FreeRTOS on STM32

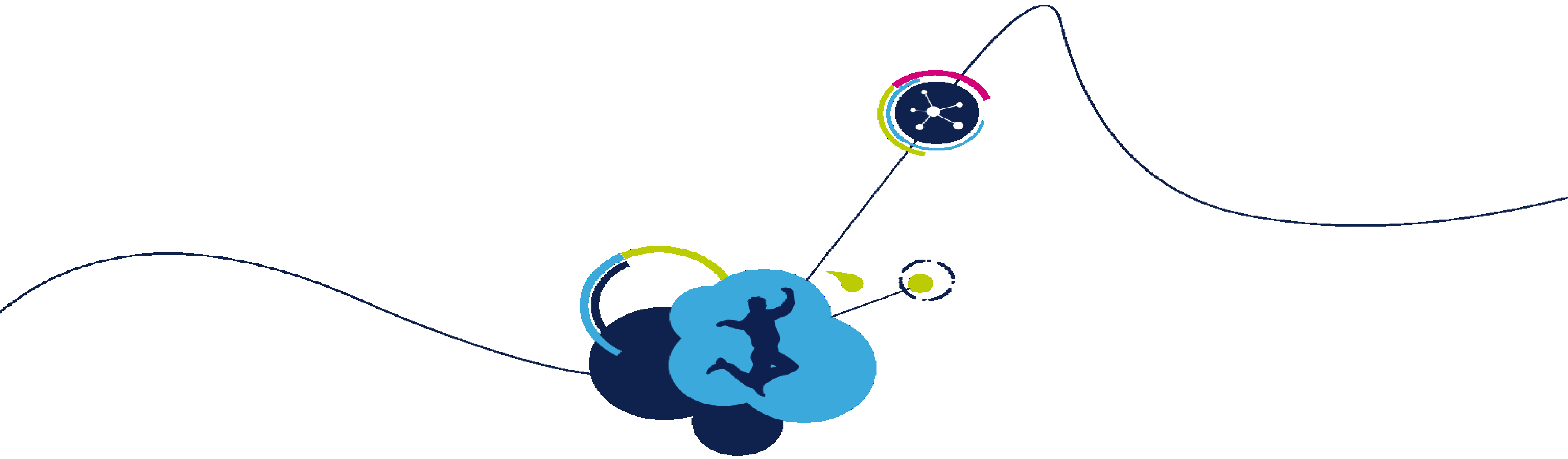
CMSIS_OS API

T.O.M.A.S – Technically Oriented Microcontroller Application Services
v1.7

life.augmented

Agenda

- FreeRTOS
  - Operating system: what is … ?
  - Basic features
  - CMSIS_OS API vs FreeRTOS API
  - FreeRTOS and STM32CubeMX
  - Configuration
  - Memory allocation
  - Scheduler
  - Tasks
  - Intertask communication
    - Queues (messages, mail)
    - Semaphores (binary, counting)
    - Signals
  - Resources management
  - Mutexes
  - Software Timers
  - Advanced topics (hooks, stack overflow protection, gatekeeper task)
  - Debugging
  - Low power support (tickless modes)
  - Footprint



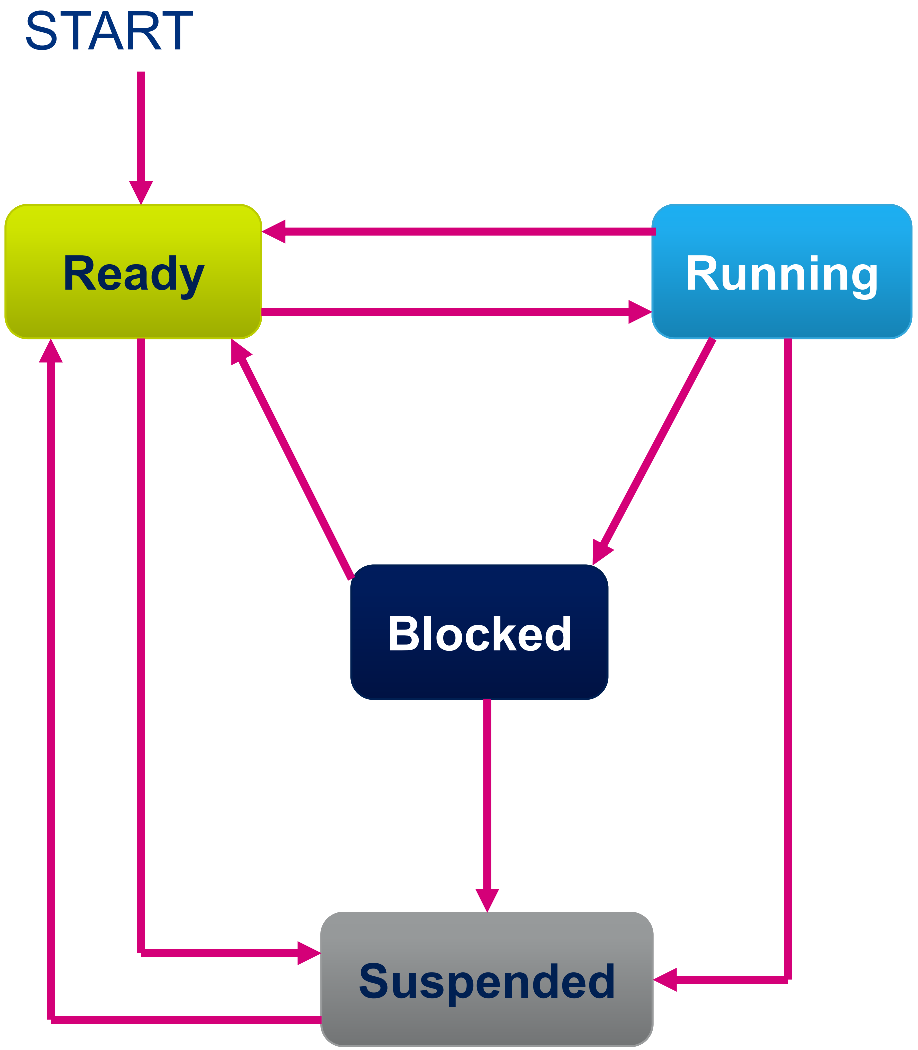
Operating System what is … ?

What is Task?

- It is C function:
- It should be run within infinite loop, like:

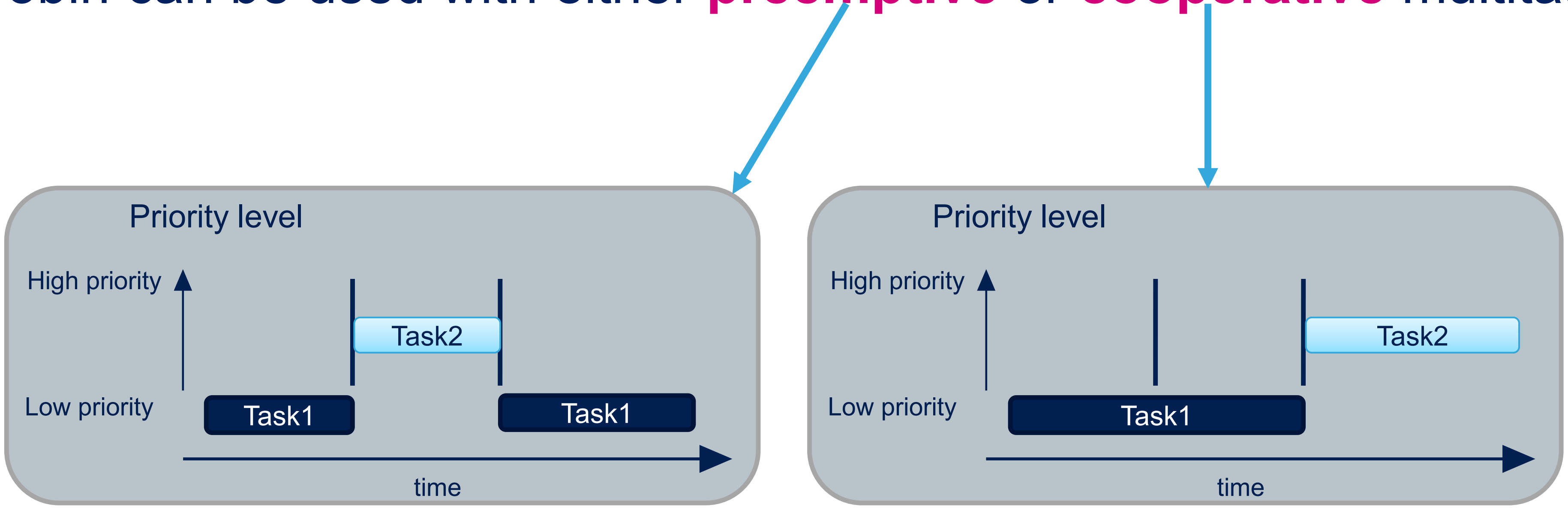
```
for(;;)
{
  /* Task code */
}
```

- It has its own part of stack, and priority
- It can be in one of 4 states (RUNNING, BLOCKED, SUSPENDED, READY)
- It is created and deleted by calling API functions

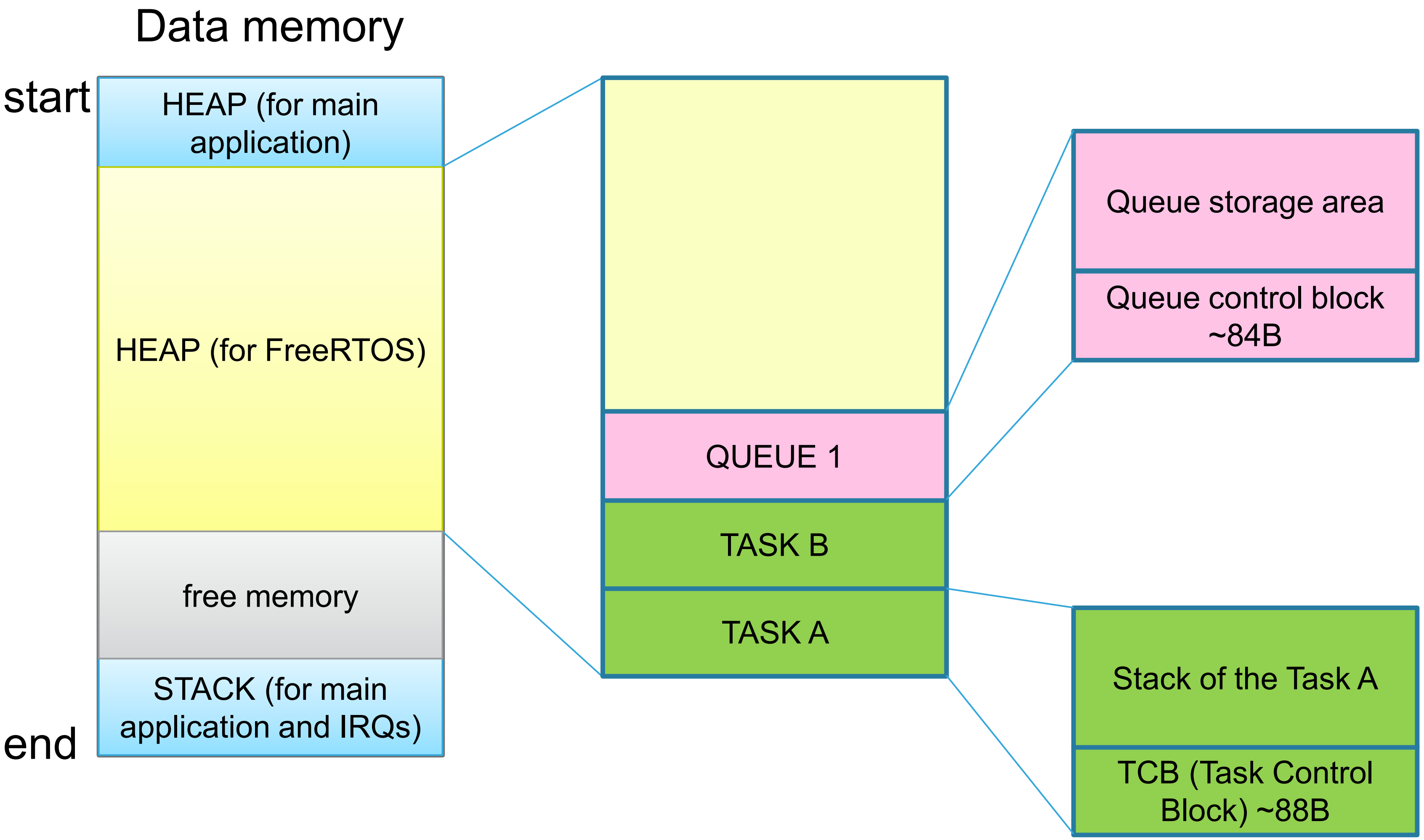


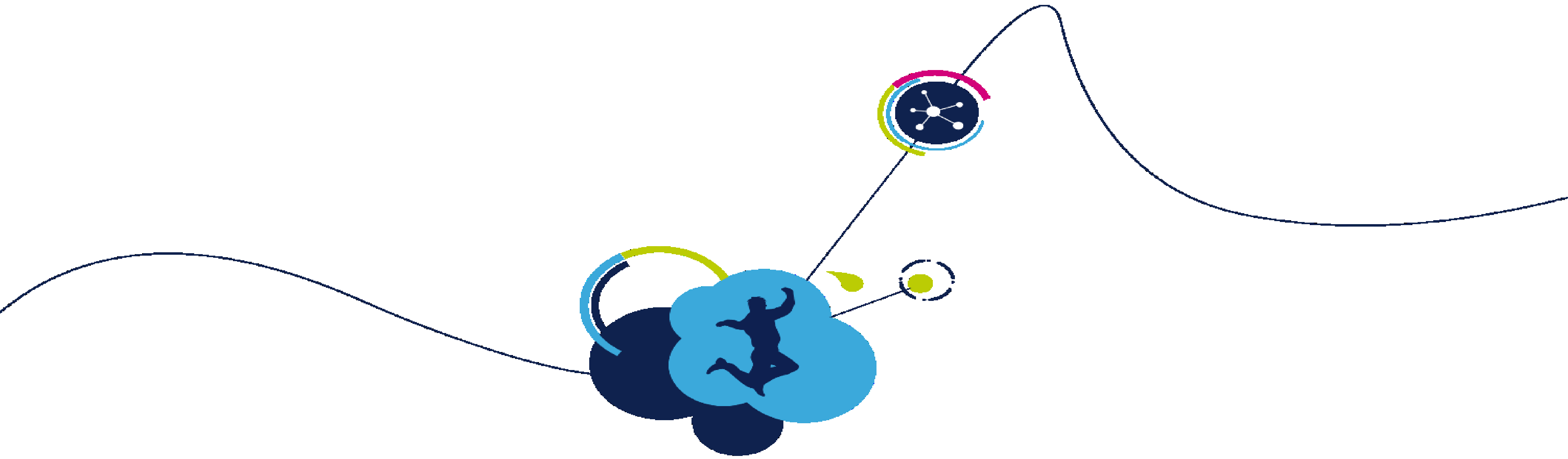
What is scheduler?

- The **scheduler** is an algorithm determining which task to execute.
  - Is select one of the task being ready to be executed (in READY state)
  - There are few mechanisms controlling access to CPU for tasks (timeslice, preemption, idle)
- In FreeRTOS **round-robin** scheduling algorithm is implemented
- Round-robin can be used with either **preemptive** or **cooperative** multitasking



What is OS heap?





FreeRTOS basic features

About FreeRTOS (1/2)

- Market leading RTOS by Real Time Engineers Ltd.
- Professionally developed with strict quality management
- Commercial versions available: OpenRTOS and SafeRTOS
- Documentation available on www.freertos.org
- Free support through forum (moderated by RTOS author Richard Barry)

About FreeRTOS (2/2)

- FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license without any requirement to expose your proprietary source code. An alternative commercial license option is also available.
- FreeRTOS license details available on :

  http://www.freertos.org/a00114.html
- In the STM32Cube firmware solution FreeRTOS is used as real time operating system through the generic CMSIS-OS wrapping layer provided by ARM. Examples and applications using the FreeRTOS can be directly ported on any other RTOS without modifying the high level APIs, only the CMSIS-OS wrapper has to be changed in this case.

FreeRTOS - Main features

- Preemptive or cooperative real-time kernel
- Tiny memory footprint (less than 10kB ROM) and easy scalable
- Includes a tickless mode for low power applications
- Synchronization and inter-task communication using
  - message queues
  - binary and counting semaphores
  - mutexes
  - group events (flags)
- Software timers for tasks scheduling
- Execution trace functionality
- CMSIS-RTOS API port

FreeRTOS - resources used

Core resources:

- System timer (SysTick) – generate system time (time slice)
- Two stack pointers: **MSP**, **PSP**

Interrupt vectors:

- **SVC** – system service call (like SWI in ARM7)
- **PendSV** – pended system call (switching context)
- **SysTick** – System Timer

Flash memory:

- **6-10kB**

RAM memory:

- **~0.5kB** + task stacks:

System Service Call (SVC)

- **SVC** – system service call / supervisor call
- It is an instruction and an exception. Once the svc instruction is executed, SVD IRQ is triggered immediately (unless there is higher priority IRQ active)
- SVC contains an 8bit immediate value what could help to determine which OS service is requested.
- Do not use SVC inside NMI or Hard Fault handler

Pended System Call (PendSV)

- **PendSV** is a priority programmable exception triggered by SW (write to the in ICSR register @0xE000ED04)

  ```
  SCB->ICSR |= (1<<28)
  ```

- It is not precise (in contrary to SVC). After set a pending bit CPU can execute a number of instructions before the exception will start. Usually it is used like a subroutine called i.e. by the system timer in OS

System timer

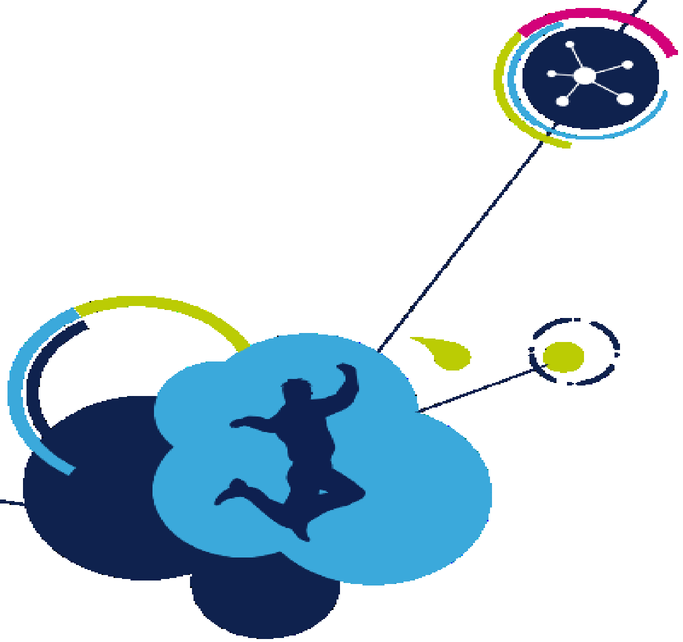
- It is necessary to trigger a context switching in regular time slots.
- In CortexM architecture 24bit downcounting SysTick is used for this purpose (it can be changed – more details in Tickless mode section)
- System timer **is triggering PendSV** SW interrupt to perform context switch.
- In case we are using HAL library it is strongly recommended to change its TimeBase timer from Systick to other timer available (i.e. TIM6)

FreeRTOS sources file structure

File / header Directory	role
croutine.c / croutine.h .\Source .\Source\include	Co-routines functions definitions. Efficient in 8 and 16bit architecture. In 32bit architecture usage of tasks is suggested
event_groups.c / event_groups.h .\Source .\Source\include	
heap_x.c .\Source\portable\MemMang	Memory management functions (allocate and free memory segment, three different approaches in heap_1, heap_2, heap_3 and heap_4 files)
list.c / list.h .\Source .\Source\include	List implementation used by the scheduler.
port.c / portmacro.h .\Source\portable\xxx\yyy	Low level functions supporting SysTick timer, context switch, interrupt management on low hw level – strongly depends on the platform (core and sw toolset). Mostly written in assembly. In portmacro.h file there are definitions of portTickType and portBASE_TYPE
queue.c / queue.h / semphr.h .\Source .\Source\include	Semaphores, mutexes functions definitions
tasks.c / task.h .\Source .\Source\include	Task functions and utilities definition
timers.c / timers.h .\Source .\Source\include	Software timers funcitons definitions
FreeRTOS.h .\Source\include	Configuration file which collect whole FreeRTOS sources
FreeRTOSConfig.h	Configuration of FreeRTOS system, system clock and irq parameters configuration

FreeRTOS sources file structure

File / header Directory	role
heap_x.c .\Source\portable\MemMang	Memory management functions (allocate and free memory segment, three different approaches in heap_1, heap_2, heap_3 and heap_4 files)
list.c / list.h .\Source .\Source\include	List implementation used by the scheduler.
port.c / portmacro.h .\Source\portable\xxx\yyy	Low level functions supporting SysTick timer, context switch, interrupt management on low hw level – strongly depends on the platform (core and sw toolset). Mostly written in assembly. In portmacro.h file there are definitions of portTickType and portBASE_TYPE
queue.c / queue.h / semphr.h .\Source .\Source\include	Semaphores, mutexes functions definitions
tasks.c / task.h .\Source .\Source\include	Task functions and utilities definition
FreeRTOS.h .\Source\include	Configuration file which collect whole FreeRTOS sources
FreeRTOSConfig.h	Configuration of FreeRTOS system, system clock and irq parameters configuration



