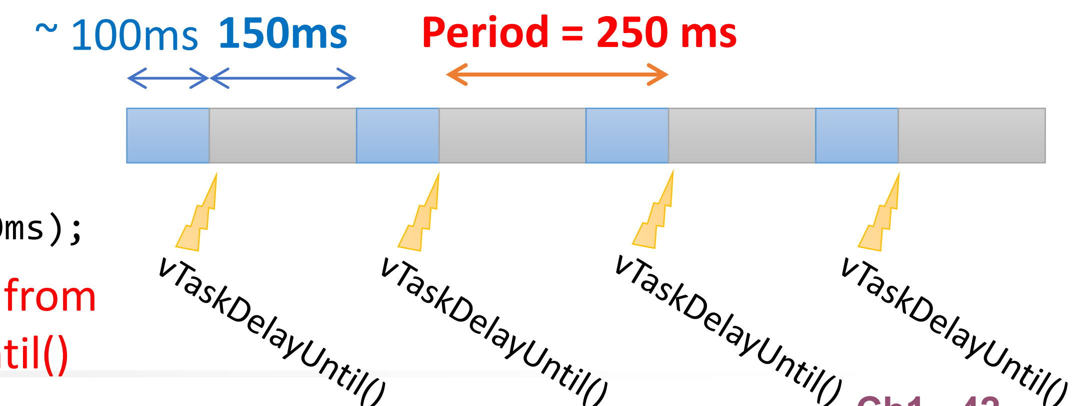
Updated by the vTaskDelayUntil()

Initialize for the first use

Task blocked for 250 ms from last call of vTaskDelayUntil()

Right period !

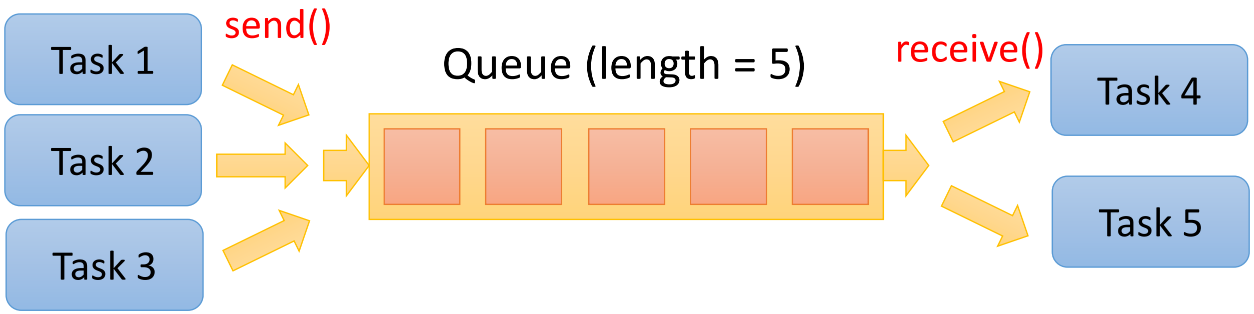
Period = 100 ms + **150 ms** = **<u>250 ms</u>**



Message Queue

Introduction

- FIFO behavior: First in First out
- Length: maximum number of items per queue
- Fixed size data items
- Queue by copy : data sent to the queue is copied byte for byte into the queue
- Classical functions: FIFO behavior
  - Send: written to the end of the queue (Tail)
  - Receive: removed from the front of the queue (Head)
- Extra functions: No FIFO behavior
  - Write item to the front (Head) of a queue
  - Overwrite item that is already at the front of a queue



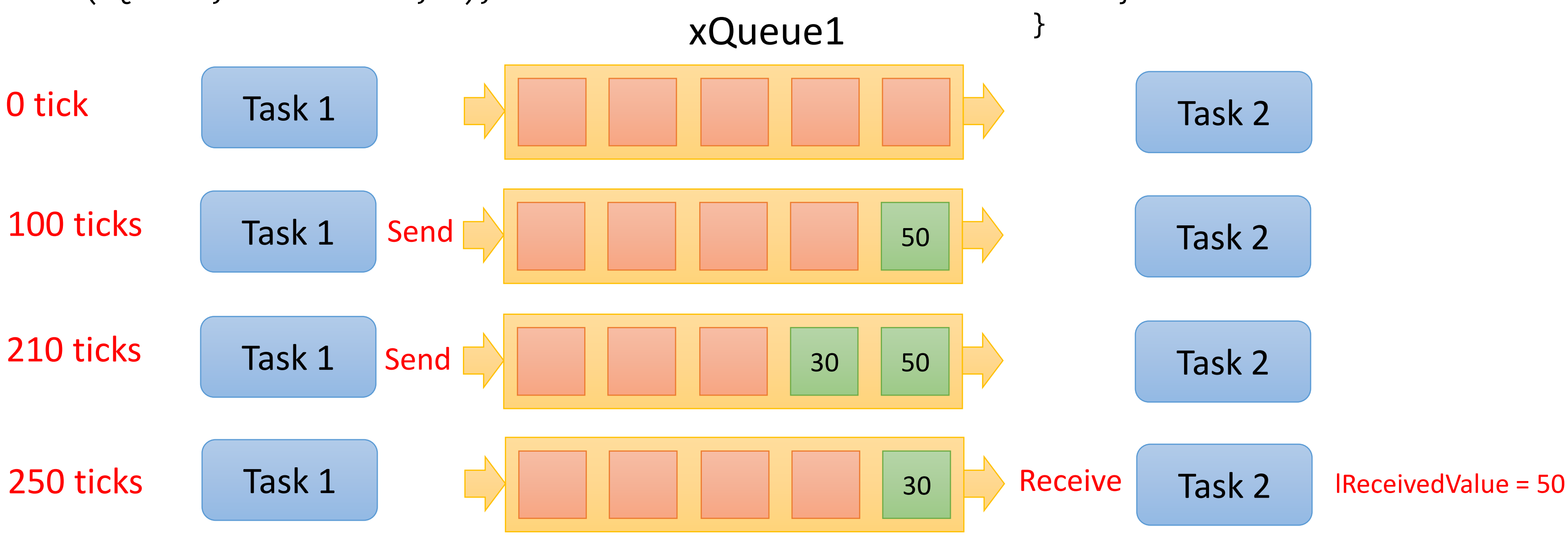
Example of behavior

Global declaration ➡ `QueueHandle_t xQueue1;`

In main() ➡ `xQueue2 = xQueueCreate(1, sizeof(uint32_t));`

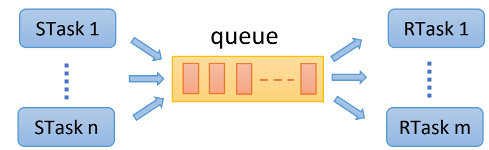
```c
void Task1(void *pvParameters) {
  int32_t lSentValue;
  for (;; ) {
    vTaskDelay(100);
    lSentValue = 50;
    xQueueSend(xQueue1, &lSentValue, 0);
    vTaskDelay(110);
    lSentValue = 30;
    xQueueSend(xQueue1, &lSentValue, 0);
  }
}
```

```c
void Task2(void *pvParameters) {
  int32_t lReceivedValue;
  BaseType_t xStatus;
  for (;; ) {
    vTaskDelay(250);
    xStatus = xQueueReceive(xQueue1,
                &lReceivedValue,
                portMAX_DELAY);
   }
}
```