FreeRTOS native API

FreeRTOS API conventions

- Prefixes at variable names:

  c – char

  s – short

  l – long

  x – portBASE_TYPE defined in *portmacro.h* for each platform (in STM32 it is long)

  u – unsigned

  p - pointer

- Functions name structure (`vTaskPrioritySet()` is taken as example):

prefix	file name	function name
v	**Task**	**PrioritySet**

  v – void

  x – returns portBASE_TYPE

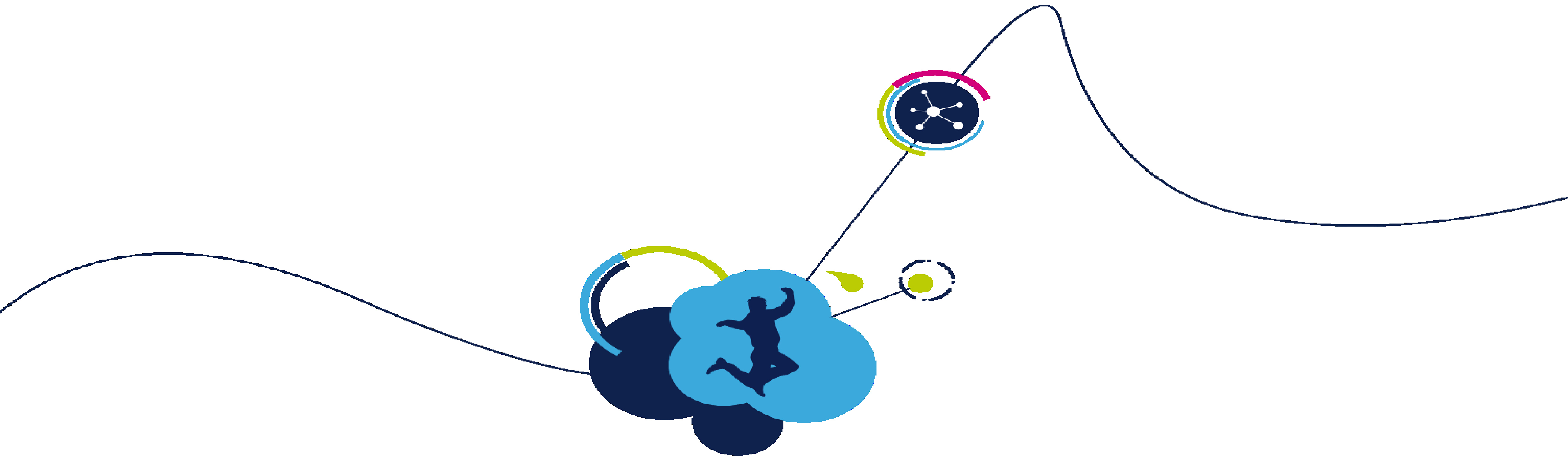
  prv – private

life.augmented

FreeRTOS API conventions - macros

- Prefixes at macros defines their definition location:
  - **port** – (ie. `portMAX_DELAY`) -> portable.h
  - **task** – (ie. `task_ENTER_CRITICAL`) -> task.h
  - **pd** – (ie. `pdTRUE`) -> projdefs.h
  - **config** – (ie. `configUSE_PREEMPTION`) -> FreeRTOSConfig.h
  - **err** – (ie. `errQUEUE_FULL`) -> projdefs.h

- Common macro definitions:
  - **`pdTRUE`** 1
  - **`pdFALSE`** 0
  - **`pdPASS`** 1
  - **`pdFAIL`** 0



FreeRTOS CMSIS_OS API

life.augmented

FreeRTOS
CMSIS-OS API

- CMSIS-OS API is a generic RTOS interface for Cortex-M processor based devices
- Middleware components using the CMSIS-OS API are RTOS independent, this allows an easy linking to any third-party RTOS
- The CMSIS-OS API defines a minimum feature set including
  - Thread Management
  - Kernel control
  - Semaphore management
  - Message queue and mail queue
  - Memory management
- The STM32Cube comes with an implementation of the CMSIS-RTOS for FreeRTOS.
- For detailed documentation regarding CMSIS-OS refer to:
  http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html

FreeRTOS

CMSIS-OS FreeRTOS implementation

- Implementation in file **cmsis-os.c** (found in folder: \Middlewares\Third_Party\FreeRTOS\Source\CMSIS_RTOS)
- The following table lists examples of the CMSIS-RTOS APIs and the FreeRTOS APIs used to implement them

API category	CMSIS_RTOS API	FreeRTOS API
Kernel control	`osKernelStart`	vTaskStartScheduler
Thread management	`osThreadCreate`	xTaskCreate
Semaphore	`osSemaphoreCreate`	vSemaphoreCreateBinary xSemaphoreCreateCounting
Mutex	`osMutexWait`	xSemaphoreTake
Message queue	`osMessagePut`	xQueueSend xQueueSendFromISR
Timer	`osTimerCreate`	xTimerCreate

- Note: CMSIS-OS implements same model as FreeRTOS for task states

- Most of the functions returns `osStatus` value, which allows to check whether the function is completed or there was some issue (**cmsis_os.h** file)

- Each OS component has its own ID:
  - Tasks: `osThreadId` (mapped to `TaskHandle_t` within FreeRTOS API)
  - Queues: `osMessageQId` (mapped to `QueueHandle_t` within FreeRTOS API)
  - Semaphores: `osSemaphoreId` (mapped to `SemaphoreHandle_t` within FreeRTOS API)
  - Mutexes: `osMutexId` (mapped to `SemaphoreHandle_t` within FreeRTOS API)
  - SW timers: `osTimerId` (mapped to `TimerHandle_t` within FreeRTOS API)

- Delays and timeouts are given in ms:
  - **0** – no delay
  - **>0** – delay in ms
  - **0xFFFFFFFF** – wait forever (defined in `osWaitForever` within **cmsis_os.h** file)

CMSIS_OS API

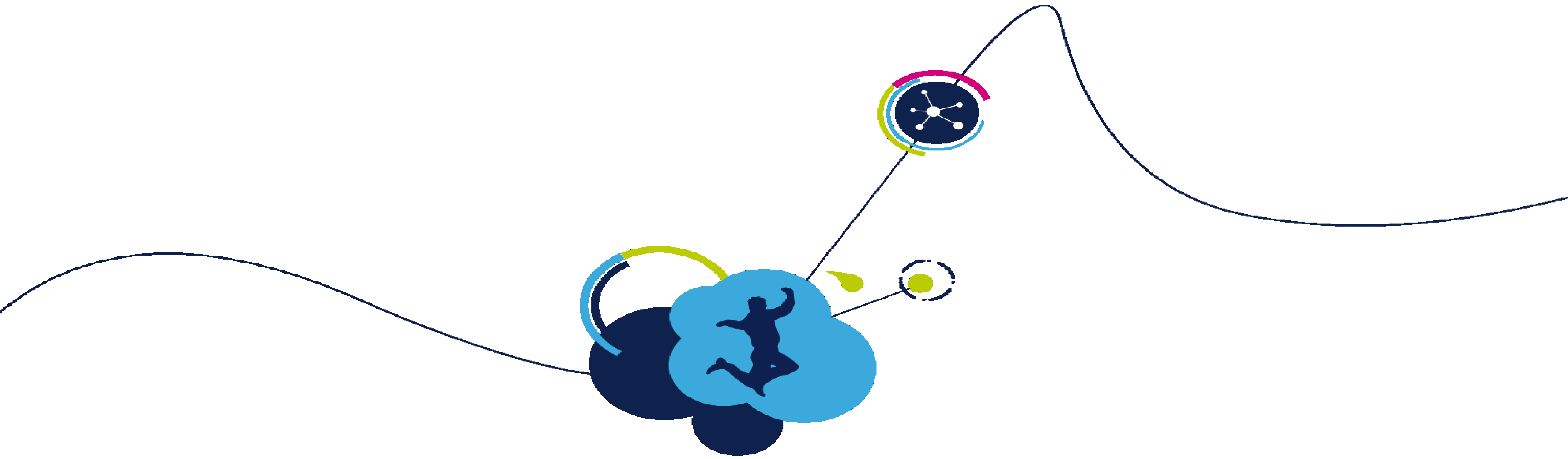
return values osStatus 1/2

- Most of the functions returns `osStatus` value, below you can find return values on function completed list (**cmsis_os.h** file)

osStatus	value	description
`osOK`	`0`	no error or event occurred
`osEventSignal`	`8`	signal event occurred
`osEventMessage`	`0x10`	message event occurred
`osEventMail`	`0x20`	mail event occurred
`osEventTimeout`	`0x40`	timeout occurred
`os_status_reserved`	`0x7FFFFFFF`	prevent from enum down-size compiler optimization

- Error status values `osStatus` (**cmsis_os.h**)

osStatus value	description
`osErrorParameter` 0x80	parameter error: a mandatory parameter was missing or specified an incorrect object.
`osErrorResource` 0x81	resource not available: a specified resource was not available
`osErrorTimeoutResource` 0xC1	resource not available within given time: a specified resource was not available within the timeout period.
`osErrorISR` 0x82	not allowed in ISR context: the function cannot be called from interrupt service routines
`osErrorISRRecursive` 0x83	function called multiple times from ISR with same object.
`osErrorPriority` 0x84	system cannot determine priority or thread has illegal priority
`osErrorNoMemory` 0x85	system is out of memory: it was impossible to allocate or reserve memory for the operation
`osErrorValue` 0x86	value of a parameter is out of range.
`osErrorOS` 0xFF	unspecified RTOS error: run-time error but no other error message fits.

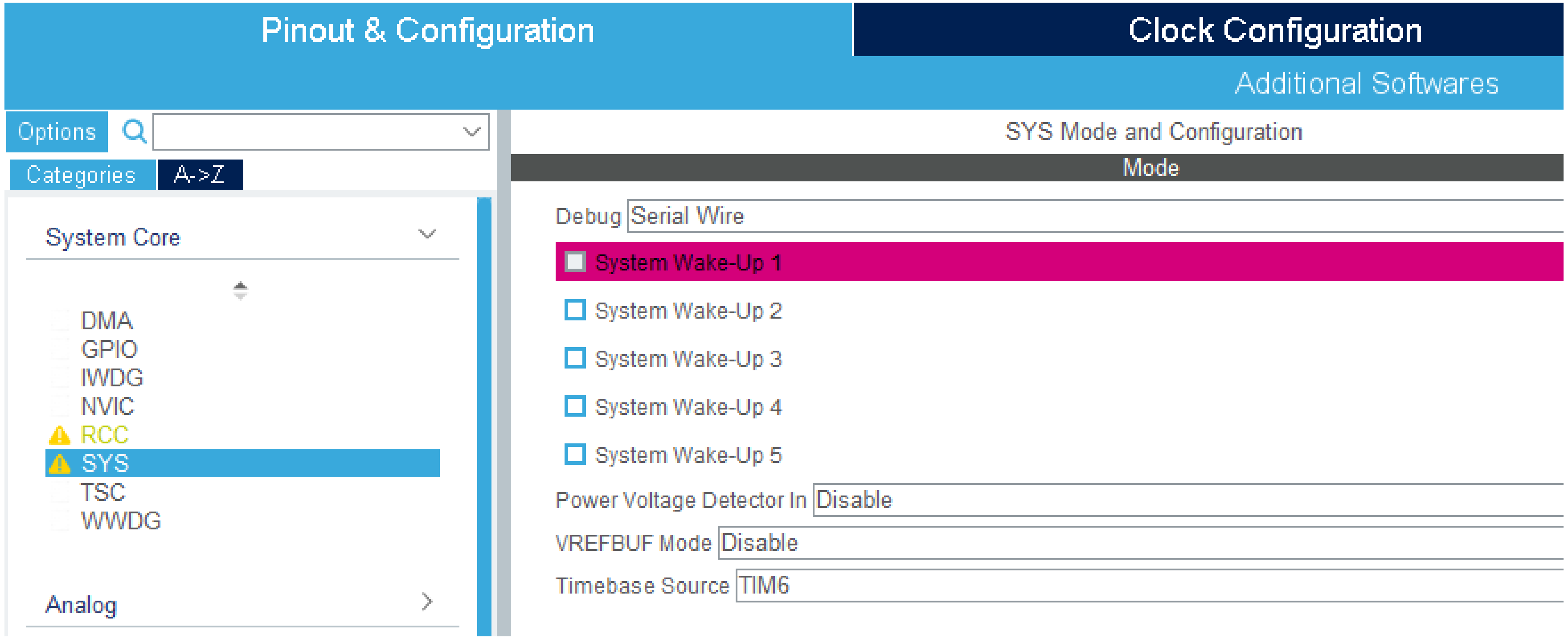


FreeRTOS and STM32CubeMX

FreeRTOS in STM32CubeMX

changing Timebase source for HAL

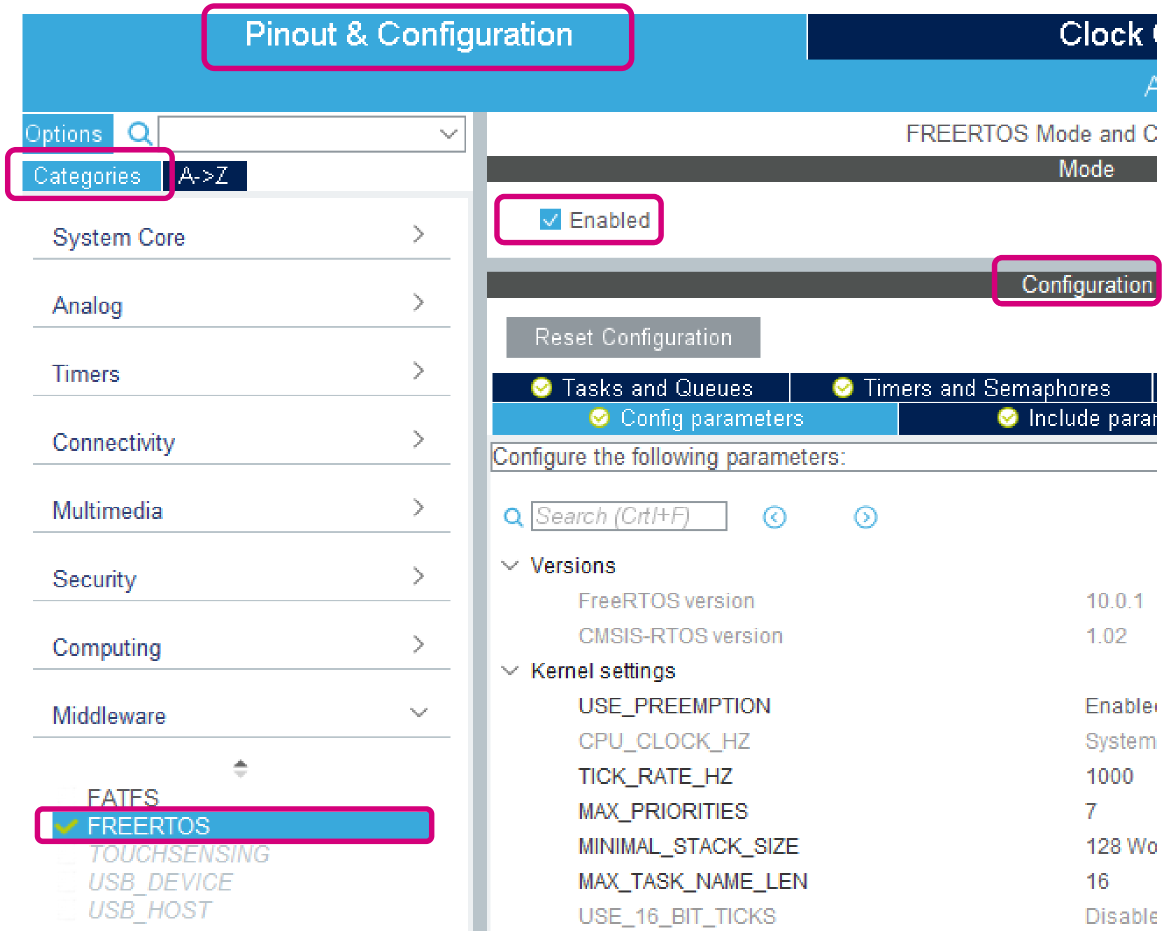
- Start a new project within STM32CubeMX for selected MCU (or open already prepared existing one – important is to have printf() implementation).
- Go to System Core section -> SYS and change Timebase source (for HAL) from SysTick to other timer i.e. TIM6



FreeRTOS in STM32CubeMX

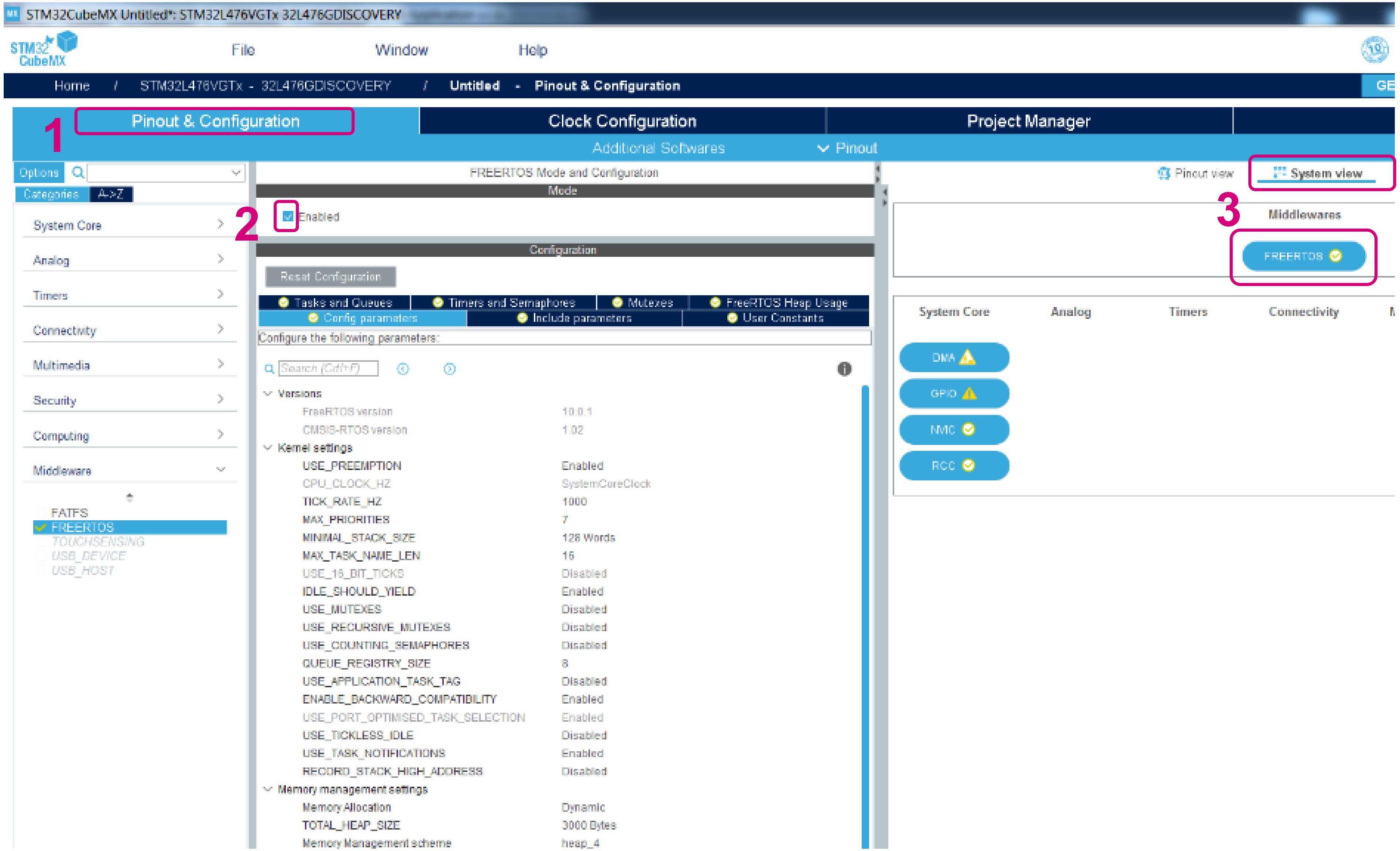
adding FreeRTOS middleware

- Go to **Pinout&Configuration** tab, select Categories→MiddleWare->FreeRTOS and check **Enabled** box in Mode window
- Go to **Configuration** tab to configure FreeRTOS parameters – refer to next slides for details



FreeRTOS configuration

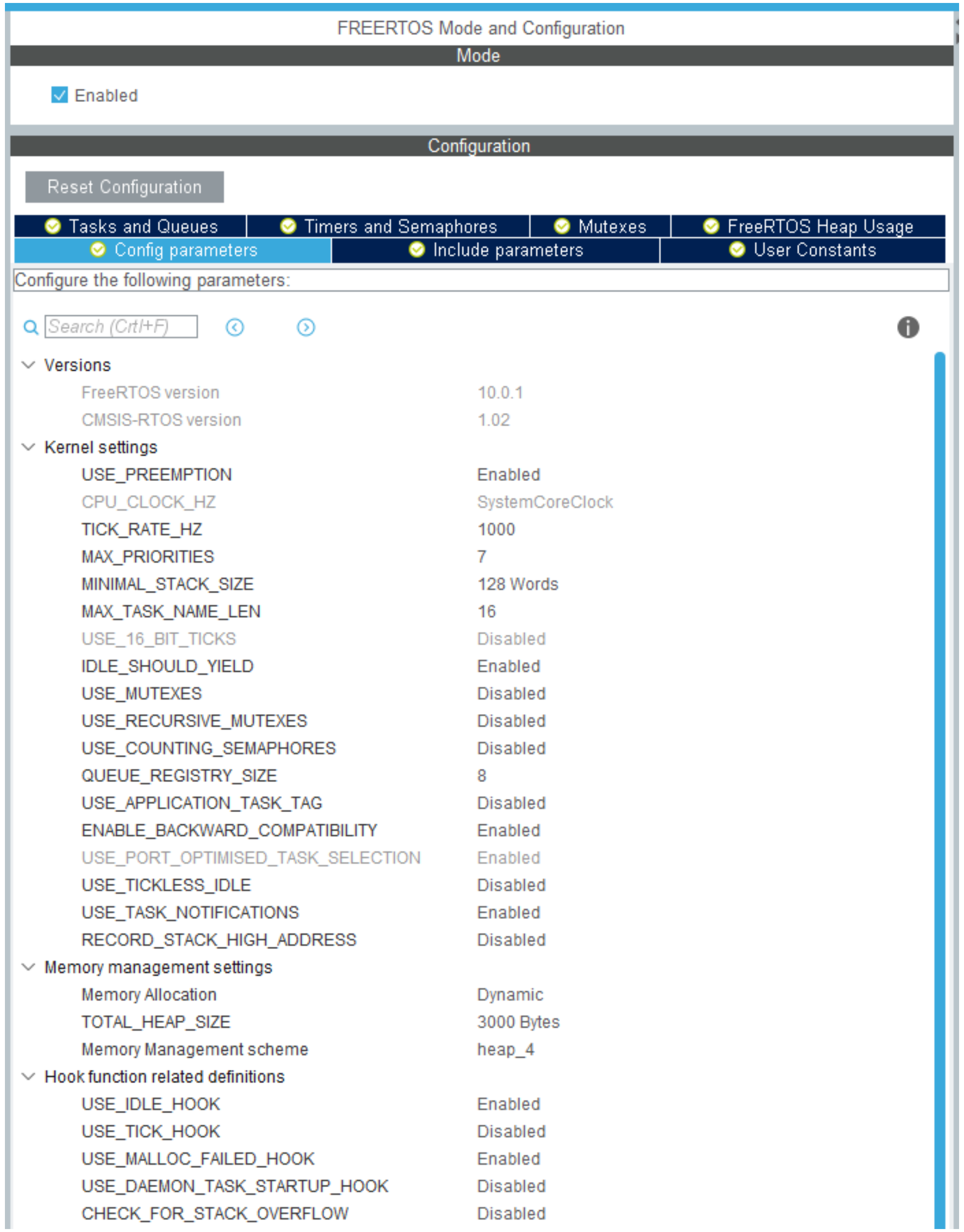
STM32CubeMX



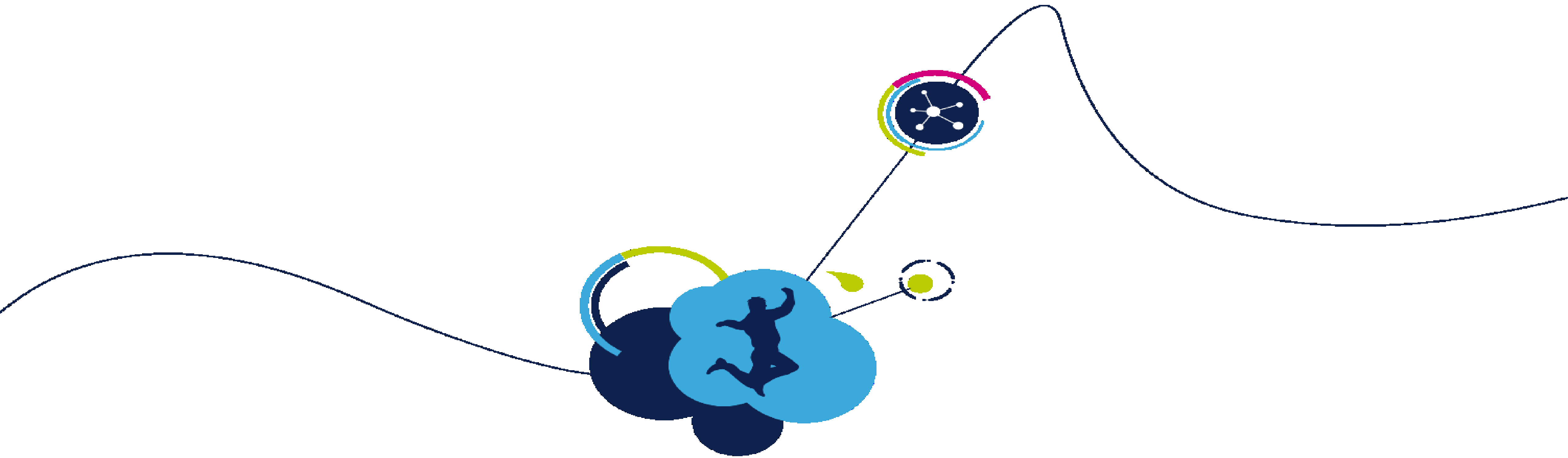
life.augmented

FreeRTOS configuration in STM32CubeMX

- **Config parameters** tab
  - Kernel settings
  - RTOS components settings
  - Memory setup
- **Include parameters** tab
  - Include some additional functions, not necessary for FreeRTOS run
- **Tasks and Queues** tab
  - Creation of tasks and queues
- **Timers and Semaphores** tab
  - Creation of timers and semaphores (binary, counting)
- **Mutexes** tab
  - Creation of mutexes



life.augmented



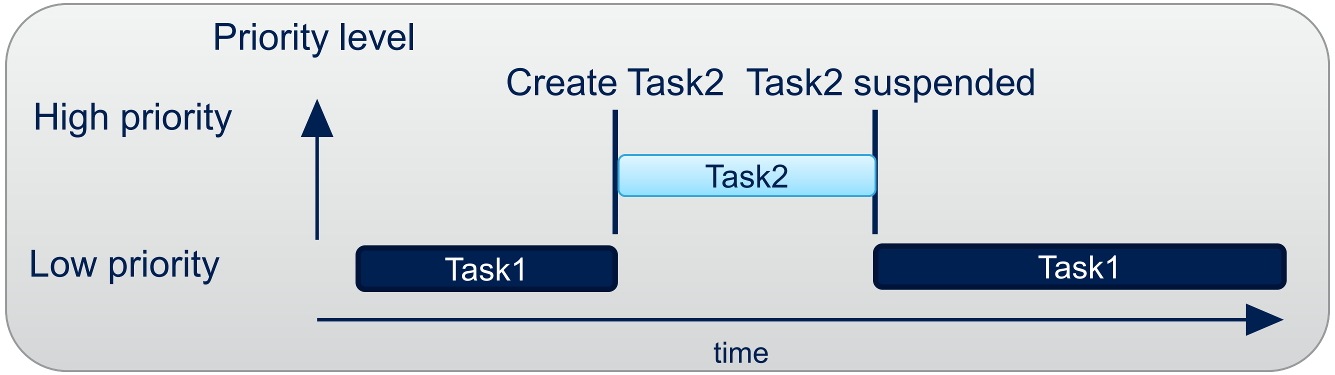
FreeRTOS configuration

- Configuration options are declared in file **FreeRTOSConfig.h**
- Important configuration options are:

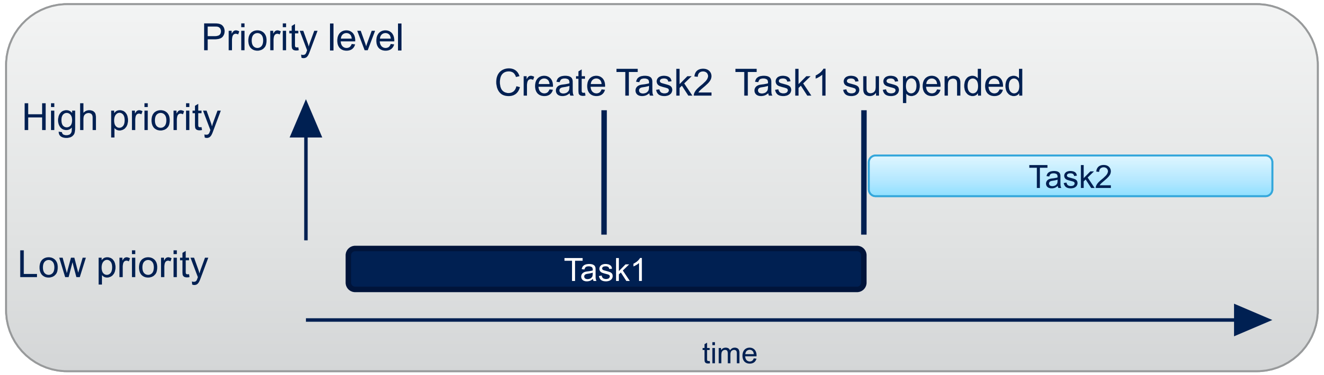
Config option	Description
configUSE_PREEMPTION	Enables Preemption
configCPU_CLOCK_HZ	CPU clock frequency in Hz
configTICK_RATE_HZ	Tick rate in Hz
configMAX_PRIORITIES	Maximum task priority
configTOTAL_HEAP_SIZE	Total heap size for dynamic allocation
configLIBRARY_LOWEST_INTERRUPT_PRIORITY	Lowest interrupt priority (0xF when using 4 cortex preemption bits)
configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY	Highest thread safe interrupt priority (higher priorities are lower numeric value)

Kernel settings

- Use preemption
  - If **enabled** use pre-emptive scheduling

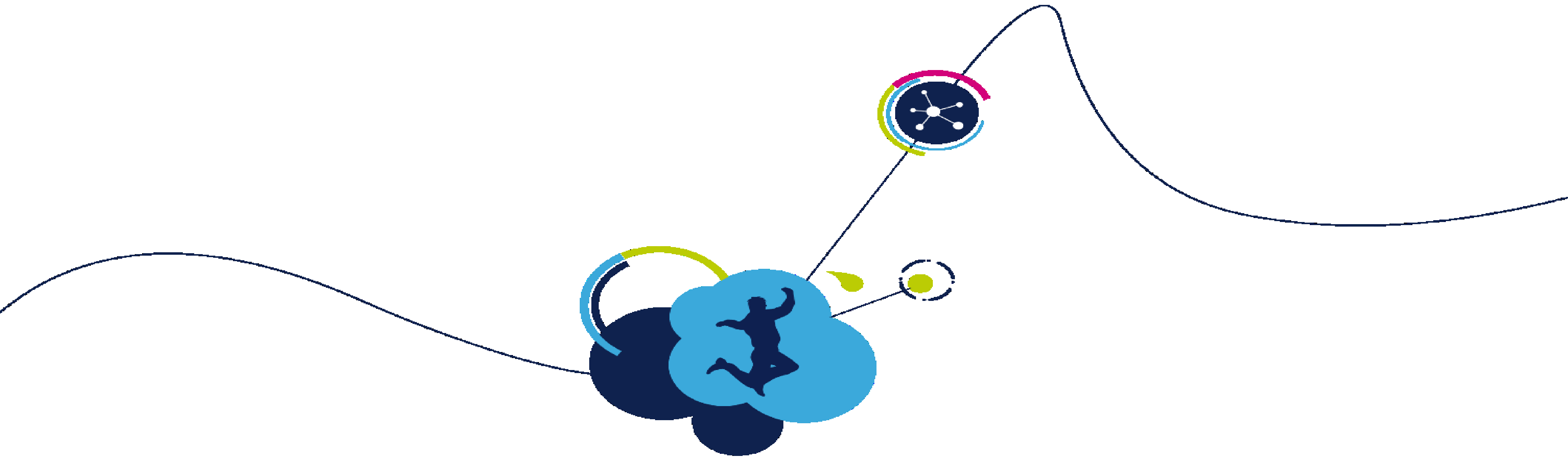


  - If **disabled** use co-operative scheduling



Kernel settings	
USE_PREEMPTION	Enabled
CPU_CLOCK_HZ	SystemCoreClock
TICK_RATE_HZ	1000
MAX_PRIORITIES	7
MINIMAL_STACK_SIZE	128 Words
MAX_TASK_NAME_LEN	16
USE_16_BIT_TICKS	Disabled
IDLE_SHOULD_YIELD	Enabled
USE_MUTEXES	Disabled
USE_RECURSIVE_MUTEXES	Disabled
USE_COUNTING_SEMAPHORES	Disabled
QUEUE_REGISTRY_SIZE	8
USE_APPLICATION_TASK_TAG	Disabled
ENABLE_BACKWARD_COMPATIBILITY	Enabled
USE_PORT_OPTIMISED_TASK_SELECTION	Enabled
USE_TICKLESS_IDLE	Disabled
USE_TASK_NOTIFICATIONS	Enabled
RECORD_STACK_HIGH_ADDRESS	Disabled
Memory management settings	
Memory Allocation	Dynamic
TOTAL_HEAP_SIZE	3000 Bytes
Memory Management scheme	heap_4
Hook function related definitions	
USE_IDLE_HOOK	Enabled
USE_TICK_HOOK	Disabled

life.augmented



FreeRTOS memory management HEAP

Heap (1/6)

- FreeRTOS uses a region of memory called Heap (into the RAM) to allocate memory for tasks, queues, timers , semaphores, mutexes and when dynamically creating variables. FreeRTOS heap is different than the system heap defined at the compiler level.

- When FreeRTOS requires RAM instead of calling the standard malloc it calls `PvPortMalloc()`. When it needs to free memory it calls `PvPortFree()` instead of the standard `free()`.

- FreeRTOS offers several heap management schemes that range in complexity and features. It includes five sample memory allocation implementations, each of which are described in the following link :
  - http://www.freertos.org/a00111.html

- The total amount of available heap space is set by **configTOTAL_HEAP_SIZE** which is defined in FreeRTOSConfig.h.

- The `xPortGetFreeHeapSize()` API function returns the total amount of heap space that remains unallocated (allowing the **configTOTAL_HEAP_SIZE** setting to be optimized). The total amount of heap space that remains unallocated is also available with xFreeBytesRemaining variable for heap management schemes 2 to 5.

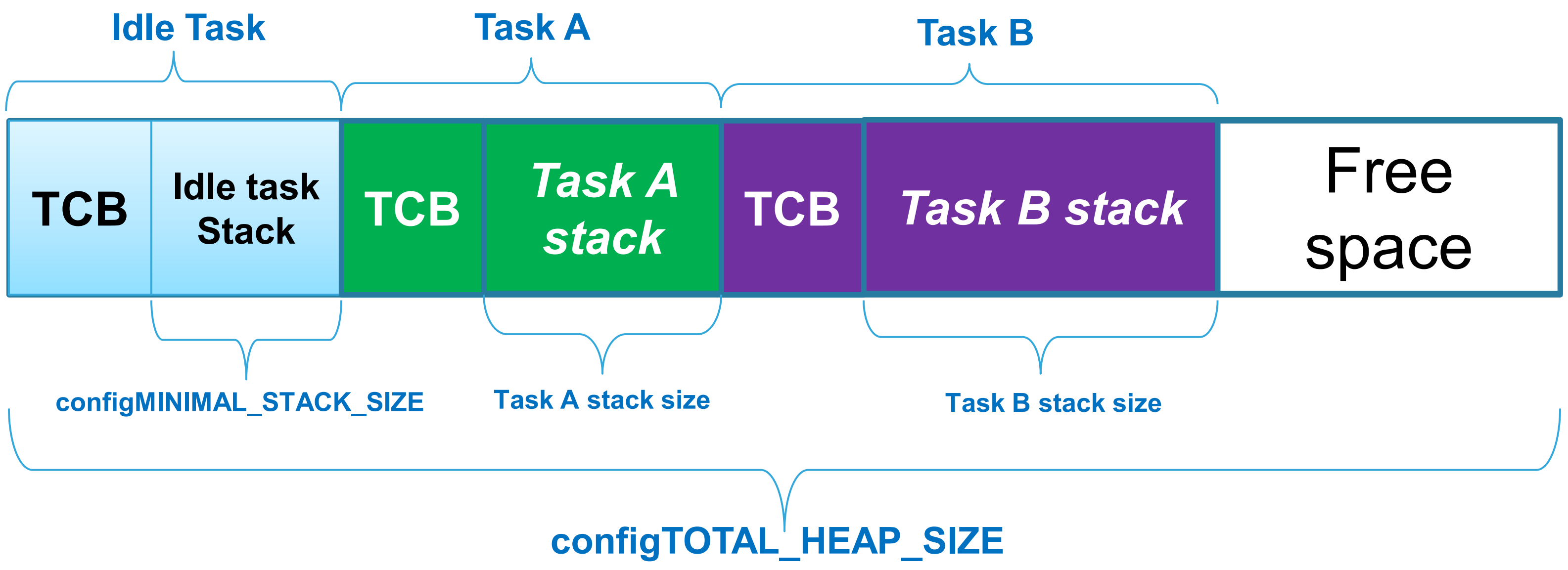
Heap (2/6)

- Each created task (including the idle task) requires a Task Control Block (TCB) and a stack that are allocated in the heap.
- The TCB size in bytes depends of the options enabled in the FreeRTOSConfig.h.
  - With minimum configuration the TCB size is 24 words i.e 96 bytes.
  - if **configUSE_TASK_NOTIFICATIONS** enabled add 8 bytes (2 words)
  - if **configUSE_TRACE_FACILITY** enabled add 8 bytes (2 words)
  - if **configUSE_MUTEXES** enabled add 8 bytes (2 words).
- The task stack size is passed as argument when creating at task. The task stack size is defined in words of 32 bits not in bytes.
  - osThreadDef(Task_A, Task_A_Function, osPriorityNormal, 0, stacksize);
- FreeRTOS requires to allocate in the heap for each task :
  - **number of bytes = TCB_size + (4 x task stack size)**
- **configMINIMAL_STACK_SIZE** defines the minimum stack size that can be used in words. the idle task stack size takes automatically this value

life.augmented

Heap (3/6)

- The necessary task stack size can be fine-tuned using the API `uxTaskGetStackHighWaterMark()` as follow:
  - Use an initial large stack size allowing the task to run without issue (example 4KB)
  - The API `uxTaskGetStackHighWaterMark()` returns the minimum number of free bytes (ever encountered) in the task stack. Monitor the return of this function within the task.
  - Calculate the new stack size as the initial stack size minus the minimum stack free bytes.
  - The method requires that the task has been running enough to enter the worst path (in term of stack consumption).



Heap (4/6)

- FreeRTOS requires to allocate in the heap for each message queue:
  - **number of bytes = 76 + queue_storage_area.**
  - **queue_storage_area (in bytes) = (element_size * nb_elements) + 16**
- When Timers are enabled (**configUSE_TIMERS** enabled) , the scheduler creates automatically the timers service task (daemon) when started. The timers service task is used to control and monitor (internally) all timers that the user will create. The timers task parameters are set through the fowling defines :
  - configTIMER_TASK_PRIORITY : priority of the timers task
  - configTIMER_TASK_STACK_DEPTH : timers task stack size (in words)
- The scheduler also creates automatically a message queue used to send commands to the timers task (timer start, timer stop ...)

Heap (5/6)

- The number of elements of this queue (number of messages that can be hold) are configurable through the define:
  - configTIMER_QUEUE_LENGTH.