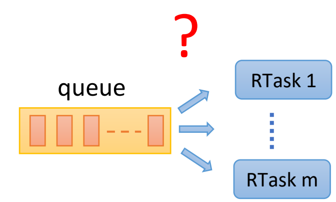


Blocking on single Queue

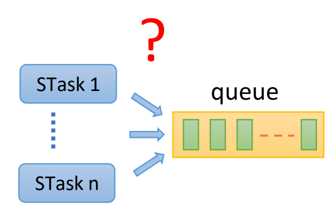
- Access by Multiple Tasks
  - Any number of tasks can write to the same queue
  - Any number of tasks can read from the same queue



- Blocking on Queue Reads : Empty Queue
  - Specify block time or Time out (optional)
  - More than one task blocked on waiting for data
  - Only one task will be unblocked when data becomes available
    - Highest priority task
    - Same priority : the longest blocked task

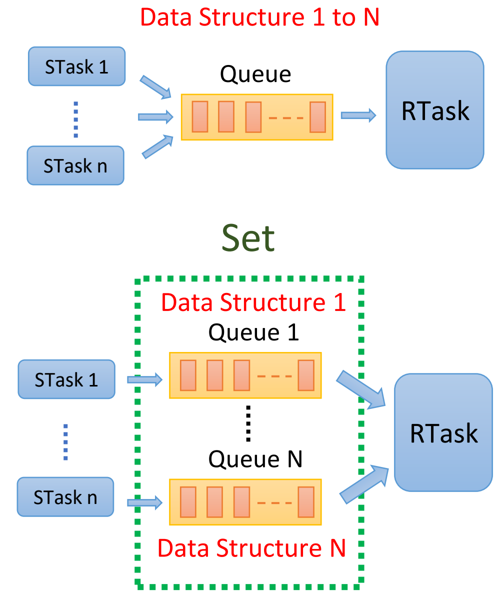


- Blocking on Queue Writes : Full Queue
  - Specify block time or Time out (optional)
  - More than one task blocked on it waiting to complete a send operation
  - Only one task will be unblocked when space on the queue becomes available
    - Highest priority task
    - Same priority : the longest blocked task



Blocking on multiple Queues (1)

- 2 solutions
  - (1) Using a single queue that receives structures
  - (2) Using separate queues for some data sources
- Second solution
  - Set configUSE_QUEUE_SETS = 1
  - Creating a queue set
  - Adding queues to the set*
  - Reading from the queue set to determine which queues within the set contain data




```c
QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;
QueueSetHandle_t xQueueSet;

int main(void) {
   xQueue1 = xQueueCreate(1, sizeof(char *));
   xQueue2 = xQueueCreate(1, sizeof(uint32_t));

   /* Create the queue setwith 2 events */
   xQueueSet = xQueueCreateSet(2);

   /* Add the two queues to the set. */
   xQueueAddToSet(xQueue1, xQueueSet);
   xQueueAddToSet(xQueue2, xQueueSet);
   ...
}
```

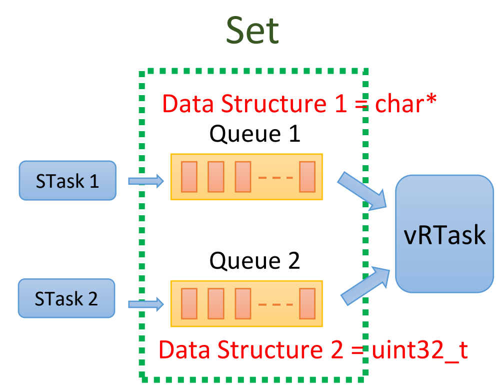
Example with N = 2

* Semaphores can also be added to a queue set

Blocking on Multiple Queues (2)

```
void vRTask(void *pvParameters)
{
   QueueSetMemberHandle_t xHandle;
   QueueHandle_t xQueueThatContainsData;
   char *pcReceivedString;
   uint32_t ulRecievedValue;

   for (;; ) {
      /* Block on the queue set for a maximum of 100ms */
      xHandle = xQueueSelectFromSet(xQueueSet, pdMS_TO_TICKS(100));

      if (xHandle == NULL) {
         /* The call to xQueueSelectFromSet() timed out. */
         ...
      }
      else if (xHandle == (QueueSetMemberHandle_t)xQueue1) {
         xQueueReceive(xQueue1, &pcReceivedString, 0);
         ...
      }
      else if (xHandle == (QueueSetMemberHandle_t)xQueue2) {
         xQueueReceive(xQueue2, &ulRecievedValue, 0);
         ...
      }
   }
}
```

Example with N = 2



Mailbox

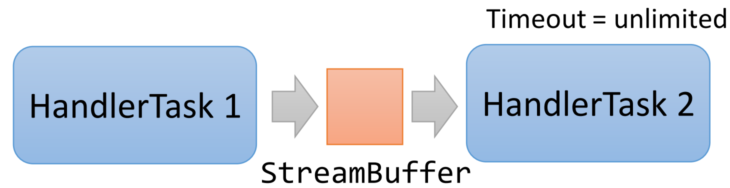
- Queue : length = 1
- A mailbox is used to hold data that can be read by any task, or any interrupt service routine
- Behavior
  - Remains in the mailbox until it is overwritten by sender
  - The receiver just reads the value from the mailbox
  - The receiver does not remove the value from the mailbox
- Using xQueueOverwrite() et xQueuePeek()

Example of 2 functions : update, read

```
typedef struct xMyDataStruct_t {
   TickType_t xTimeStamp;
   uint32_t ulValue;
} xMyData_t;

void vUpdateMailbox(uint32_t ulNewValue) {
   xMyData_t xData;

   /* Write the new data into the Example_t structure.*/
   xData.ulValue = ulNewValue;

   /* Use the RTOS tick count as the timestamp */
   xData.xTimeStamp = xTaskGetTickCount();

   /* Send the structure to the mailbox, overwriting any data */
   xQueueOverwrite(xMailbox, &xData);
}
```

```
BaseType_t vReadMailbox(xMyData_t *pxData) {
   TickType_t xPreviousTimeStamp;
   BaseType_t xDataUpdated;

   xPreviousTimeStamp = pxData->xTimeStamp;

   /* Update the xMyData_t structure pointed to by pxData. */
   xQueuePeek(xMailbox, pxData, portMAX_DELAY);

   /* Return pdTRUE if the value read from the mailbox
   has been updated since this function was last called. */
   return (pxData->xTimeStamp > xPreviousTimeStamp?pdTRUE:pdFALSE);
}
```