- FreeRTOS requires to allocate in the heap for timers (in bytes):
  - Timers Daemon Task (in bytes) :
    - TCB_size + (4 x configTIMER_TASK_STACK_DEPTH)
  - Timers message queue : number of bytes = 76 + queue_storage_area
    - With queue_storage_area = (12 * configTIMER_QUEUE_LENGTH) + 16
  - For each timer created by the user (by calling osTimerCreate()) needs 48 bytes

- To save heap size (i.e RAM footprint) it is recommended to disable the define "**configUSE_TIMERS**" when timers are not used by the application

life.augmented

Heap (6/6)

- Each semaphore declared by the user application requires 88 bytes to be allocated in the heap.
- Each mutex declared by the user application requires 88 bytes to be allocated in the heap.
- To save heap size (i.e RAM footprint) it is recommended to disable the define **configUSE_MUTEXES** when mutexes are not used by the application (task TCB static size being reduced)

How to reduce RAM footprint (1/2)

- Optimize stack allocation for each task :
  - `uxTaskGetStackHighWaterMark()`. This API returns the minimum number of free bytes (ever encountered) in the task stack
  - `vApplicationStackOverflowHook()`. This API is a stack overflow callback called when a stack overflow is detected (available when activating the define configCHECK_FOR_STACK_OVERFLOW)

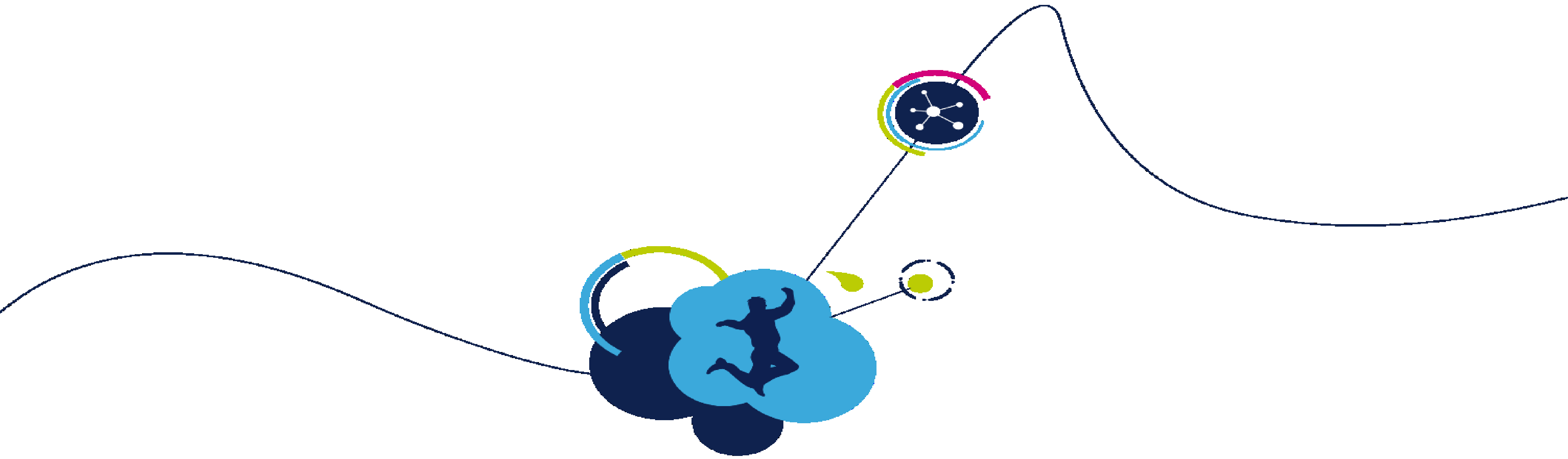
- Adjust heap dimensioning :
  - `xPortGetFreeHeapSize()`. API that returns the total amount of heap space that remains unallocated. Must be used after created all tasks, message queues, semaphores, mutexes in order to check the heap consumption and eventually re-adjust the application define " **configTOTAL_HEAP_SIZE**".
  - The total amount of heap space that remains unallocated is also available with xFreeBytesRemaining variable for heap management schemes 2 to 5

- If heap_1.c, heap_2.c, heap_4.c or heap_5.c are being used, and nothing in your application is ever calling malloc() directly (as opposed to pvPortMalloc()), then ensure the linker is not allocated a heap to the C library, it will never get used.

life.augmented

How to reduce RAM footprint (2/2)

- Recover and minimize the stack used by main and rationalize the number of tasks.
- If the application doesn't use any software timers then disable the define **configUSE_TIMERS**.
- If the application doesn't use any mutexe then disable the define **configUSE_MUTEXES**.
- **configMAX_PRIORITIES defines** the number of priorities available to the application tasks. Any number of tasks can share the same priority. Each available priority consumes RAM within the RTOS kernel so this value should not be set any higher than actually required by the application. It is recommended to declare tasks with contiguous priority levels: 1, 2, 3, 4, etc… rather than 10, 20, 30, 40, etc. The scheduler actually allocates statically the ready task list of size configMAX_PRIORITIES * list entry structure : so high value of configMAX_PRIORITIES shall be avoided to reduce RAM footprints

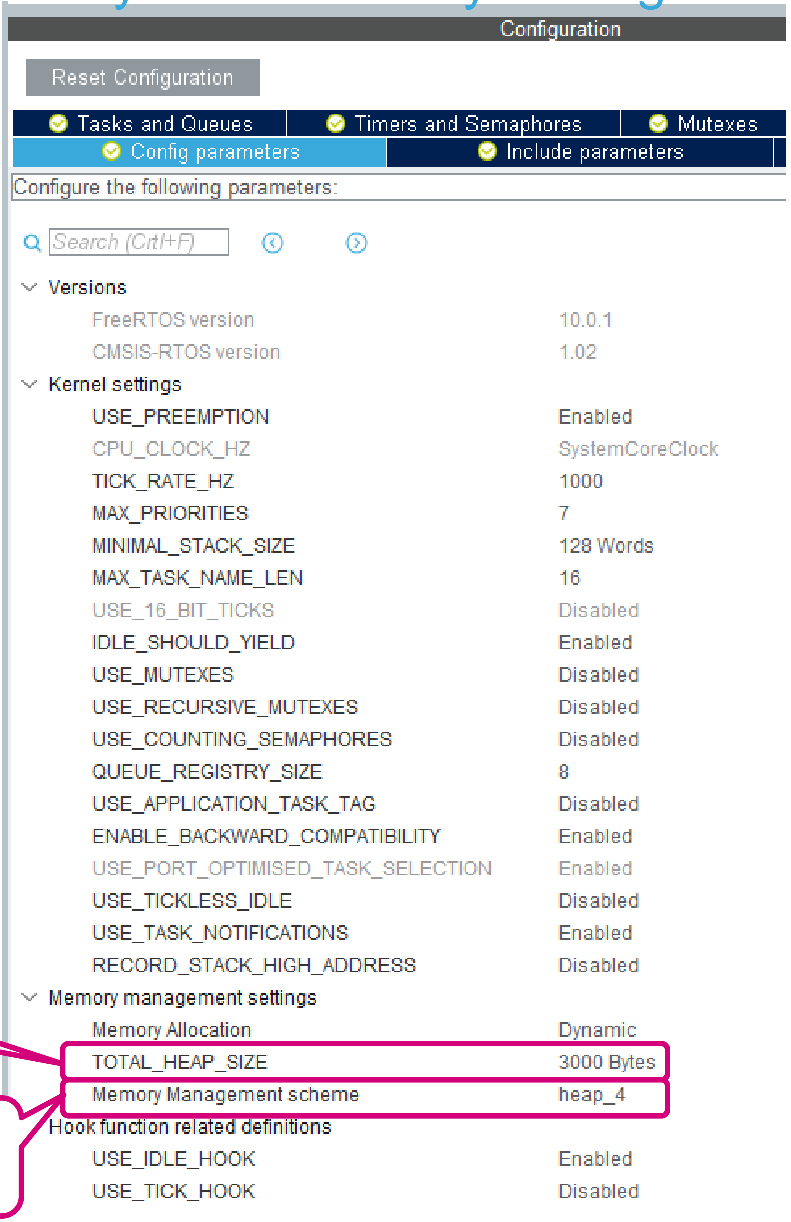


FreeRTOS
Memory allocation

FreeRTOS

Dynamic memory management

- FreeRTOS manages own heap for:
  - Tasks
  - Queues
  - Semaphores
  - Mutexes
  - Dynamic memory allocation
- It is possible to select type of memory allocation



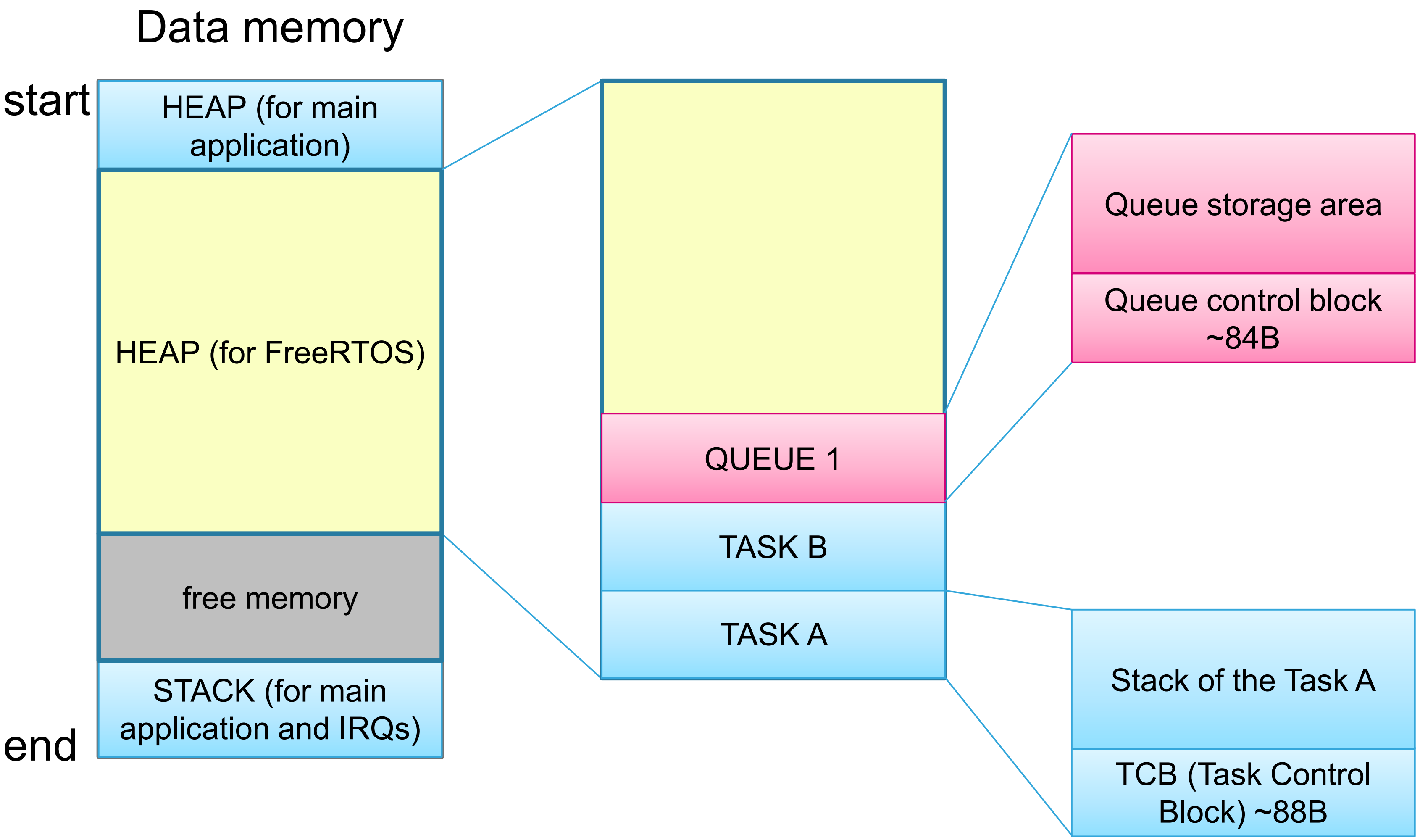
Total heap size for FreeRTOS

How is memory allocated and dealocated

life.augmented

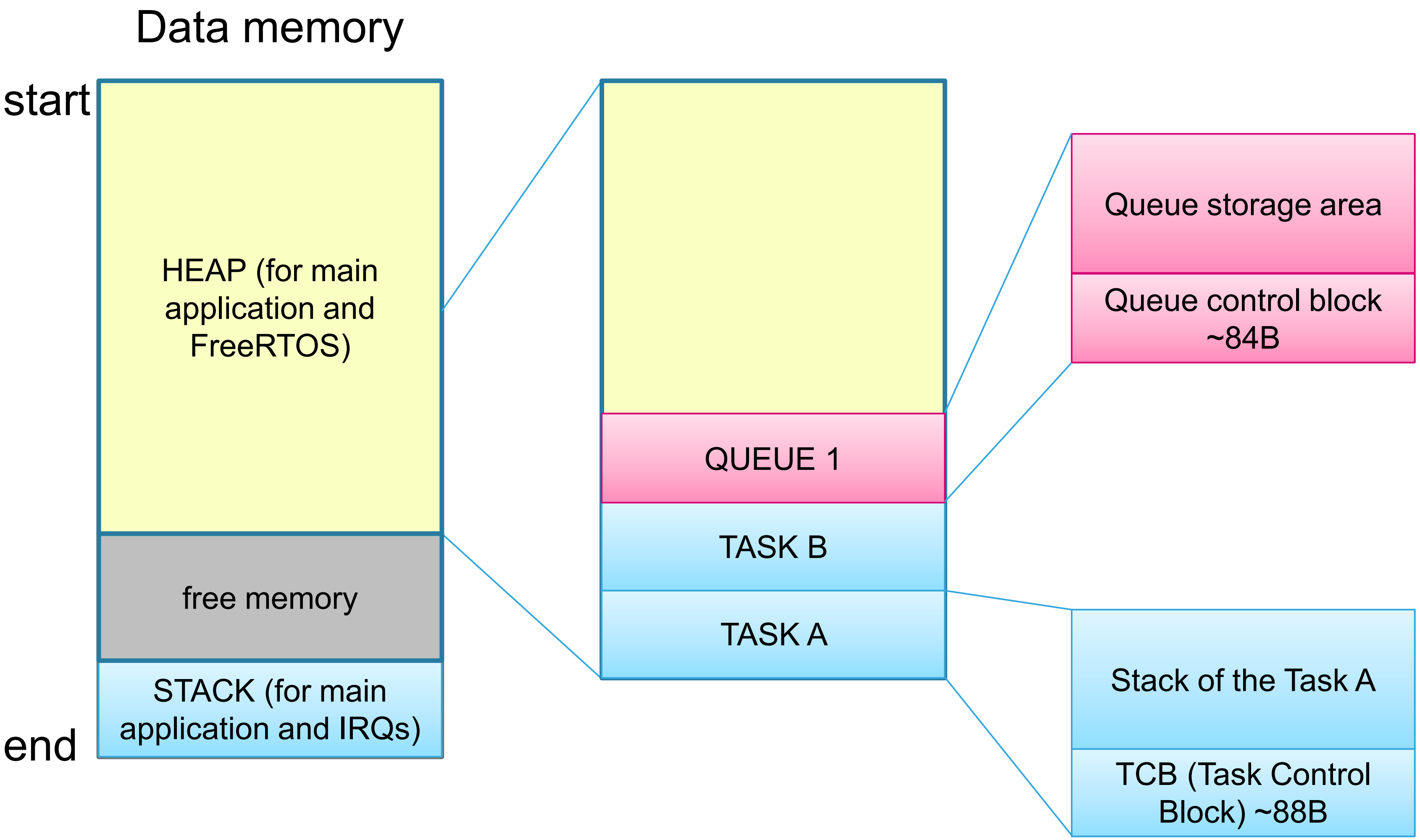
FreeRTOS in STM32

memory management (except Heap_3.c model)

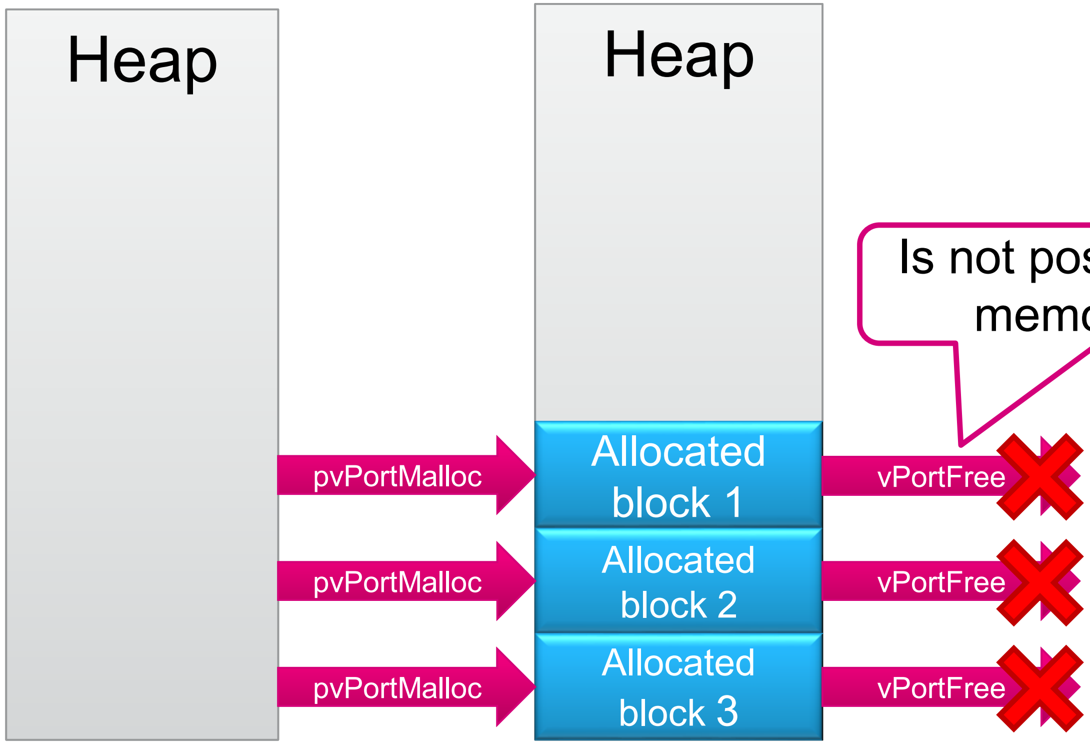


FreeRTOS in STM32

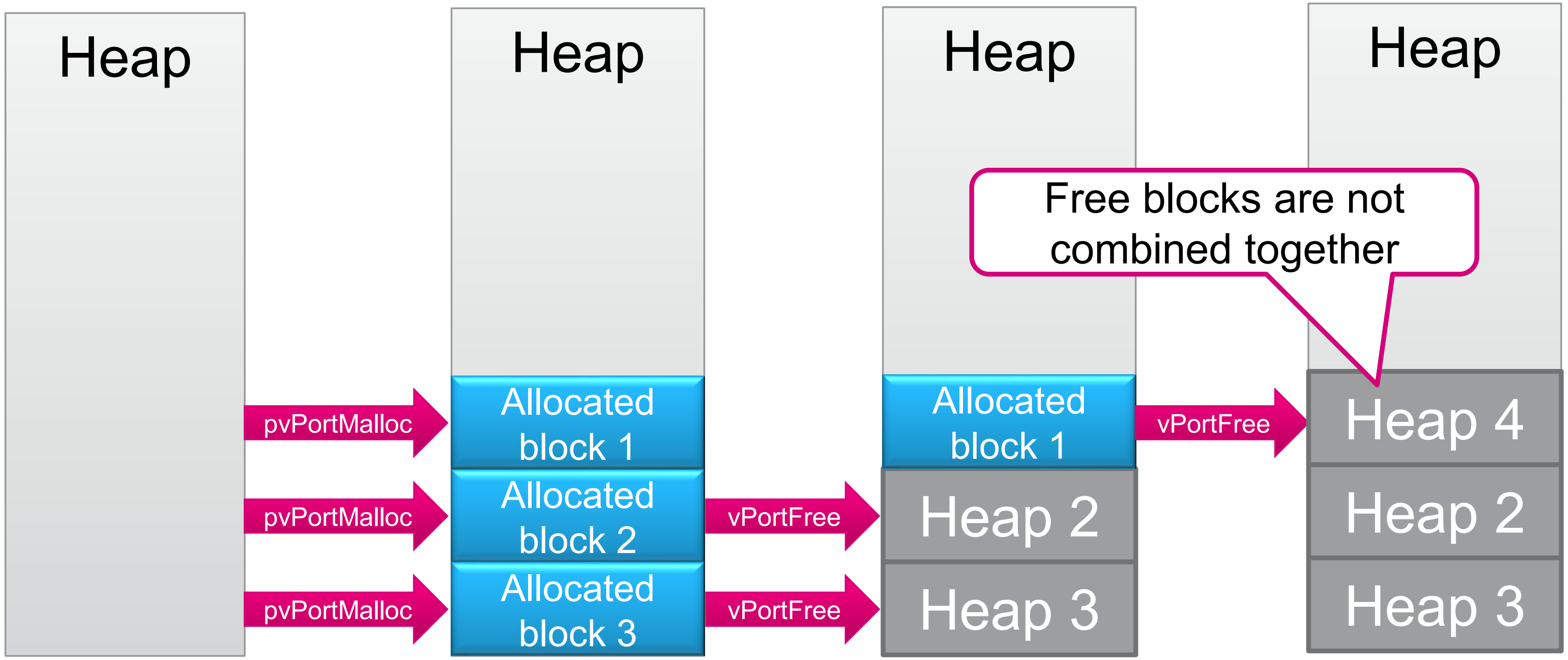
memory management (Heap_3.c model)



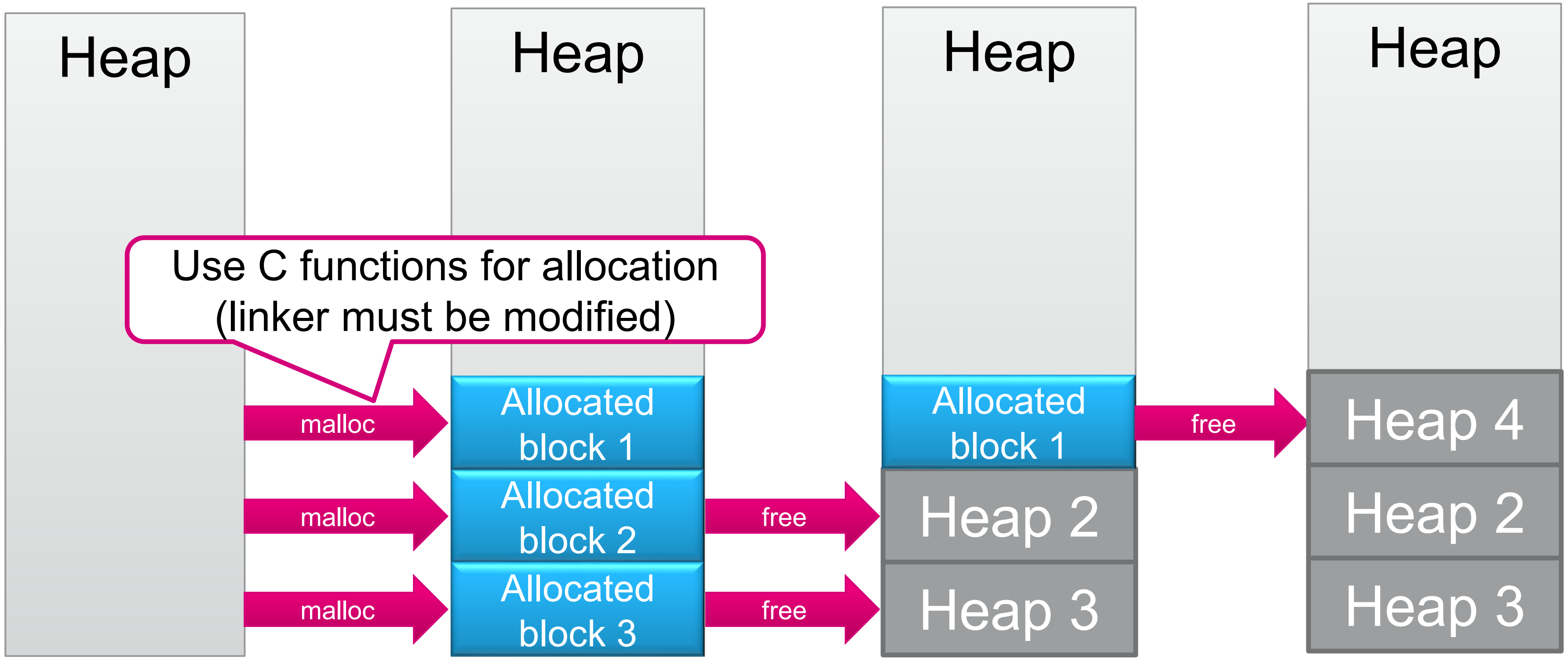
- **Heap_1.c**
  - Uses **first fit algorithm** to allocate memory. Simplest allocation method (deterministic), but does not allow freeing of allocated memory => could be interesting when no memory freeing is necessary



- **Heap_2.c**
  - Not recommended to new projects. Kept due to backward compatibility.
  - Implements the best fit algorithm for allocation
  - Allows memory free() operation but doesn't combine adjacent free blocks => risk of fragmentation

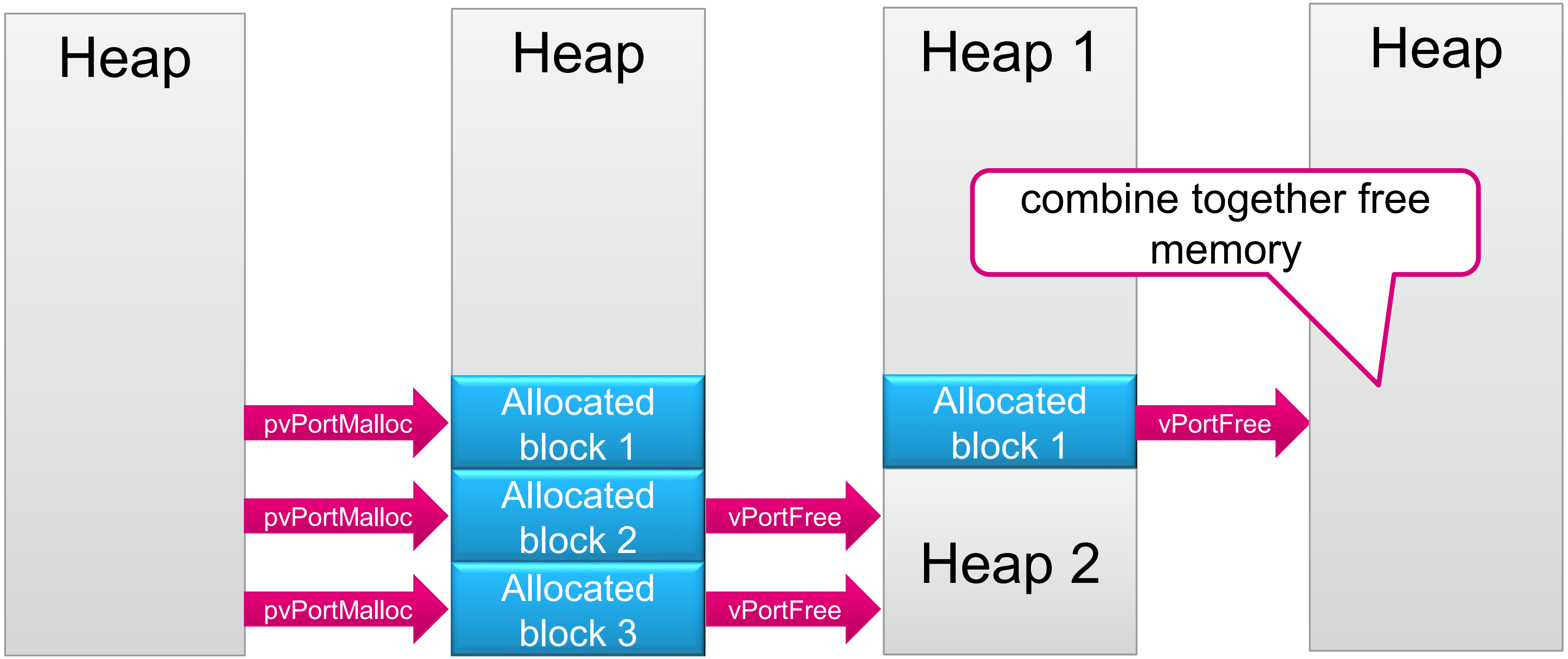


- **Heap_3.c**
  - Implements simple wrapper for standard C library malloc() and free(); wrapper makes these functions thread safe, but makes code increase and not deterministic
  - It uses linker heap region.
  - configTOTAL_HEAP_SIZE setting has no effect when this model is used.



• Heap_4.c (1/2)

- Uses **first fit algorithm** to allocate memory. It is able to combine adjacent free memory blocks into a single block
  => this model is used in STM32Cube examples



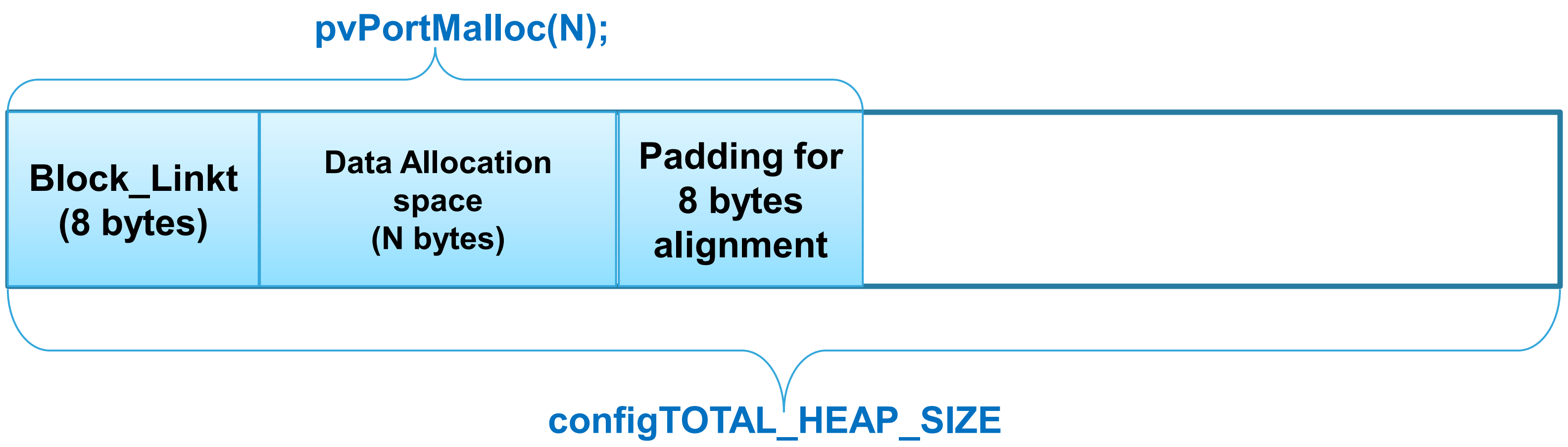
life.augmented

Heap_4.c (2/2) – place the heap in specific location

- The memory array used by heap_4 is declared within heap_4.c file and its start address is configured by the linker automatically.
- To use your own declaration `configAPPLICATION_ALLOCATED_HEAP` must be set to 1 (within FreeRTOSConfig.h file) and the array must be declared within user code with selected start address and size specified by `configTOTAL_HEAP_SIZE`.
- Memory array used by heap_4 is specified as:

```
uint8_t ucHeap[configTOTAL_HEAP_SIZE];
```

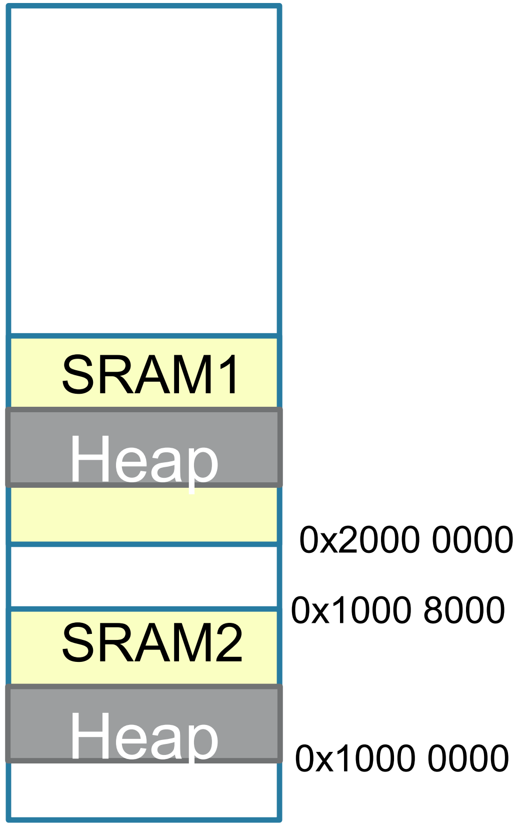
- Using heap_4.c : heap is organized as a linked list: for better efficiency when dynamically allocating/Freeing memory.
- As consequence when allocating “N” bytes in the heap memory using “pvPortMalloc” API it consumes:
  - Sizeof (BlockLink_t) (structure of the heap linked list) : **8 bytes**.
  - Data to be allocated itself : **N bytes**.
  - Add padding to total allocated size (N + 8) to be **8 bytes aligned** :
    - Example if trying to allocate 52 Bytes : it consumes from the heap : 52 + 8 = 60 bytes aligned to 8 bytes it gives 64 bytes consumed from the heap.



- **Heap_5.c (1/2)**
  - Fit algorithm able to combine adjacent free memory blocks into a single block using the same algorithms like in heap_4, but supporting different memory regions (i.e. SRAM1, SRAM2) being not in linear memory space
  - It is the only memory allocation scheme that must be explicitly initialized before any OS object cab be created (before first call of pvPortMalloc()).
  - To inialize this scheme vPortDefineHeapRegions() function should be called.
  - It specifies start address and size od each separate memory area.
  - An example for STM32L476 device with SRAM1 and SRAM2 areas is on the next slide

Heap_5.c (2/2)

- An example for STM32L476 device with SRAM1 and SRAM2 areas.:




```
#define SRAM1_OS_START (uint8_t *)0x2000 1000
#define SRAM1_OS_SIZE  0x0800 //2kB
#define SRAM2_OS_START (uint8_t *)0x1000 0000
#define SRAM2_OS_SIZE  0x1000 //4kB

Const HeapRegion_t xHeapRegions[] =
{
 {SRAM2_OS_START, SRAM2_OS_SIZE},
 {SRAM1_OS_START, SRAM1_OS_SIZE},
 {NULL,0} /*terminates the array*/
}

/*before call of any OS create function*/
vPortDefineHeapRegions(HeapRegions);
```

Lower address appears in the array first.

Manual memory allocation

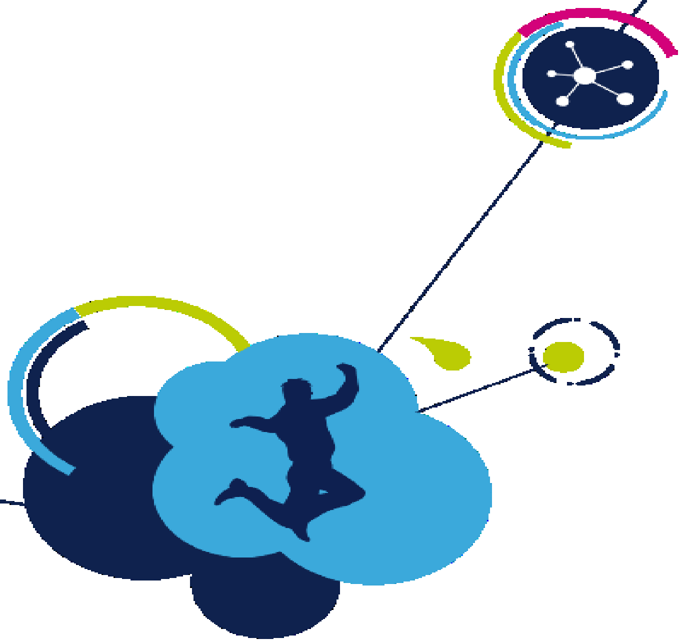
- There is an option to use alternative functions for memory management, however it is not recommended (inefficient) way of operation

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
osPoolId PoolHandle;

void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  osPoolDef(Memory,0x100,uint8_t);
  PoolHandle = osPoolCreate(osPool(Memory));
  uint8_t* buffer=osPoolAlloc(PoolHandle);
  /* Infinite loop */
  for(;;)
  {
    osDelay(5000);
  }
  /* USER CODE END 5 */
}
```

Create memory pool

Allocate memory from pool



FreeRTOS Scheduler

Multitasking 1/3

- **Cooperative** multitasking
  - Requires cooperation of all tasks
  - Context gets switched ONLY when RUNNING task
    - goes to BLOCKED state (i.e. by call `osDelay()` function) or
    - goes to READY state (i.e. by call `osThreadYield()` function) or
    - is put into SUSPEND mode by the system (other task)
  - Tasks are not preempted with higher priority tasks
  - No time slice preemption as well
  - It requires the following setting in FreeRTOSConfig.h:
    - #define **configUSE_PREEMPTION** **0**

life.augmented

Multitasking 2/3

- **Preemptive** multitasking (default in FreeRTOS)
  - Tasks with the same priority share CPU time
  - Context gets switched when:
    - Time slice has passed
    - Task with higher priority has come
    - Task goes to BLOCKED state (i.e. by call `osDelay()` function)
    - Task goes to READY state (i.e. by call `osThreadYield()` function)
  - It requires the following setting in FreeRTOSConfig.h:
    - #define **configUSE_PREEMPTION 1**

Multitasking 3/3

- **Cooperative with preemption by IRQ** multitasking
  - IRQs are used to trigger context switch
  - Preemptive system without time slice
  - It requires the following setting in FreeRTOSConfig.h:
    - #define **configUSE_PREEMPTION 0**

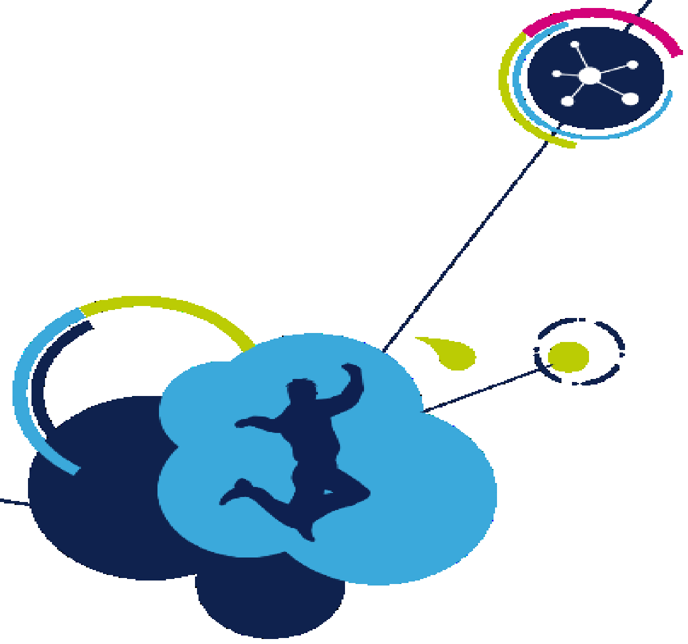
Scheduling 1/2

- The **scheduler** is an algorithm determining which task to execute.
  - Common point between schedulers is that they distinguish between tasks being ready to be executed (in READY state) and those being suspended for any reason (delay, waiting for mailbox, waiting for semaphore(s),…)
  - The main difference between schedulers is how they distribute CPU time between the tasks in READY state.

Scheduling 2/2

- In FreeRTOS **round-robin** scheduling algorithm is implemented:
  - Round-robin can be used with either preemptive or cooperative multitasking (**configUSE_PREEMPTION** in FreeRTOSConfig.h).
  - It works well if response time is not an issue or all tasks have same priority.
  - The possession of the CPU changes periodically after a predefined execution time called timeslice* (**configTICK_RATE_HZ** in FreeRTOSConfig.h)

*An exception to this rule are **critical sections**

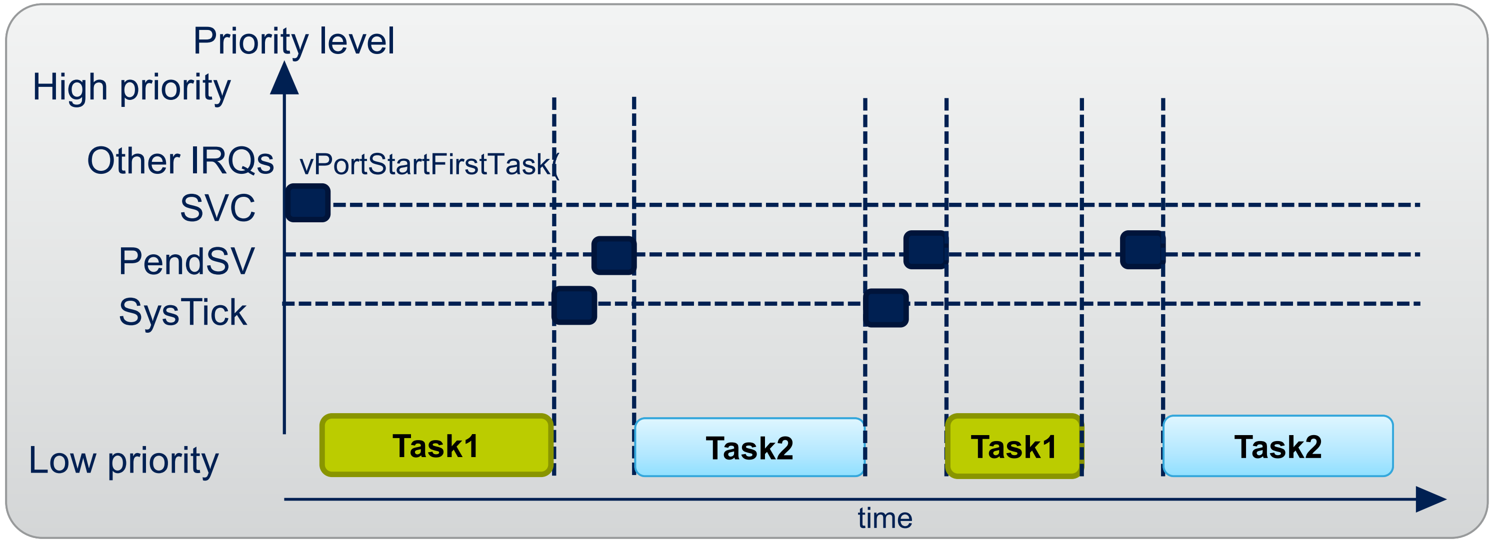


FreeRTOS – interrupts and connection to hardware

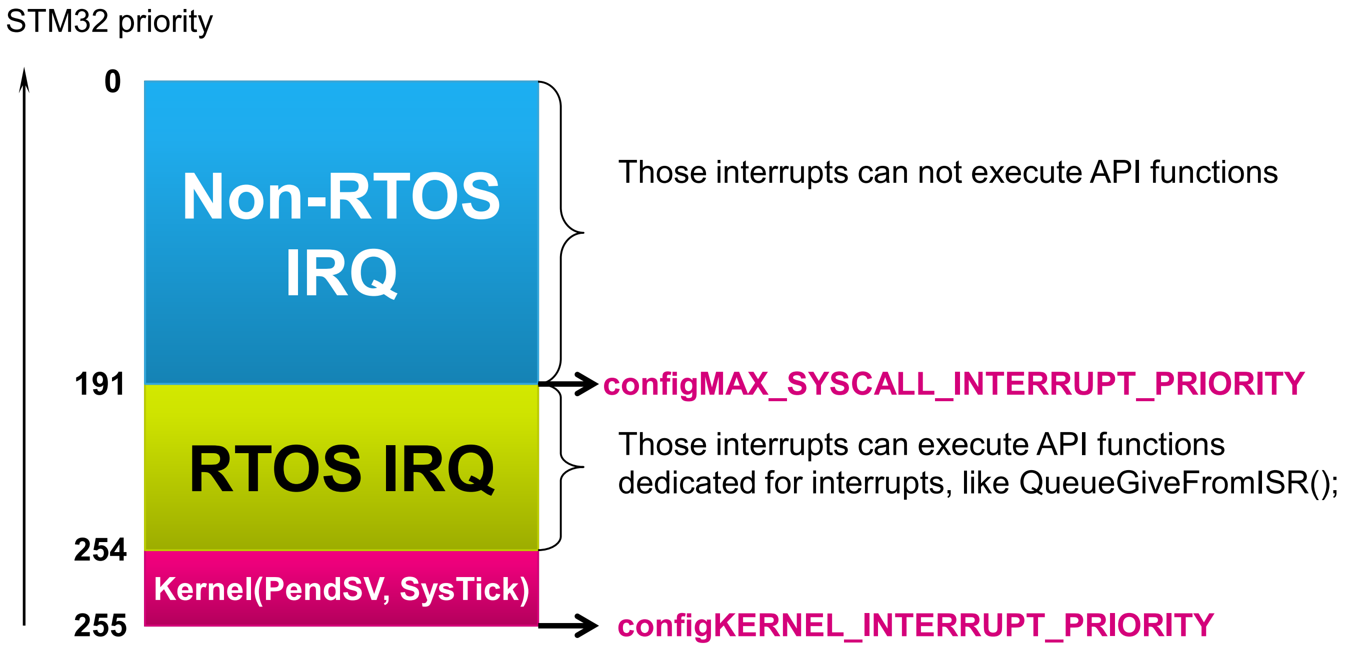
life.augmented

FreeRTOS OS interrupts

- **PendSV** interrupt
  - Used for task switching before tick rate
  - Lowest NVIC interrupt priority
  - Not triggered by any peripheral
- **SVC** interrupt
  - Interrupt risen by SVC instruction
  - SVC 0 call **used only once**, to start the scheduler (within vPortStartFirstTask() which is used to start the kernel)
- **SysTick** timer
  - Lowest NVIC interrupt priority
  - Used for task switching on configTICK_RATE_HZ regular timebase
  - Set PendSV if context switch is necessary



NVIC configuration



- FreeRTOS kernel and its irq procedures (PendSV, SysTick) have lowest possible interrupt priority (255) set in FreeRTOSConfig.h (**configKERNEL_INTERRUPT_PRIORITY**)
- There is a group of interrupts which can cooperate with FreeRTOS API by calling its functions. Maximum level for those peripherals (based on the position in vector table) is set in **configMAX_SYSCALL_INTERRUPT_PRIORITY**
- It is possible to use nested interrupts.

API functions in IRQ procedures

- Within **FreeRTOS API** there are dedicated functions to be executed within IRQ procedures. All of those functions has **FromISR** suffix in its names, like i.e.:

  xSemaphoreGiveFromISR(semaphore, *hp_task)

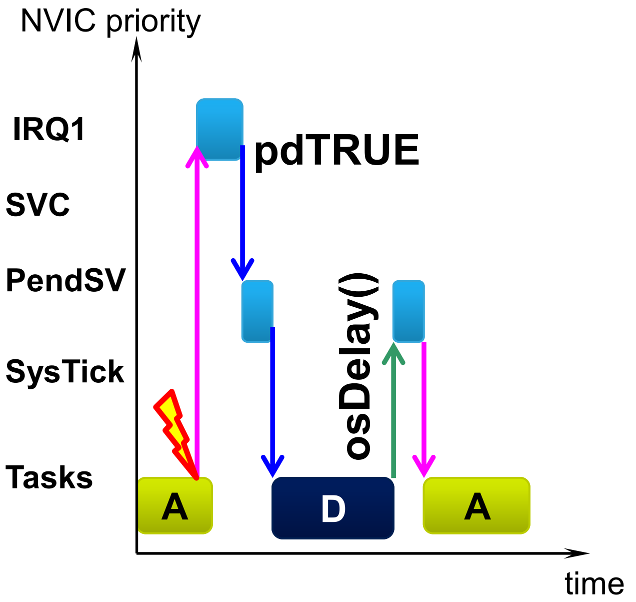
  vs

  xSemaphoreGive (semaphore)

- The only difference for the programmer is additional argument *hp_task. It is a pointer to the variable which is used to indicate whether operation on queue or semaphore within IRQ causes unblocking of the task with higher priority than currently running. If this parameter is pdTRUE, context switch (PendSV irq) should be requested by kernel before the interrupt exits.

- When using **CMSIS API**, this process is **automatically handled by the library** (by checking IPSR content) and is transparent for the programmer, i.e.:

```
osSemaphoreRelease(semaphore)
```



Example: Task A has been interrupted by IRQ1. During an interrupt, Task D with higher priority was unblocked, thus it will be executed once IRQ will finish

life.augmented

API functions in IRQ procedures

list of the functions which could be run from IRQ procedure

Function name (CMSIS_OS API)	Function name (FreeRTOS API)
osKernelSysTick()	xTaskGetTickCountFromISR()
osThreadResume()	xTaskResumeFromISR()
osThreadGetPriority()	uxTaskPriorityGetFromISR()
osSignalSet	xTaskGenericNotifyFromISR()
osMessagePut(), osMailPut()	xQueueSendFromISR()
osMessageGet(), osMailGet()	xQueueReceiveFromISR()
osMessageWaiting()	uxQueueMessagesWaitingFromISR()
osMutexWait(), osSemaphoreWait()	xSemaphoreTakeFromISR()
osMutexRelease(), osSemaphoreRelease()	xSemaphoreGiveFromISR()
osTimerStart()	xTimerChangePeriodFromISR()
osTimerStop()	xTimerStopFromISR()

life.augmented

FreeRTOS – boot sequence & timing

time

HW dependent:

- Configure the CPU clocks
- Initialize static and global variables that contain only the value zero (bss)
- Initialize variables that contain a value other than zero
- Perform any other hardware set up required

FreeRTOS related *)

- Create application queues, semaphores and mutexes (~500 CPU cycles/object)
- Create application tasks (~1100 CPU cycles/task)
- Start the RTOS scheduler (~1200 CPU cycles)

  The RTOS scheduler is started by calling vTaskStartScheduler(). The start up process includes configuring the tick interrupt, creating the idle task, and then restoring the context of the first task to run

life.augmented

*) calculations based on ARM CortexM3 device, using ARM RVDS compiler with low optimization level (1)
Source: *FreeRTOS FAQ – Memory Usage, Boot Time & Context Switch Times* on www.freertos.org web page

Idle task code

- Idle task code is generated automatically when the scheduler is started
- It is portTASK_FUNCTION() function within task.c file
- It is performing the following operations (in endless loop):
  - Check for deleted tasks to clean the memory
  - taskYIELD() if we are not using preemption (configUSE_PREEMPTION=0)
  - Get yield if there is another task waiting and we set configIDLE_SHOULD_YIELD=1
  - Executes vApplicationIdleHook() if configUSE_IDLE_HOOK=1
  - Perform low power entrance if configUSE_TICKLESS_IDLE!=0) -> let's look closer on this

FreeRTOS start

step by step 1/2

- FreeRTOS is started by `osKernelStart()` function (**main.c** file) from CMSIS_OS API
- It is calling `vTaskStartScheduler()` function (**cmsis_os.c** file) from FreeRTOS API
- It is creating an IDLE task (`xTaskCreate()`), then disable all interrupts (`portDISABLE_INTERRUPTS()`) to be sure that no tick will happened before or during call to `xPortStartScheduler()` function (**task.c** file)
- `xPortStartScheduler()` function (**port.c** file) is configuring lowest priority level for SysTick and PendSV interrupts, then it is starting the timer that generates the tick (in CortexM architecture usually it is SysTick), enables FPU if present (CortexM4) and starts the first task using `prvPortStartFirstTask()` function

FreeRTOS start

step by step 2/2

- `prvPortStartFirstTask()` function (**port.c** file, usually written in assembler) locates the stack and set MSP (used by the OS) to the start of the stack, then enables all interrupts. After this triggers software interrupt SVC
- As a result of SVC interrupt `vPortSVCHandler()` is called (**port.c** file)
- `vPortSVCHandler()` function (**port.c** file) restores the context, loads TCB (Task Control Block) for the first task (highest priority) form ready list and starts executing this task

FreeRTOS – lists management

name	Description	conditions
ReadyTasksLists[0] … ReadyTasksList[configMAX_PRIORITIES]	Prioritized ready tasks lists separate for each task priority (up to configMAX_PRIORITIES Value stored in FreeRTOSConfig.h)	configMAX_PRIORITIES
TasksWaitingTermination	List of tasks which have been deleted but their memory pools are not freed yet.	INCLUDE_vTaskDelete == 1
SuspendedTaskList	List of tasks currently suspended	INCLUDE_vTaskSuspend == 1
PendingReadyTaskList	Lists of tasks that have been read while the scheduler was suspended	-
DelayedTaskList	List of delayed tasks	-
OverflowDelayedTaskList	List of delayed tasks which have overflowed the current tick count	-

There is no dedicated list for task in Running mode (as we have only one task in this state at the moment), but the currently run task ID is stored in variable **pxCurrentTCB**

life.augmented

API - Operations on scheduler

- Start the scheduler
  `osKernelStart()`
  - Set priorities for PendSV and SysTick IRQs (minimum possible)
  - Starts kernel of the FreeRTOS (by executing SVC procedure)
  - IDLE task is created automatically
    (with handler or without it if INCLUDE_xTaskGetIdleTaskHandle is not defined)
  - There could be another thread creation done.

- Stop the scheduler -> **not implemented in STM32 (function `vTaskEndScheduler()` is empty)**

- Check if the RTOS kernel is already started
  `osKernelRunning()`
  - Return values:
    0 – RTOS is not started,
    1 – RTOS already started,
    -1 this feature is disabled in FreeRTOS configuration (INCLUDE_xTaskGetSchedulerState)

- Get the value of the Kernel SysTick timer
  `osKernelSysTick()`
  - Returns value of the SysTick timer (uint32)

FreeRTOS Tasks

What is Task?

- It is C function:

```
FirstTask(void const * argument)
```

- It should be run within infinite loop, like:

```
for(;;)
{
  /* Task code */
}
```

- It can be used to generate any number of tasks (separate instances)
- It has its own part of stack (each instance), and priority
- It can be in one of 4 states (RUNNING, BLOCKED, SUSPENDED, READY)
- It is created and deleted by calling API functions of the CMSIS_OS (`osThreadCreate()` and `osThreadDelete()`)

life.augmented

Task structure

- A task consists of three parts:
  - The **program code** (ROM)
  - A **stack**, residing in a RAM area that can be accessed by the stack pointer (The stack has the same function as in a single-task system: storage of return addresses of function calls, parameters and local variables, and temporary storage of intermediate calculation results and register values.
  - **TCB** - task control block (data structure assigned to a task when it is created. It contains status information of the task, including the stack pointer, task priority, current task status)
- Two calls to `pvPortMalloc()` are made during task creation. First one allocates TCB, second one allocates the task stack (it is taken from declared FreeRTOS heap area).
- The process of saving the context of a task that is being suspended and restoring the context of a task being resumed is called **context switching**.

Task Control Block (TCB)

Name	Description	condition
*pxTopOfStack	Points to the location of the last item placed on the tasks stack. THIS MUST BE THE FIRST MEMBER OF THE TCB STRUCT	
xMPUSettings	The MPU settings are defined as part of the port layer. THIS MUST BE THE SECOND MEMBER OF THE TCB STRUCT	portUSING_MPU_WRAPPERS == 1
xGenericListItem	The list that the state list item of a task is reference from denotes the state of that task (Ready, Blocked, Suspended).	
xEventListItem	Used to reference a task from an event list	
uxPriority	The priority of the task. 0 is the lowest priority	
*pxStack	Points to the start of the stack	
Task Name	Descriptive name given to the task when created. Facilitates debugging only	
*pxEndOfStack	Points to the end of the stack on architectures where the stack grows up from low memory	portSTACK_GROWTH > 0
uxCriticalNesting	Holds the critical section nesting depth for ports that do not maintain their own count in the port layer	portCRITICAL_NESTING_IN_TCB == 1
uxTCBNumber	Stores a number that increments each time a TCB is created. It allows debuggers to determine when a task has been deleted and then recreated.	configUSE_TRACE_FACILITY == 1
uxTaskNumber	Stores a number specifically for use by third party trace code	configUSE_TRACE_FACILITY == 1
uxBasePriority	The priority last assigned to the task - used by the priority inheritance mechanism	configUSE_MUTEXES == 1
uxMutexesHeld		configUSE_MUTEXES == 1
pxTaskTag		configUSE_APPLICATION_TASK_TAG == 1
ulRunTimeCounter	Stores the amount of time the task has spent in the Running state	configGENERATE_RUN_TIME_STATS == 1
_reent xNewLib_reent	Allocate a Newlib reent structure that is specific to this task. Note Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for resulting newlib operation. User must be familiar with newlib and must provide system-wide implementations of the necessary stubs.	configUSE_NEWLIB_REENTRANT == 1

life.augmented

Task Control Block (TCB)

Main fields within TCB (task.c file)

```
typedef struct tskTaskControlBlock
{
 volatile StackType_t  *pxTopOfStack; //Points to the location of the last item placed on the tasks stack
           …
 ListItem_t xStateListItem;        //The list that the state list item of a task is reference from denotes
                                  //the state of that task (Ready, Blocked, Suspended )
 ListItem_t xEventListItem;        //Used to reference a task from an event list
 UBaseType_t uxPriority;           //The priority of the task.  0 is the lowest priority
 StackType_t *pxStack; //Points to the start of the stack
 char pcTaskName[ configMAX_TASK_NAME_LEN ];//Descriptive name given to the task when created.
           …
#if ( configUSE_MUTEXES == 1 )
 UBaseType_t uxBasePriority; //The priority last assigned to the task - for priority inheritance
 UBaseType_t uxMutexesHeld;
#endif
           …
#if( configUSE_TASK_NOTIFICATIONS == 1 )
 volatile uint32_t ulNotifiedValue;
 volatile uint8_t ucNotifyState;
#endif
           …
} tskTCB;
```

life.augmented

Task function example