Stream & Message Buffers



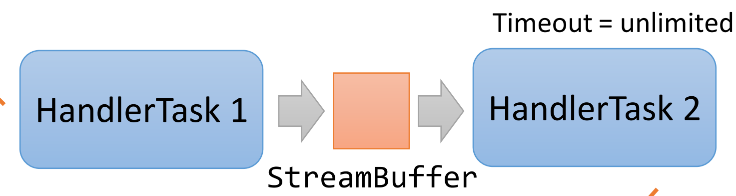
- From FreeRTOS 10.0
- Optimized for single reader single writer scenarios
  - passing data from an interrupt service routine to a task
  - from one microcontroller core to another on dual core CPUs
- Data is passed by copy
- Stream buffers pass a continuous stream of bytes
- Message buffers pass variable sized but discrete messages

```c
StreamBufferHandle_t xStreamBuffer;
int main(void) {
  const size_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;
  xStreamBuffer = xStreamBufferCreate(xStreamBufferSizeBytes, xTriggerLevel);

  if (xStreamBuffer == NULL) {
    // Error: Not enough heap memory space available
    ...
  }
  else {
    ...
  }
```

Task will be unblocked when a *xTriggerLevel* byte is written to the buffer

Stream buffer Example

```
void vHandlerTask1(void) {
  size_t xBytesSent;
  char *pcStringToSend = "My message to send";
  for (;; ) {
    vTaskDelay(20);
    xBytesSent = xStreamBufferSend(xStreamBuffer,
                                   (void *)pcStringToSend,
                                   strlen(pcStringToSend));

    if (xBytesSent != strlen(pcStringToSend)) {
      /* There was not enough free space in the stream buffer
      for the entire string to be written */
    }
  }
}
```




```
void vHandlerTask2(StreamBuffer_t xStreamBuffer) {
  uint8_t ucRxData[20];
  size_t xReceivedBytes;

  xReceivedBytes = xStreamBufferReceive(xStreamBuffer,
                                        (void *)ucRxData,
                                        sizeof(ucRxData), portMAX_DELAY);

  if (xReceivedBytes > 0) {
    ...
  }
}
```

Resource management

Introduction

- **Shared/guarded resource**
- Critical section
  - Protection of a region of code from access by other tasks and by interrupts
- Binary semaphore
  - Used for synchronization : tasks/tasks or tasks/interrupts
  - Task notification is also a good alternative for synchronization
- Counting semaphore
  - Used for counting events or resource management
- Mutual exclusion (Mutex)
  - Binary semaphore
  - Included a priority inheritance mechanism
  - Can be a Recursive Mutex

Critical section / region

- Code segment executed as an atomic action
  - No preemption, surrounded by P()/V() operations
  - Only interrupts may still execute whose logical priority is above the value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY
- Execution of the critical section must be as short as possible
- Primitives
  - taskENTER_CRITICAL(), taskENTER_CRITICAL_FROM_ISR()
  - taskEXIT_CRITICAL(), taskEXIT_CRITICAL_FROM_ISR()

Task

Critical section

```
void vPrintString(const char *pcString) {
  taskENTER_CRITICAL();
  {
    DISPLAY("%s", pcString);
    fflush(stdout);
  }
  taskEXIT_CRITICAL();
}
```

Interrupt

Critical section

```
void vAnInterruptServiceRoutine(void) {
  UBaseType_t uxSavedIsrStatus;
  ...
  uxSavedIsrStatus = taskENTER_CRITICAL_FROM_ISR();
  {
    ...
  }
  taskEXIT_CRITICAL_FROM_ISR(uxSavedIsrStatus);
  ...
}
```

Critical section / region
Suspended Scheduler

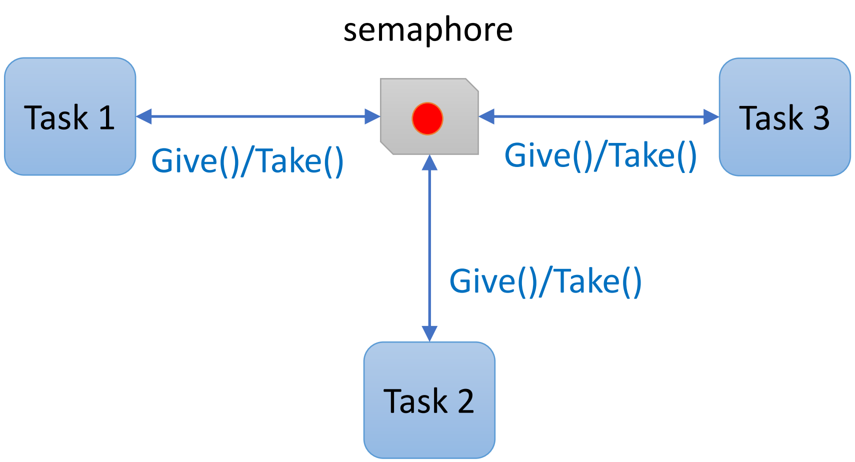
- Suspending/locking the scheduler
- No preemption but interrupts enabled
  - If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending, and is performed only when the scheduler is resumed.
- FreeRTOS API functions must not be called while the scheduler is suspended
- Primitives
  - vTaskSuspendScheduler()
  - xTaskResumeScheduler()

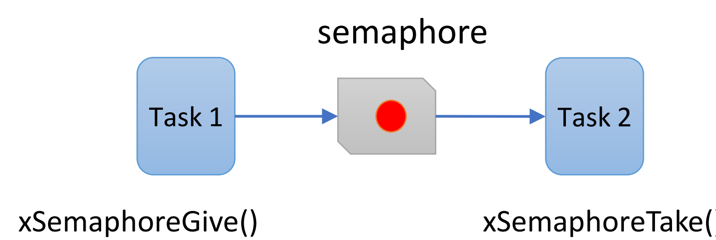
Critical section

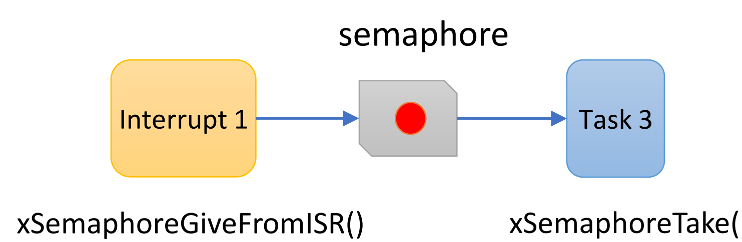
```
void vPrintString(const char *pcString) {
  vTaskSuspendScheduler();
  {
    DISPLAY("%s", pcString);
    fflush(stdout);
  }
  xTaskResumeScheduler();
}
```

Binary semaphore

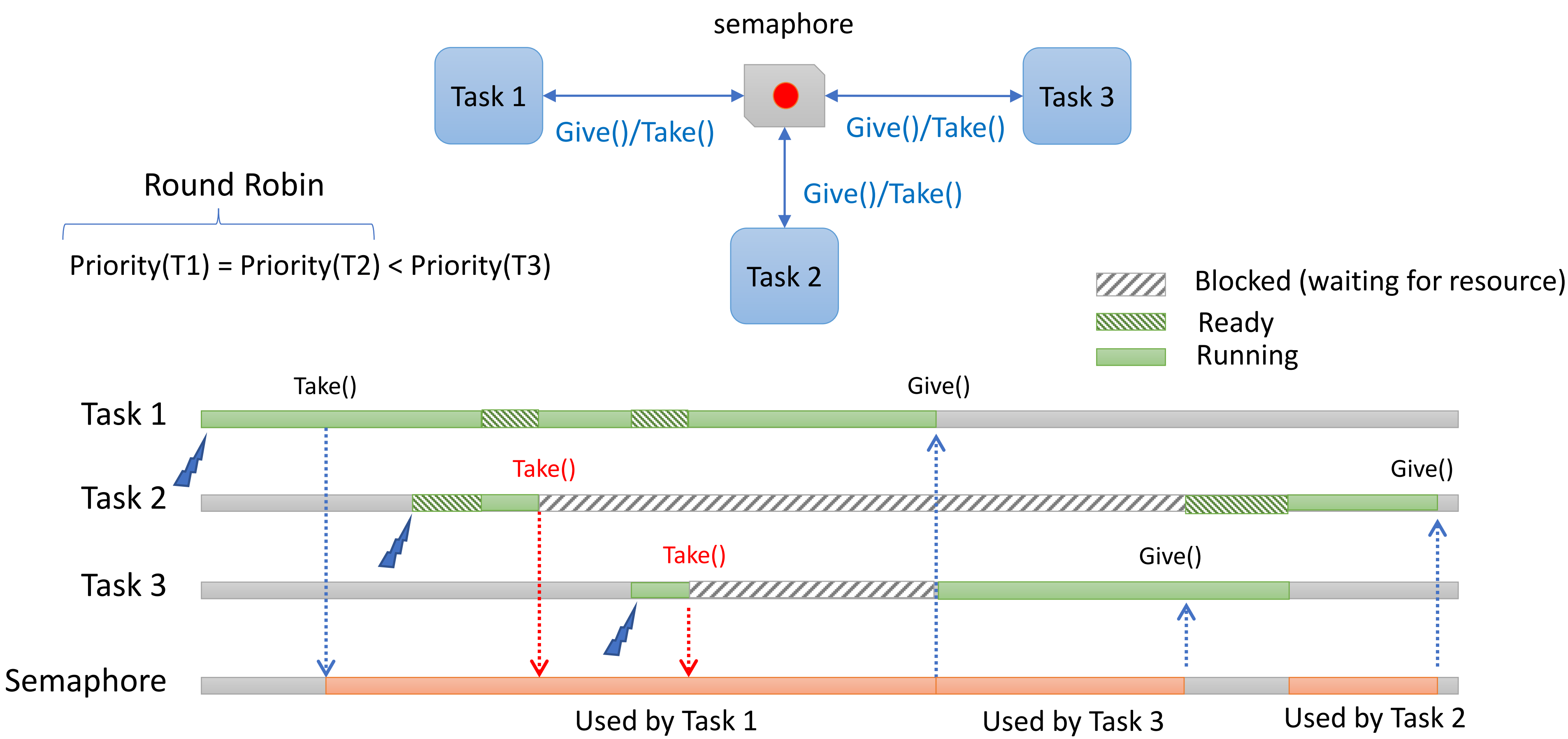
- Queue with one item (called token)
- Full / Empty queue = binary
- Highest priority task will be unblocked when the semaphore becomes available