```
void FirstTask(void const * argument)
{
 /* task initialization */

 for(;;)
 {
  /* Task code */
 }

 /* we should never be here */
}
```

Run once at first run of each task instance

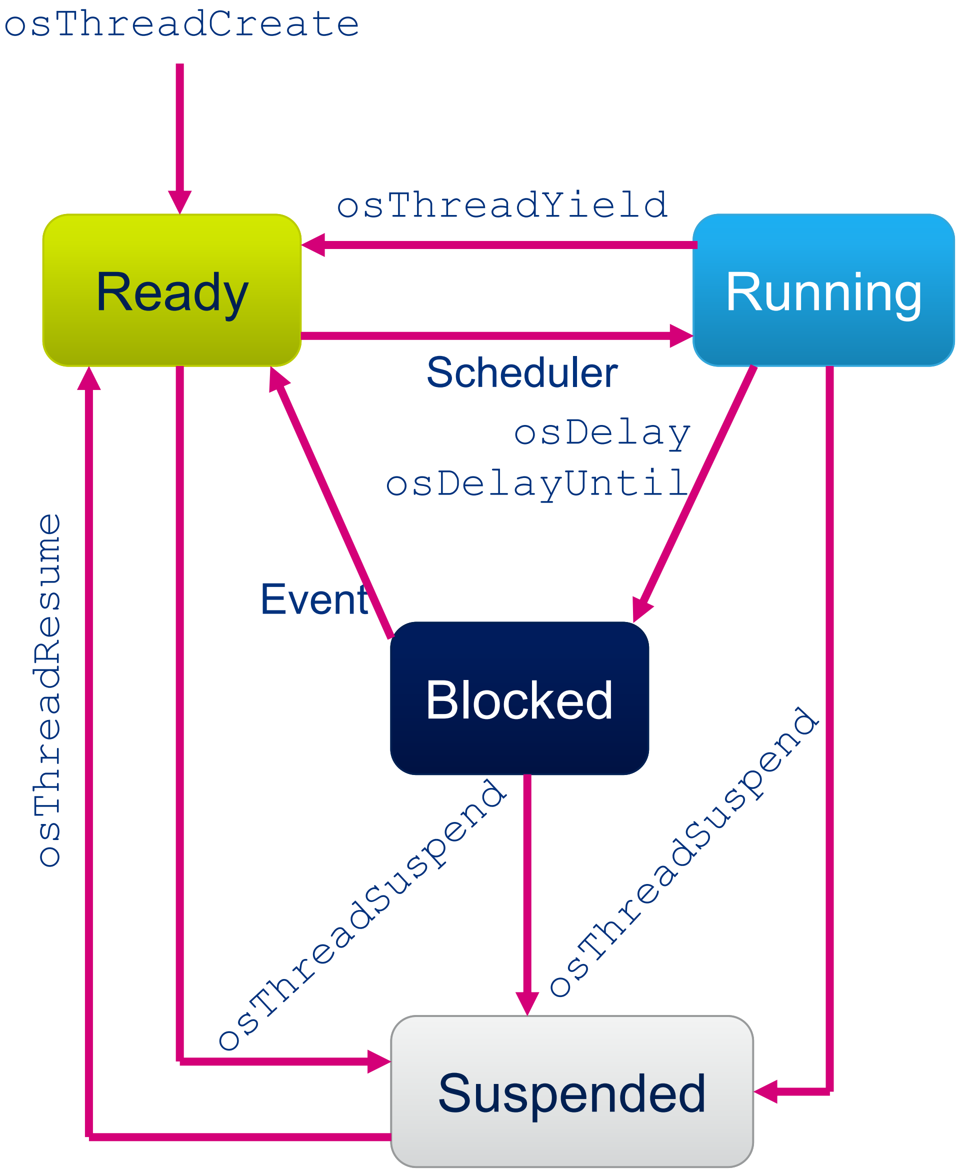
Run when task instance is in RUN mode

Should be never executed.

life.augmented

Task states

- **Ready**
  - Task is ready to be executed but is not currently executing because a different task with equal or higher priority is running
- **Running**
  - Task is actually running (only one can be in this state at the moment)
- **Blocked**
  - Task is waiting for either a temporal or an external event
- **Suspended**
  - Task not available for scheduling, but still being kept in memory



Task states – CMSIS_OS

Tasks states are stored within `osThreadState` enum (**cmsis_os.h** file)

State name	value	comment
osThreadRunning	0	RUNNING
osThreadReady	1	READY
osThreadBlocked	2	BLOCKED
osThreadSuspended	3	SUSPEND
osThreadDeleted	4	Task has been deleted, but its TCB has not yet been freed
osThreadError	0x7FFFFFFF	Error code

Task priorities

- Each task is assigned a priority from [**tskIDLE_PRIORITY**] (defined in task.h) to [**MAX_PRIORITIES – 1**] (defined in *FreeRTOSConfig.h*)
- The order of execution of tasks depends on this priority
- The scheduler activates the task that has the highest priority of all tasks in the **READY** state.
- Task with higher priority can preempt running task if **configUSE_PREEMPTION** (in *FreeRTOSConfig.h*) is set to 1
- Task priorities can be changed during work of the application

lower number = lower priority

Task priorities

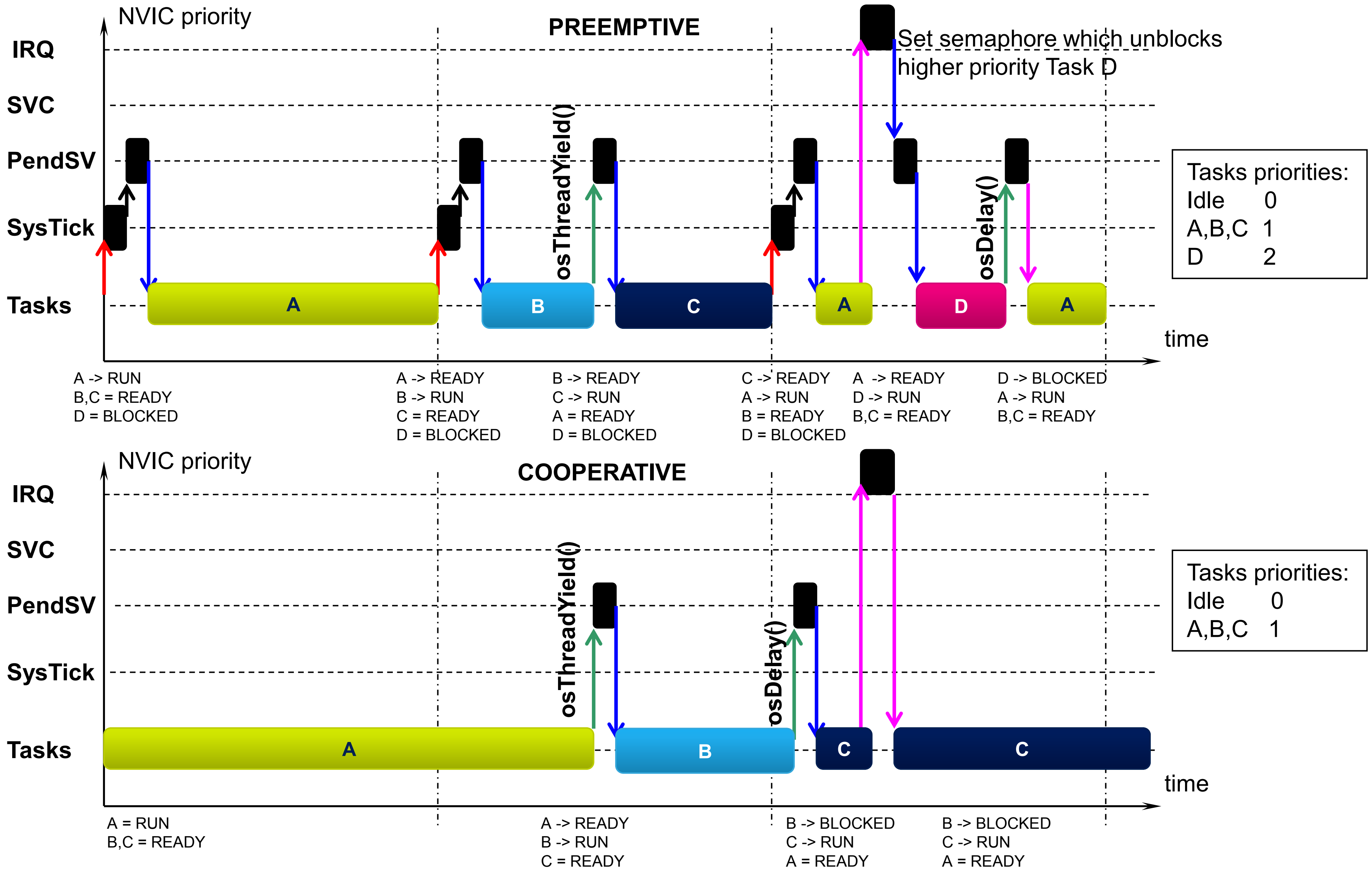
CMSIS_OS

Tasks priorities can be set within `osPriority` enum (**cmsis_os.h** file)

Priority name	value	comment
`osPriorityIdle`	-3	priority: idle (lowest)
`osPriorityLow`	-2	priority: low
`osPriorityBelowNormal`	-1	priority: below normal
`osPriorityNormal`	0	priority: normal (default)
`osPriorityAboveNormal`	1	priority: above normal
`osPriorityHigh`	2	priority: high
`osPriorityRealtime`	3	priority: realtime (highest)
`osPriorityError`	0x84	system cannot determine priority or thread has illegal priority

life.augmented

Context switching



Context switching

list.h

Tasks are grouped within lists at `List_t` objects (**list.h** file)

Field name	comment
`listFIRST_LIST_INTEGRITY_CHECK_VALUE`	known test value – not used
`UBaseType_t`	priority: low
`ListItem_t *`	Used to walk through the list. Points to the last item returned by a call to listGET_OWNER_OF_NEXT_ENTRY ()
`MiniListItem_t`	List item that contains the maximum possible item value meaning it is always at the end of the list and is therefore used as a marker.
`listSECOND_LIST_INTEGRITY_CHECK_VALUE`	known test value – not used

life.augmented

Context switching

list.h

Tasks are grouped within lists at `ListItem_t` objects (**list.h** file)

Field name	comment
`listFIRST_LIST_INTEGRITY_CHECK_VALUE`	known test value – not used
`TickType_t`	The value being listed. In most cases this is used to sort the list in descending order.
`ListItem_t *`	Pointer to the next ListItem_t in the list.
`ListItem_t *`	Pointer to the previous ListItem_t in the list.
`Void *`	Pointer to the object (normally a TCB) that contains the list item. There is therefore a two way link between the object containing the list item and the list item itself.
`Void *`	Pointer to the list in which this list item is placed (if any).
`listSECOND_LIST_INTEGRITY_CHECK_VALUE`	known test value – not used

Context switching

list.h

Tasks are grouped within lists at `MiniListItem_t` objects (**list.h** file)

Field name	comment
`listFIRST_LIST_INTEGRITY_CHECK_VALUE`	known test value – not used
`TickType_t`	The value being listed. In most cases this is used to sort the list in descending order.
`ListItem_t *`	Pointer to the next ListItem_t in the list.
`ListItem_t *`	Pointer to the previous ListItem_t in the list.

life.augmented

FreeRTOS – context switching

tick source - step by step

- Tick timer (CortexM architecture uses SysTick) interrupt causes execution of `xPortSysTickHandler()` (**port.c** file)
- `xPortSysTickHandler()` (usually written in assembly):
  - blocks all interrupts (as its own priority is the lowest possible) using `portDISABLE_INTERRUPTS()` macro (**portmacro.h** file)
  - Activates PendSV bit to run an interrupt what executes `xPortPendSVHandler()` function (**port.c** file):
    - Calls `vTaskSwitchContext()` function (**task.c** file), which is calling a macro `taskSELECT_HIGHEST_PRIORITY_TASK()` (**task.c** file) to select the READY task on the highest possible priority list.
  - Unblocks all interrupts using `portENABLE_INTERRUPT()` macro (**portmacro.h** file)

FreeRTOS – context switch time (1/2)

- Context switch time depends on the port, compiler and configuration. A context switch time of 84 CPU cycles was obtained under the following test conditions:
  - FreeRTOS ARM Cortex-M3 port for the Keil compiler
  - Stack overflow checking turned off
  - Trace features turned off
  - Compiler set to optimization for speed
  - configUSE_PORT_OPTIMISED_TASK_SELECTION set to 1 in FreeRTOSConfig.h

Remarks:

- Under these test conditions the context switch time is not dependent on whether a different task was selected to run or the same task was selected to continue running.
- The ARM Cortex-M port performs all task context switches in the PendSV interrupt. The quoted time does not include interrupt entry time.
- The quoted time includes a short section of C code. It has been determined that 12 CPU cycles could have been saved by providing the entire implementation in assembly code. It is considered that the benefit of maintaining a short section of generic C code (for reasons of maintenance, support, robustness, automatic inclusion of features such as tracing, etc.) outweighs the benefit of removing 12 CPU cycles from the context switch time.
- The Cortex-M CPU registers that are not automatically saved on interrupt entry can be saved with a single assembly instruction, then restored again with a further single assembly instruction. These two instructions on their own consume 12 CPU cycles.

*) source: *FreeRTOS FAQ – Memory Usage, Boot Time & Context Switch Times* on www.freertos.org web page

FreeRTOS – context switch time (2/2)

- Context switch time can be much longer in CortexM4 and CortexM7 based devices with Floating Point Unit due to necessity of stacking FPU registers (additional 17 32bit registers: S0-S15 and FPSCR).
- Rest of FPU registers (S16-S31) should be handled by software
- Within PendSV handler there is a check done whether floating point unit instruction has been used and based on this informaiton those registers are stacked/unstacked from/for current task or not:

```
/* Is the task using the FPU context?
   If so, push high vfp registers. */
tst      r14, #0x10
it       eq
vstmdbeq r0!, {s16-s31}
```

- And then on PendSV exit after the task switch:

```
/* Is the task using the FPU context?
   If so, pop the high vfp registers too. */
tst      r14, #0x10
it       eq
vldmiaeq r0!, {s16-s31}
```

- More information can be found in **Application note 298** from ARM.

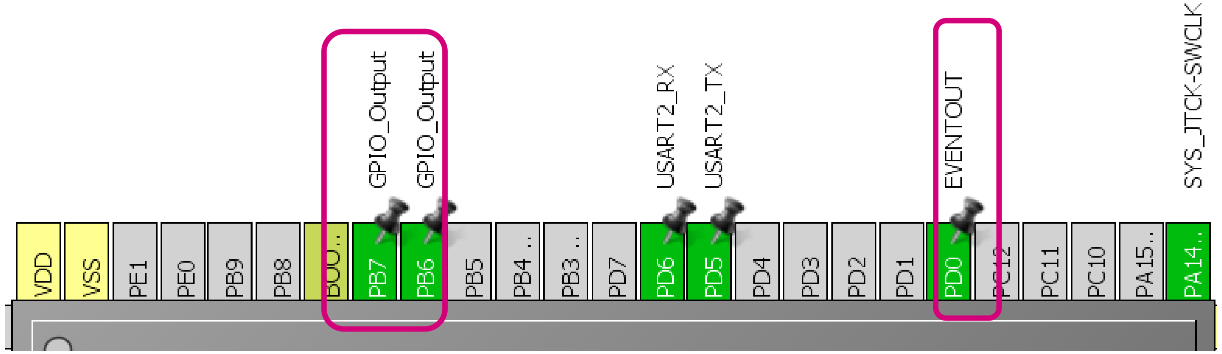
life.augmented

Context switching time

STM32CubeMX modifications

Within STM32CubeMX, pinout tab:

- Configure PB6, PB7 as GPIO_Output
- Configure PD0 as EVENTOUT



Re-generate the code and within the code please add some modifications:

1. To set both pins (PB6, PB7), please use `GPIOB->ODR |= 0xC0;`
2. To reset PB6, you can used `GPIOB->ODR &= 0xFFBF;`
3. To reset PB7, you can used `GPIOB->ODR &= 0xFF7F;`
4. To generate 1 sys clk long pulse on PD0 use `sev` (assembly code)

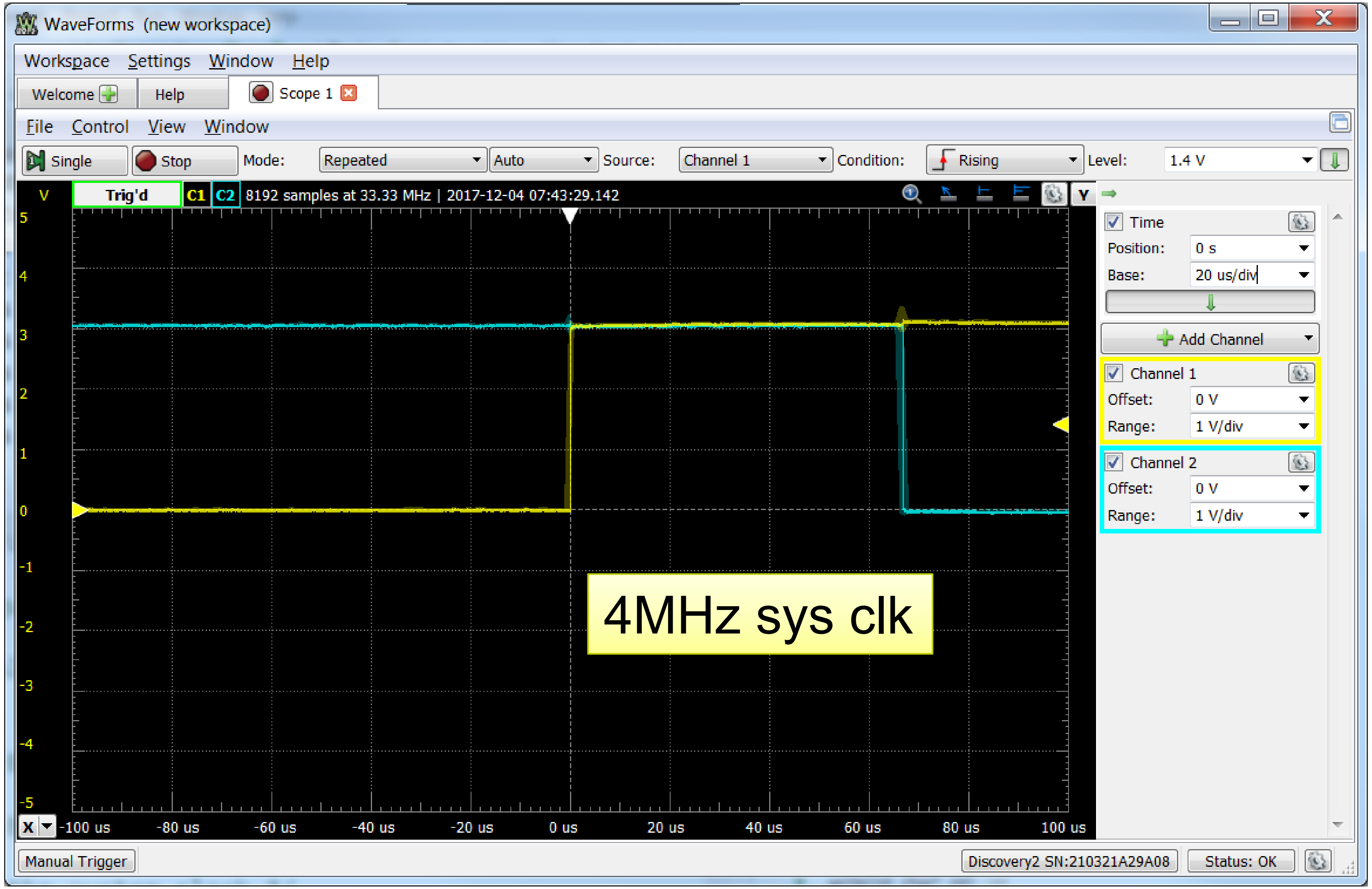
Put above lines in various places in the code to measure time intervals (on the next slide instruction 1) has been placed within `SysTick_Handler()` in **stm32L4xx_it.c**, instruction 2 and 3 in empty for(;;) loop within Task1 and Task2 accordingly (**main.c** file). Instruction 4 has been placed within `xPortPendSVHandler()` function (**port.c** file) just before its jump to user task (line BX LR).

In case of issues with GPIOB declaration, please include stm32l4xx.h file

gcc

Context switching time

Time between beginning of SysTick and user task code **~65us**

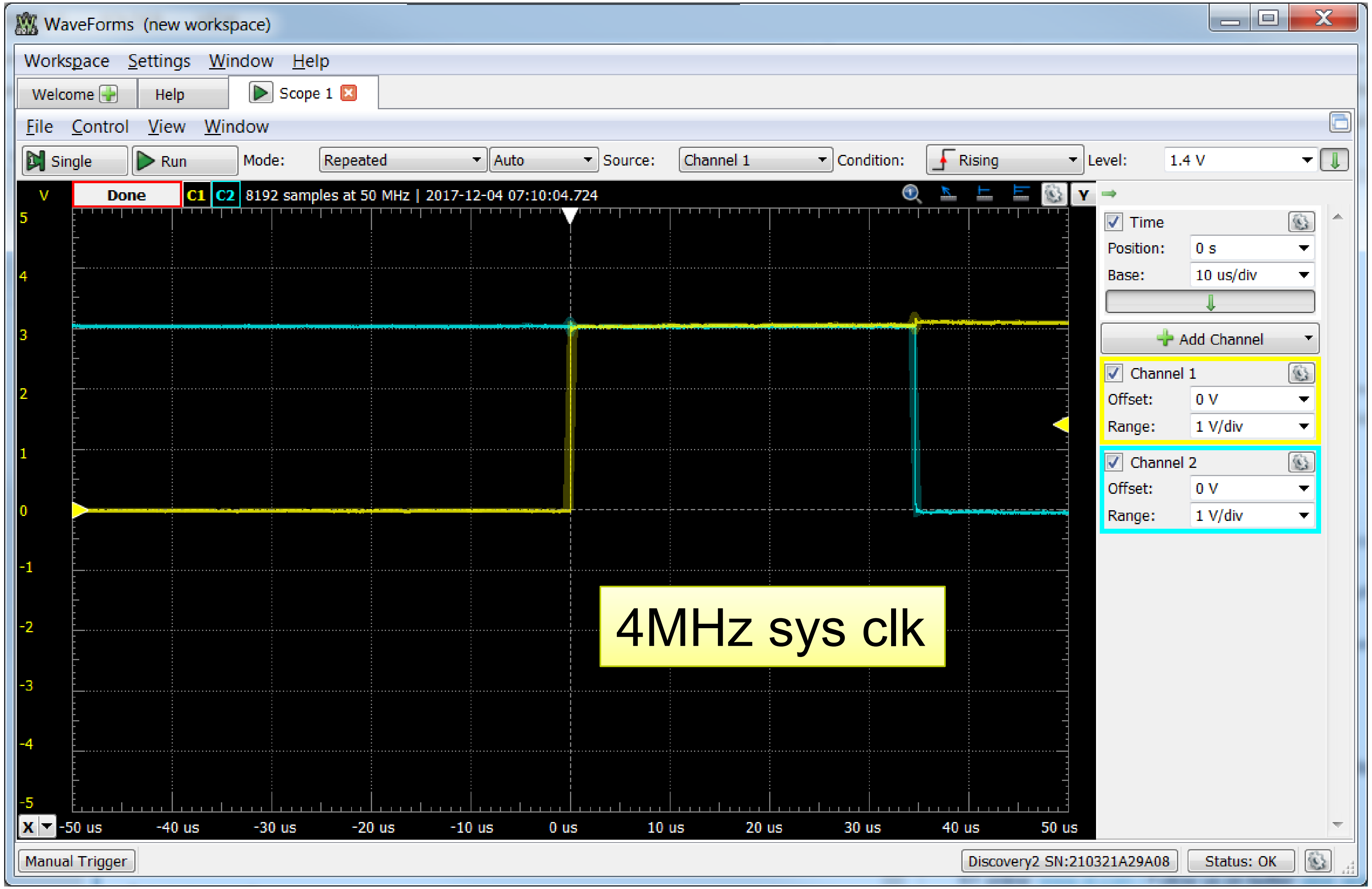


life.augmented

gcc

Context switching time

Time between beginning of beginning of PendSV code and user task code **~37us**



life.augmented

gcc

Context switching time

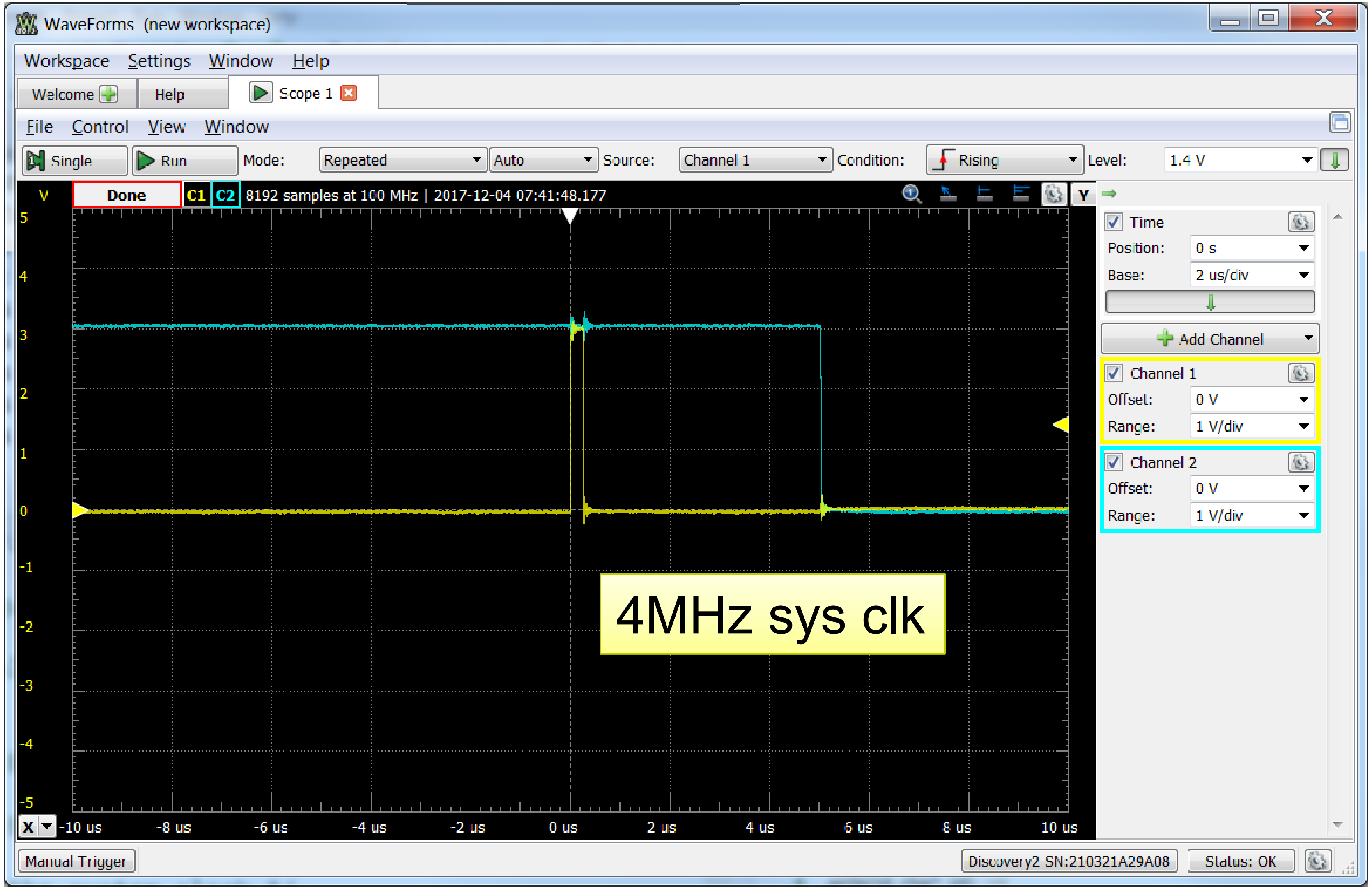
Time between beginning of SysTick and jump to user task within PendSV **~30us**

life.augmented

gcc

Context switching time

Time between jump to user task within PendSV and user task code **~5us**



life.augmented

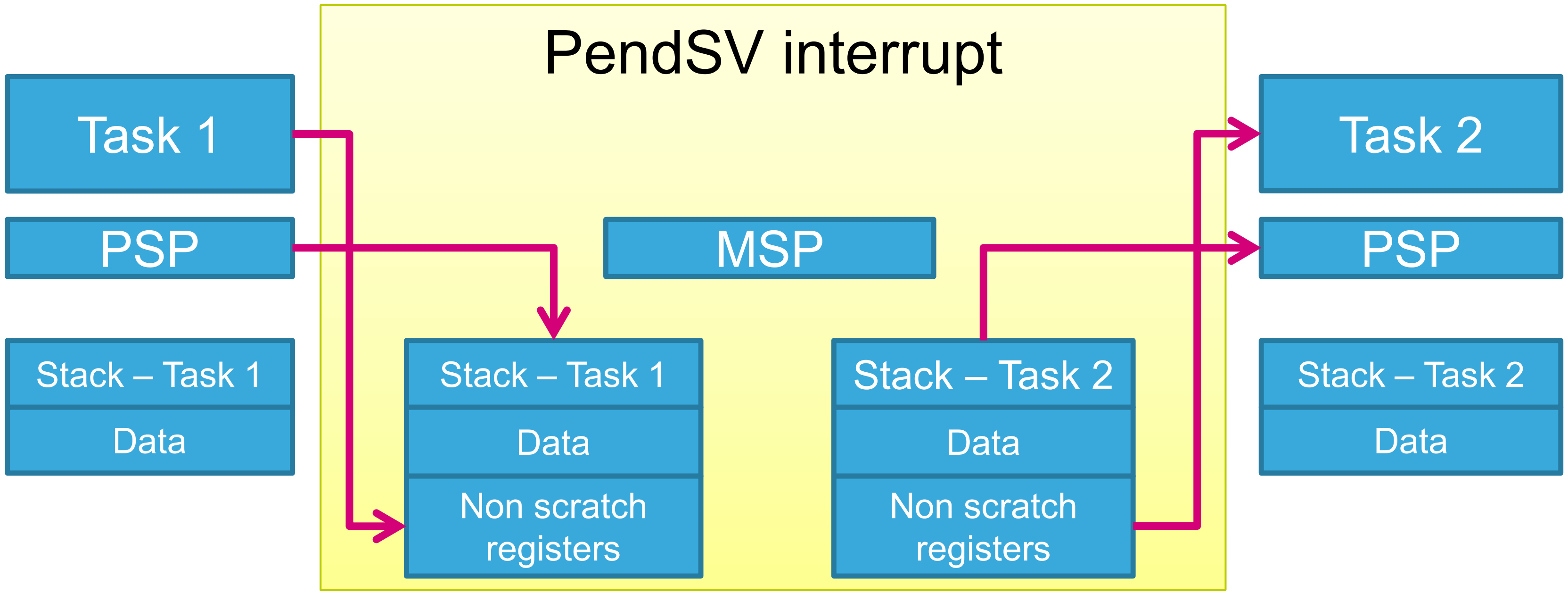
gcc

Context switching time

Length of the pulse generated by __sev() **~250ns** (1clk cycle @4MHz sys clk)

Stack pointers

- Main stack pointer (**MSP**)
  - Used in interrupts
  - Allocated by linker during compiling
- Process stack pointer (**PSP**)
  - Each task have own stack pointer
  - During context switch the stack pointer is initialized for correct task



Dual stack

- There are two independent stack pointers in CortexM devices:
  - Main Stack Pointer (MSP) – enabled by default.
  - Process Stack Pointer (PSP) – could be enabled (bit 1 in CONTROL register)
- Both 32bit registers are visible as R13 register of the Core and only one can be used at one time.
- Dual stack architecture is used for OS:
  - MSP – OS kernel and exception handlers
  - PSP – application tasks

Tasks API

- Create Task example