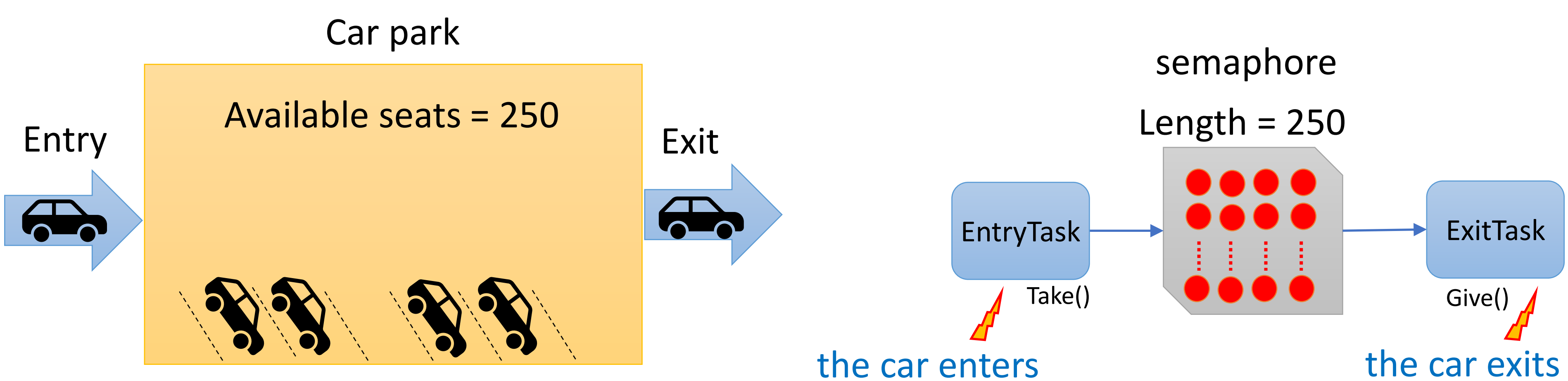


Binary semaphore example



Counter semaphore

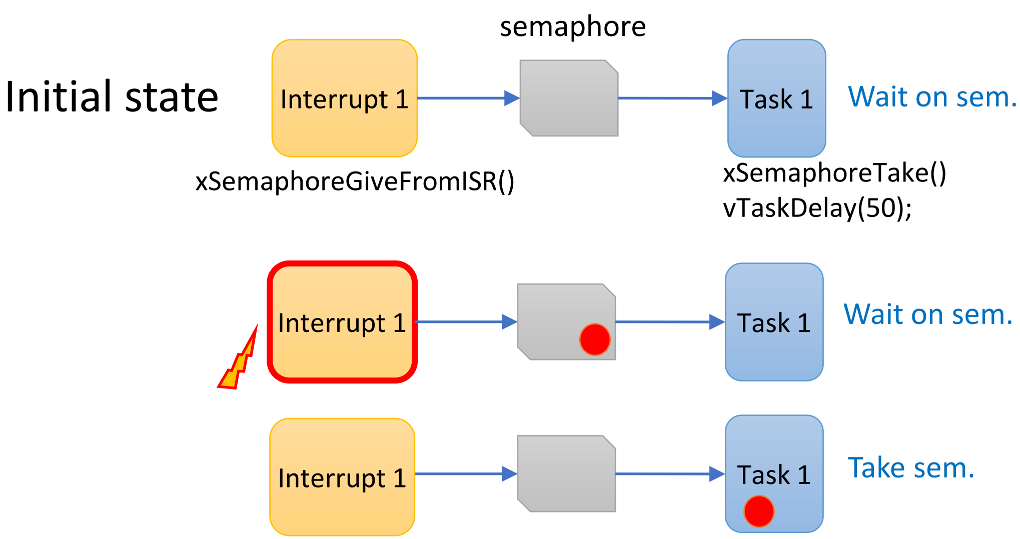
- Queue with length of more than one item (token)
- Count the number of items in the queue
- Set configUSE_COUNTING_SEMAPHORES
- Example: Resource management
  - Count value indicates the number of resources available

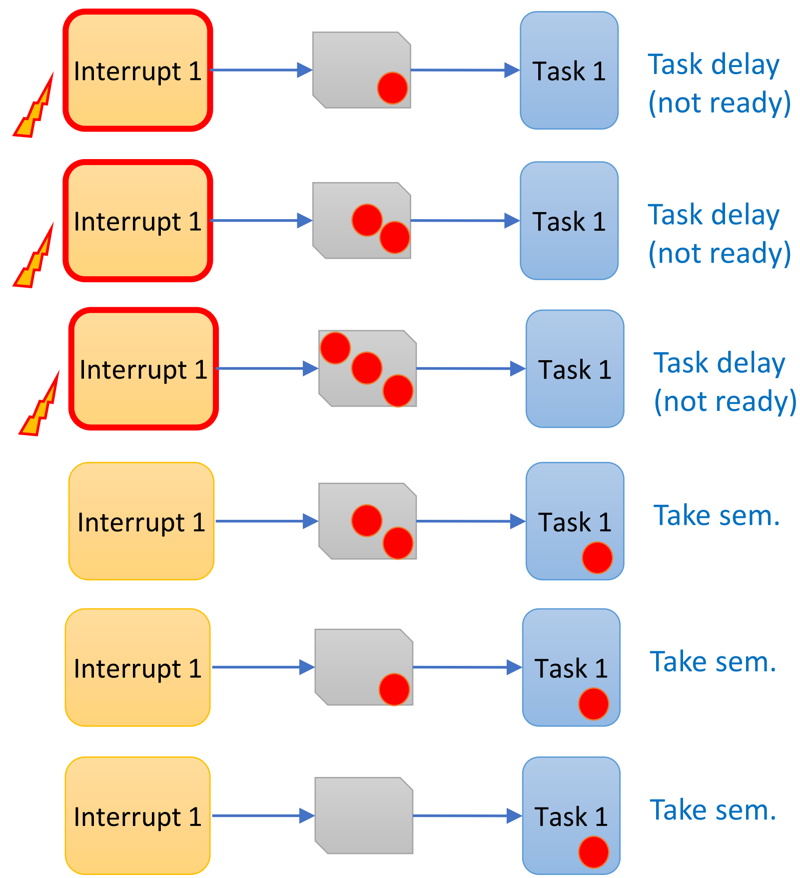


Counter semaphore

Counting event example

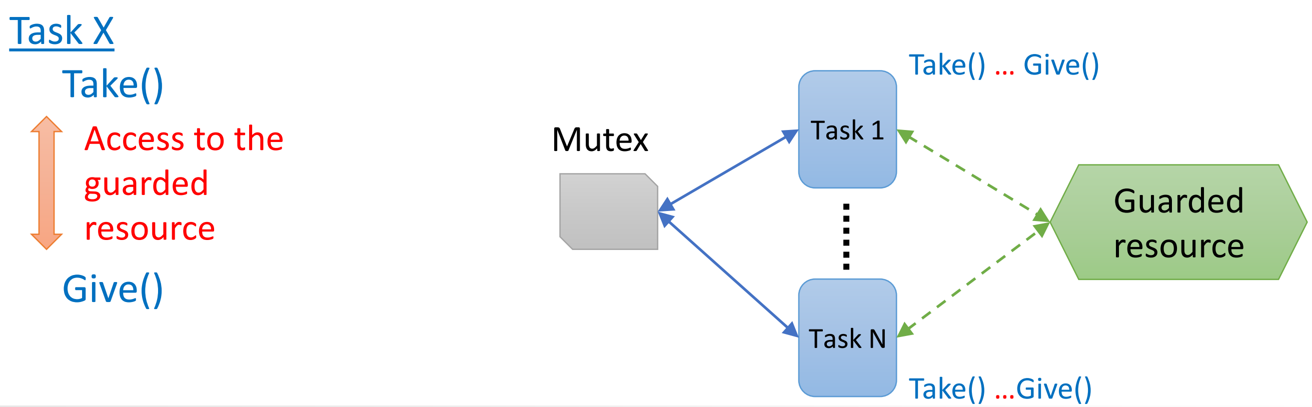
- Count value indicates the number of events that have occurred but have not been processed
- Will allow events to be processed by the task even if it is not ready



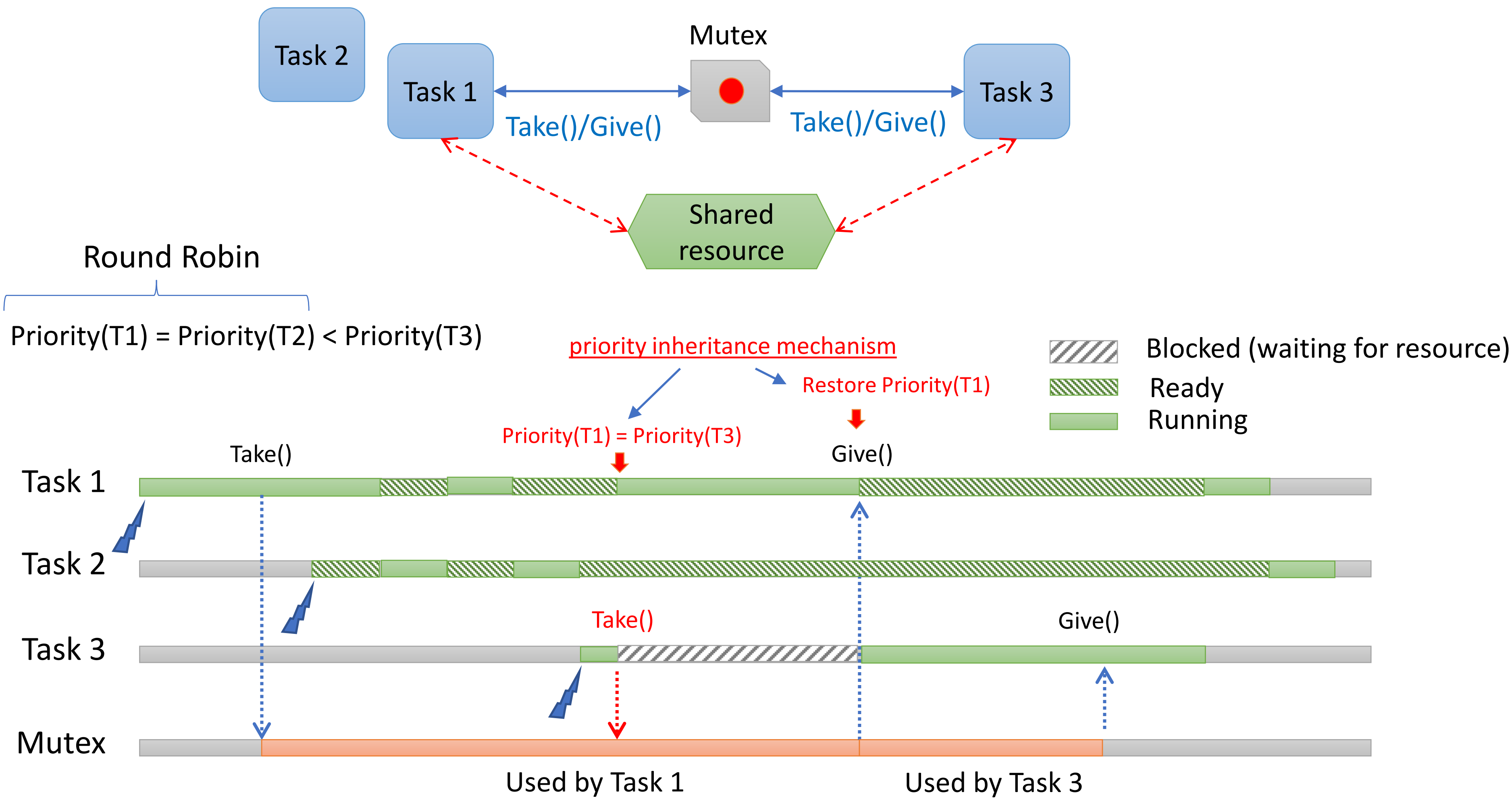


Mutex – Mutual Exclusion

- Used to control access to a resource shared between tasks
- A task should never (or the least possible!) get blocked by a lower priority task
- Included a priority inheritance mechanism
- Set configUSE_MUTEXES = 1



Mutex example



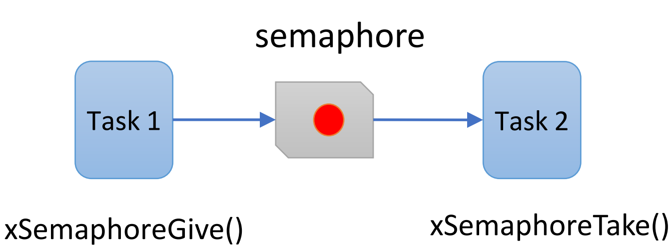
Recursive Mutex

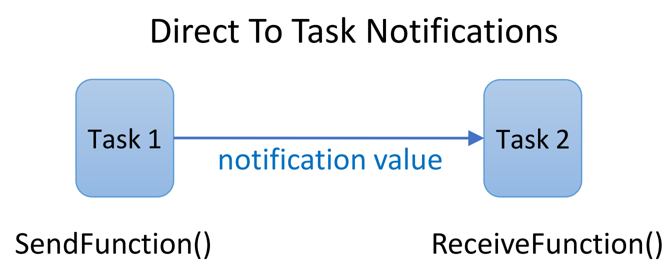
- Possible for a task to deadlock with itself
- Attempts to take the same mutex more than once
- Scenario
  - Task 1 successfully obtains a mutex
  - While holding the mutex, the task 1 calls a library function
  - Library function attempts to take the same mutex
  - The task 1 is in blocked state ! (deadlock)
- Avoided by using a recursive mutex
  - Can “take” more than one by the same task
  - Just call once “give”

Direct To Task Notifications

Introduction

- Tasks communicate through intermediary objects
  - Queues, Semaphore …
  - Data are not sent directly to a receiving task/ISR
- Another solution: Using Direct To Task Notifications
- Advantages
  - Faster than using a queue, semaphore or event group
  - RAM Footprint Benefits : less RAM than using a queue, semaphore or event group
- Set configUSE_TASK_NOTIFICATIONS = 1





Limitations

- Task notification cannot be used
  - Sending an event or data to an ISR
  - Enabling more than one receiving task
  - Buffering multiple data items
    - Task notifications send data to a task by updating the receiving task's notification value.
  - Broadcasting to more than one task
    - Task notifications are sent directly to the receiving task
  - Waiting in the blocked state for a send to complete
    - If a task attempts to send a task notification to a task that already has a notification pending

First Example
processing all at once

```
TaskHandle_t xHandlerTask = NULL;

int main(void) {
  xTaskCreate(vHandlerTask, "Handler", 1000, NULL, 3, &xHandlerTask);
  vPortSetInterruptHandler(3, ulInterruptHandler);
  vTaskStartScheduler();
  for (;; );
  return 0;
}

void vHandlerTask1(void *pvParameters) {
  uint32_t ulEventsToProcess;
  for (;; ) {
    ulEventsToProcess = ulTaskNotifyTake(pdTRUE, pdMS_TO_TICKS(500));
    if (ulEventsToProcess != 0) {
      while (ulEventsToProcess > 0) {
        DISPLAY("Handler task - Processing event.");
        ...
        ulEventsToProcess--;
      }
    }
    else {
      ...
    }
  }
}
```

Timeout = 500 ms

InterruptHandler → HandlerTask 1

notification value = 0

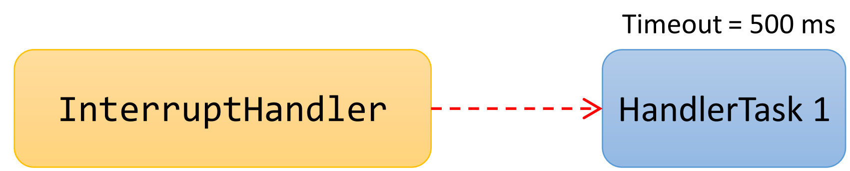
Time out

All at once

```
uint32_t ulInterruptHandler(void) {
  BaseType_t xHigherPriorityTaskWoken;
  vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Second Example
Processing one by one

```c
TaskHandle_t xHandlerTask = NULL;

int main(void) {
  xTaskCreate(vHandlerTask, "Handler", 1000, NULL, 3, &xHandlerTask);
  vPortSetInterruptHandler(3, ulInterruptHandler);
  vTaskStartScheduler();
  for (;; );
  return 0;
}
```



One by one

```c
void vHandlerTask1(void *pvParameters) {
  for (;; ) {
```

notification value = notification value - 1

```c
    if (ulTaskNotifyTake(pdFALSE, pdMS_TO_TICKS(500)) != 0) {
      DISPLAY("Handler task - Processing event.");
      ...
    }
    else {
      ...
    }
  }
}
```

```c
uint32_t ulInterruptHandler(void) {
  BaseType_t xHigherPriorityTaskWoken;
  vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

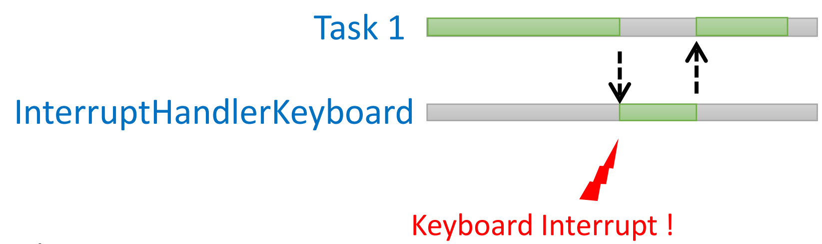
Advanced functions

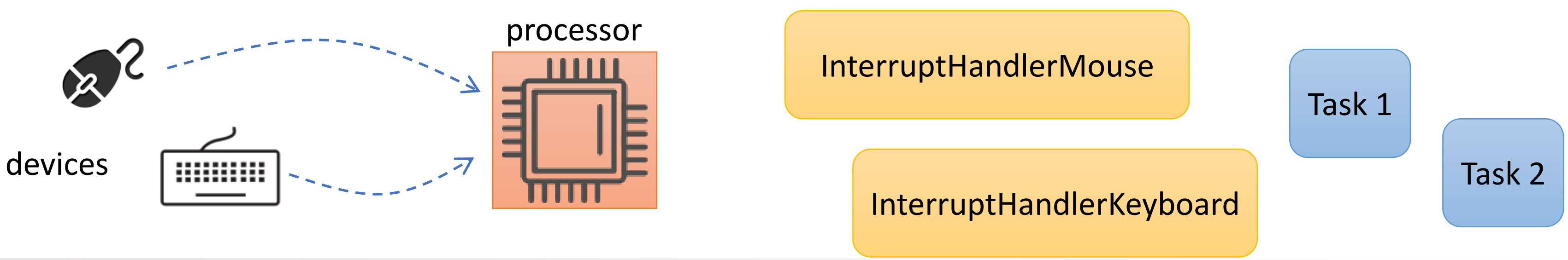
- xTaskNotify(), xTaskNotifyFromISR()
  - More flexible and powerful than xTaskNotifyGive()
  - Can be used to update the receiving task's notification value
    - Increment
    - Set one or more bits in the receiving task's notification value
    - Write a completely new number into the receiving task's notification value
- xTaskNotifyWait()
  - More powerful than ulTaskNotifyTake()
  - Allows a task to wait, with an optional timeout
  - To be cleared in the calling task's notification value
    - entry to the function
    - on exit from the function

Interrupt Management

What is an interrupt ?

- Signal sent from a device/program
- Request for the processor to interrupt the current program execution
- Associated with an interrupt handler
- Hardware interrupt - Interrupt ReQuest (IRQ)
  - IRQ is an electronic signal issued by an external hardware device
  - GPIO, Timer, UART, USB, Mouse, keyboard …
- Software interrupt
  - Requested by the processor itself
    - executing particular instructions
    - when certain conditions are met
    - triggered by program execution errors, called traps or exceptions
- Interrupt can be disabled or maskable, some are non-maskable interrupts (NMI)





Interrupt & task

- Distinction between the priority of a task & an interrupt
  - Tasks will only run when there are no ISRs running
  - The lowest priority interrupt will interrupt the highest priority task
  - No way for a task to pre-empt an ISR
- Interrupt Service Routine (ISR) API
  - One version for use from tasks
  - One version for use from ISRs with <u>no blocked state</u>
  - Never call a FreeRTOS API function that does not have "FromISR" in its name from an ISR
  - Allows task/ISR code to be more efficient

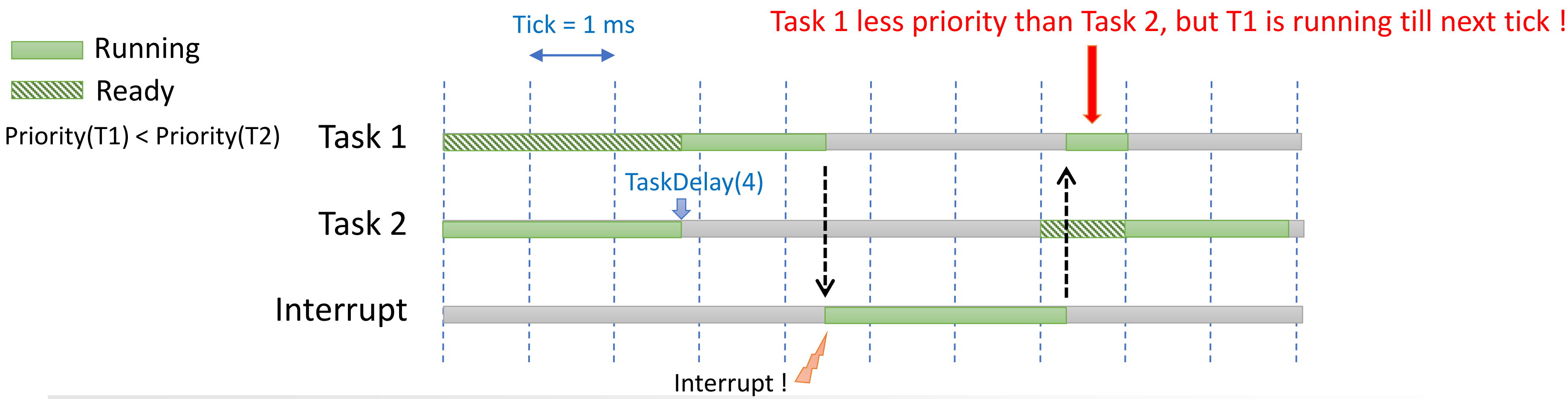
Context Switch - Problematic

- The task running when the interrupt exits might be different to the task that was running when the interrupt was entered

```
void vTask1(void *pvParameters) {
  for (;;) {
    DISPLAY("Task 1 running ...");
  }
}
```

```
void vTask2(void *pvParameters) {
  for (;;) {
    vTaskDelay(4);
    DISPLAY("Task 2 running ...");
  }
}
```

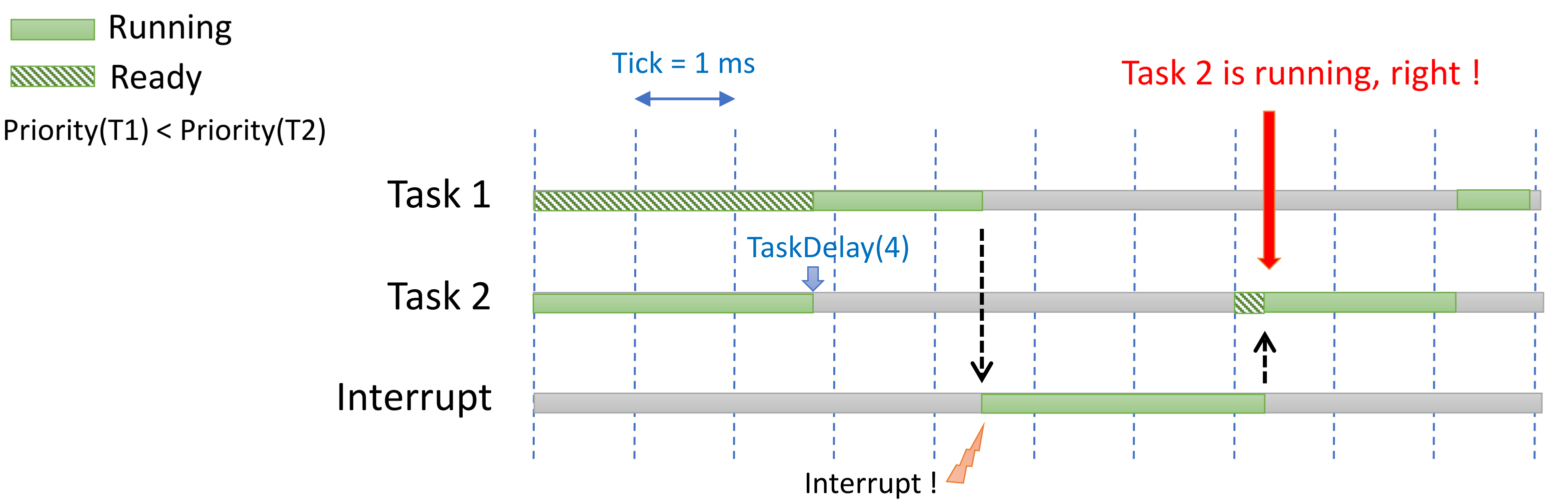
```
uint32_t ulInterruptHandler(void) {
  DISPLAY("Interrupt wake up !");
}
```



Context Switch - Solution

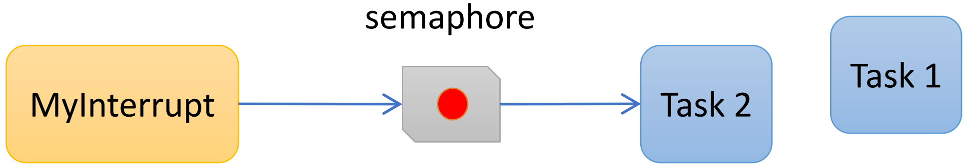
- Called a API function to request a context switch if necessary
- portYIELD_FROM_ISR(pxHigherPriorityTaskWoken)
  - pxHigherPriorityTaskWoken = true : could have a context switch
  - pxHigherPriorityTaskWoken = true : do nothing

```
uint32_t ulInterruptHandler(void) {

  DISPLAY("Interrupt wake up !");

  portYIELD_FROM_ISR(pdTRUE);
}
```



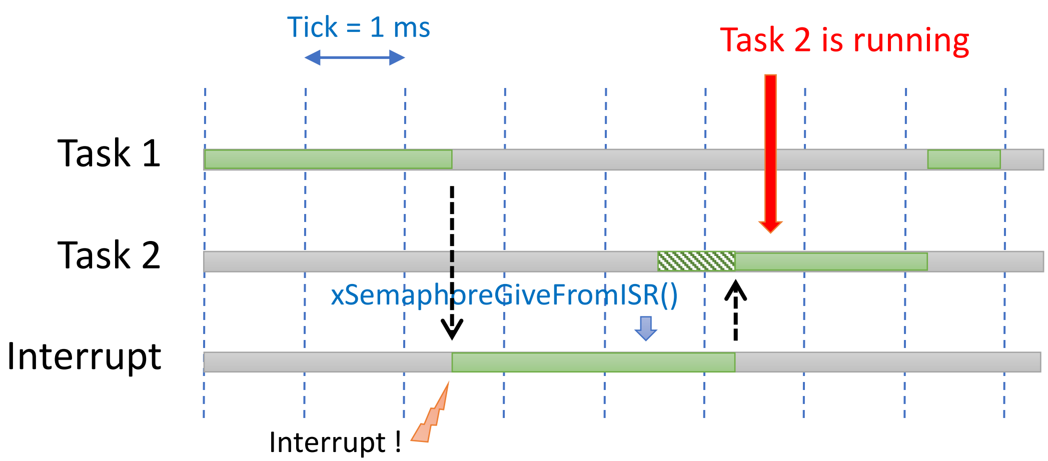
Example with semaphore

Priority(T1) < Priority(T2)




```c
uint32_t ulInterruptHandler(void) {
  BaseType_t xHigherPriorityTaskWoken= pdFALSE;

  xSemaphoreGiveFromISR(xBinarySemaphore,
                        &xHigherPriorityTaskWoken);

  DISPLAY("Interrupt wake up !");
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
void vTask2(void *pvParameters) {
  for (;;) {
    xSemaphoreTake(xCountingSemaphore,
                   portMAX_DELAY);
    DISPLAY("Task 2 running ...");
  }
}
```



Using an interrupt on Windows port

```
#define mainINTERRUPT_NUMBER 3
```

Numbers 0 to 2 are used by the FreeRTOS Windows port itself
3 is the first number available to the application.

```
int main(void) {
  vPortSetInterruptHandler(mainINTERRUPT_NUMBER, ulInterruptHandler);
  ...
}

uint32_t ulInterruptHandler(void) {
  BaseType_t xHigherPriorityTaskWoken= pdFALSE;
  ...
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Somewhere else (in a task)

```
vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );
```