```c
/* Create the thread(s) */
 /* definition and creation of Task1 */
osThreadDef(Task1, StartTask1, osPriorityNormal, 0, 128);

Task1Handle = osThreadCreate(osThread(Task1), NULL);
```

Name used for handler

Name of the function

Priority of the task

Number of instances

Stack size in bytes

const osThreadDef_t **os_thread_def_Task1**

(void *argument) to be passed to task function

&os_thread_def_Task1

Return value:
- **Task1Handle = NULL** -> error (i.e. lack of heap memory to allocate the stack)
- **Task1Handle != NULL** -> task ID for reference by other functions

Tasks API

- Task handle definition:

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
```

- Create task

```
osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument)
```

- Delete task

```
osStatus osThreadTerminate (osThreadId thread_id)
```

- Get task ID

```
osThreadId osThreadGetId (void)
```

life.augmented

Tasks API

- Yield task

```
osStatus osThreadYield(void)
```

- Check if task is suspended

```
osStatus osThreadIsSuspended(osThreadId thread_id)
```

- Resume task

```
osStatus osThreadResume (osThreadId thread_id)
```

- Check state of task

```
osThreadState osThreadGetState(osThreadId thread_id)
```

- Suspend task

```
osStatus osThreadSuspend (osThreadId thread_id)
```

- Resume all tasks

```
osStatus osThreadResumeAll (void)
```

- Suspend all tasks

```
osStatus osThreadSuspendAll (void)
```

CMSIS-RTOS API

Threads (Tasks) priorities - osPriority

code	Value	description
osPriorityIdle	-3	idle (lowest)
osPriorityLow	-2	low
osPriorityBelowNormal	-1	Below normal
osPriorityNormal	0	Normal (default)
osPriorityAboveNormal	+1	Above normal
osPriorityHigh	+2	high
osPriorityRealtime	+3	Realtime (highest)
osPriorityError	0x84	system cannot determine priority or thread has illegal priority

Too high priority (above **configMAX_PRIORITIES** within FreeRTOSConfig.h) will be set to max configured value **configMAX_PRIORITIES**

CMSIS_OS API

return values osStatus 1/2

- Most of the functions returns `osStatus` value, below you can find return values on function completed list (**cmsis_os.h** file)

osStatus	value	description
`osOK`	`0`	no error or event occurred
`osEventSignal`	`8`	signal event occurred
`osEventMessage`	`0x10`	message event occurred
`osEventMail`	`0x20`	mail event occurred
`osEventTimeout`	`0x40`	timeout occurred
`os_status_reserved`	`0x7FFFFFFF`	prevent from enum down-size compiler optimization

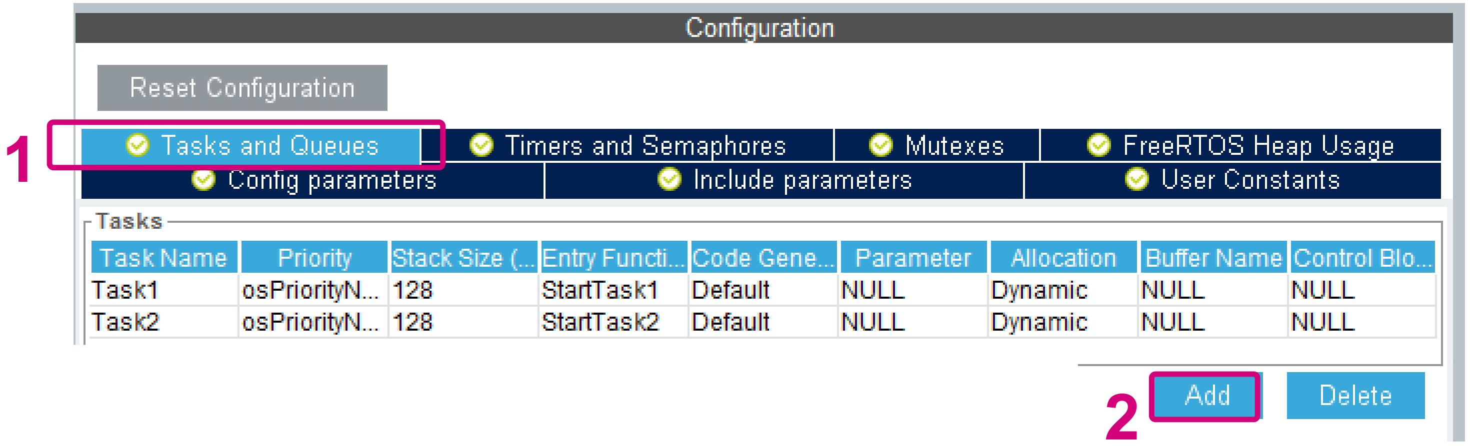
CMSIS_OS API

return values osStatus 2/2

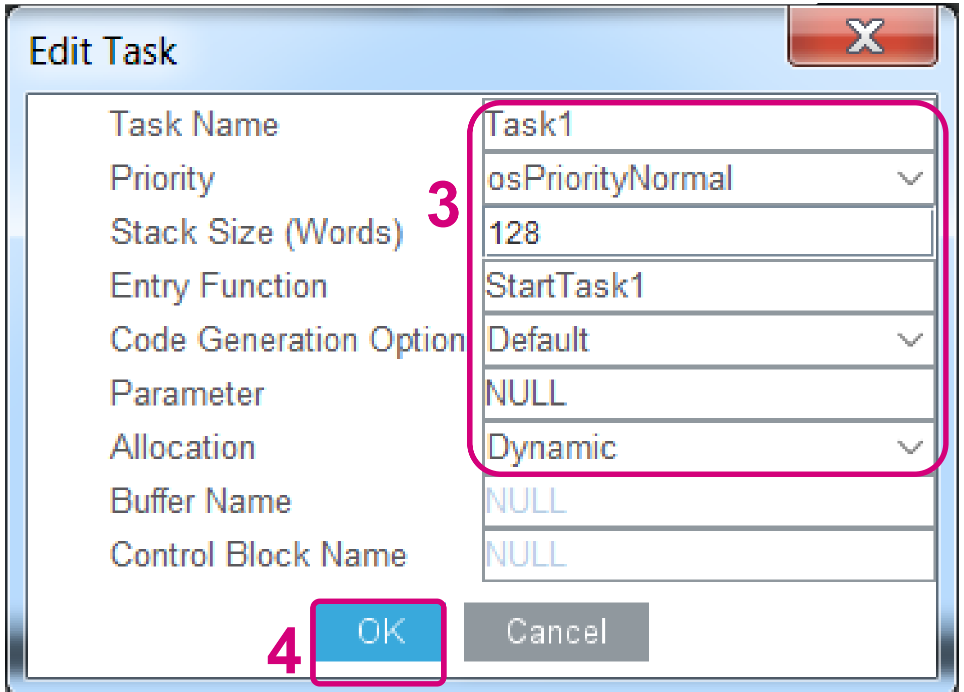
- Error status values `osStatus` (**cmsis_os.h**)

osStatus value	description
`osErrorParameter` `0x80`	parameter error: a mandatory parameter was missing or specified an incorrect object.
`osErrorResource` `0x81`	resource not available: a specified resource was not available
`osErrorTimeoutResource` `0xC1`	resource not available within given time: a specified resource was not available within the timeout period.
`osErrorISR` `0x82`	not allowed in ISR context: the function cannot be called from interrupt service routines
`osErrorISRRecursive` `0x83`	function called multiple times from ISR with same object.
`osErrorPriority` `0x84`	system cannot determine priority or thread has illegal priority
`osErrorNoMemory` `0x85`	system is out of memory: it was impossible to allocate or reserve memory for the operation
`osErrorValue` `0x86`	value of a parameter is out of range.
`osErrorOS` `0xFF`	unspecified RTOS error: run-time error but no other error message fits.

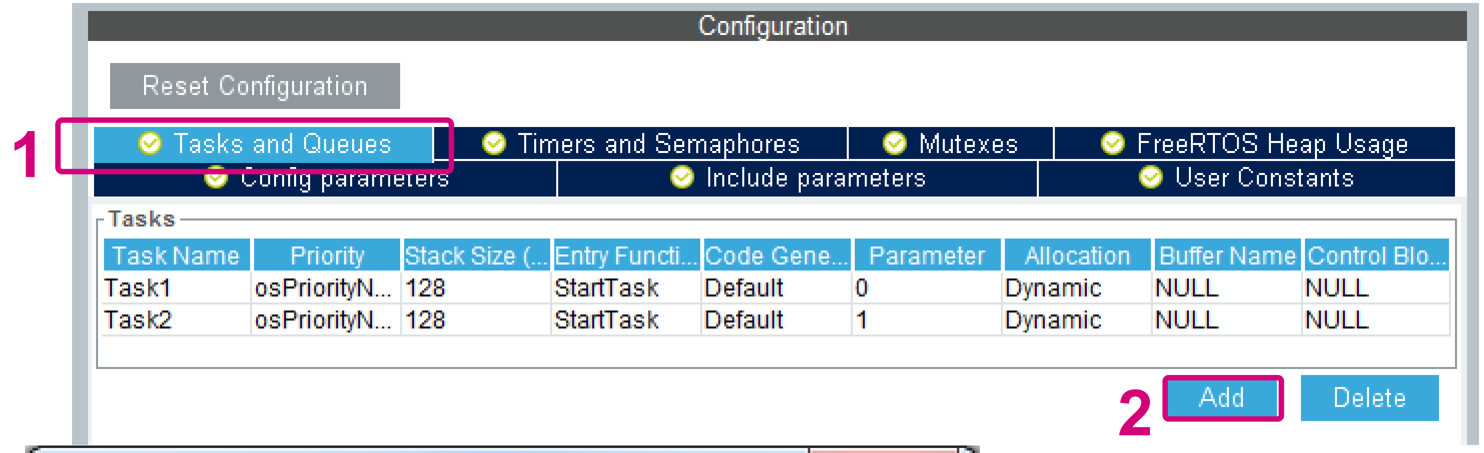
Press **FreeRTOS** button within Pinout&Configuration tab

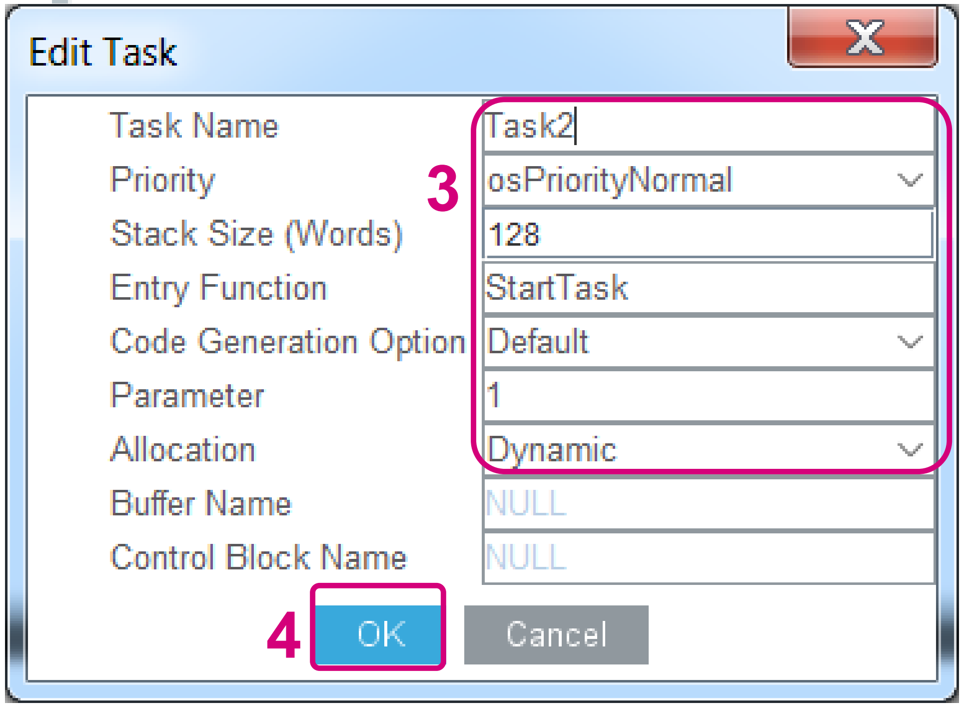


- We need to create 2 tasks:
  - **Task1**:
    - Priority: osPriorityNormal
    - Stack Size: 128 Words
    - Entry Function: StartTask1
    - Code Generation: Default
    - Parameter: NULL
    - Allocation: Dynamic
  - **Task2**:
    - Priority: osPriorityNormal
    - Stack Size: 128 Words
    - Entry Function: StartTask2
    - Code Generation: Default
    - Parameter: NULL
    - Allocation: Dynamic



Press **FreeRTOS** button within Pinout&Configuration tab

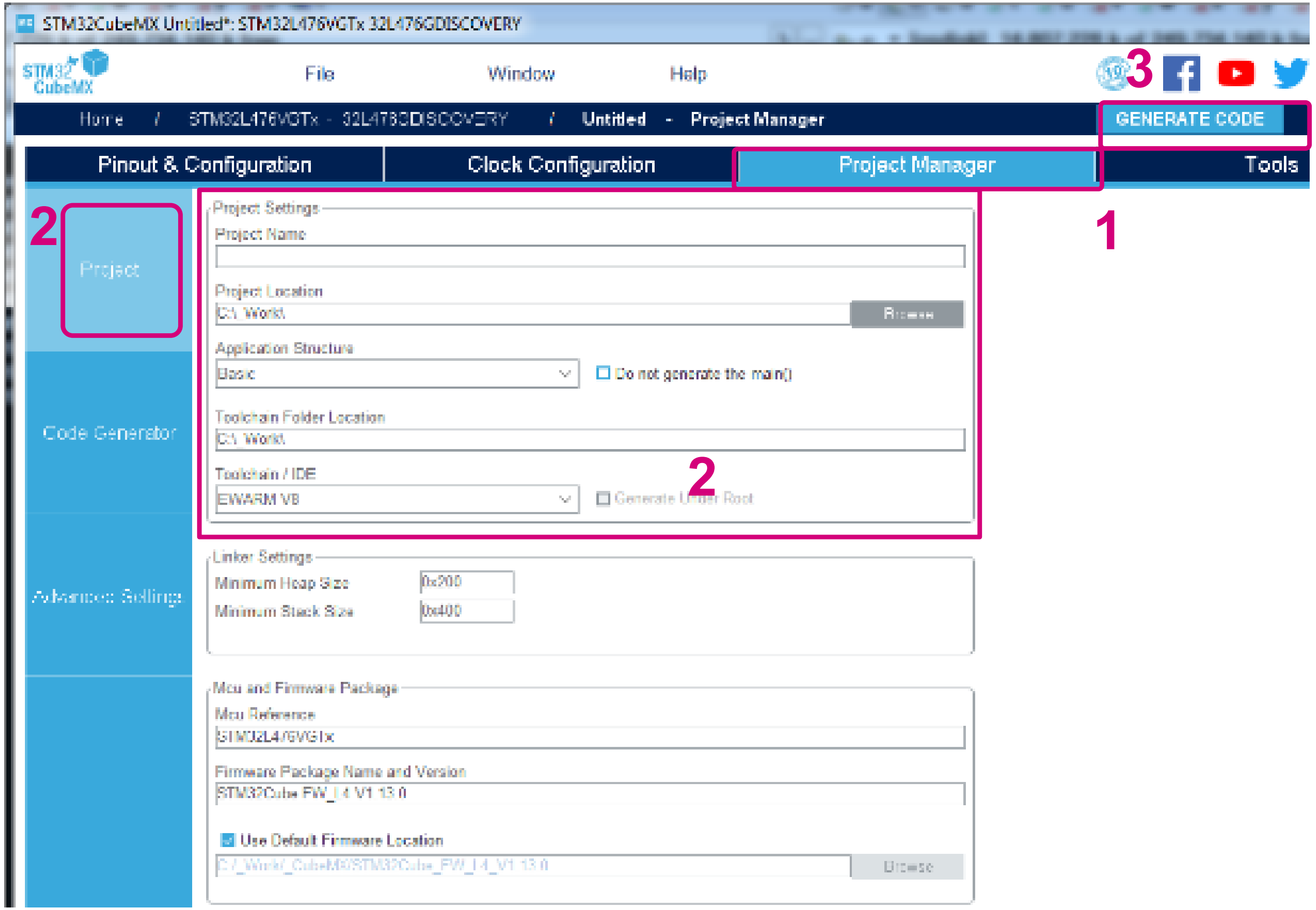




- We need to create 2 tasks:
  - **Task1**:
    - Priority: osPriorityNormal
    - Stack Size: 128 Words
    - Entry Function: StartTask
    - Code Generation: Default
    - Parameter: 0
    - Allocation: Dynamic
  - **Task2**:
    - Priority: osPriorityNormal
    - Stack Size: 128 Words
    - Entry Function: StartTask
    - Code Generation: Default
    - Parameter: 1
    - Allocation: Dynamic

Tasks lab
code generation

- To configure the project
  1. Select Project Manager tab
  2. Within Project tab select:
     - project name
     - Project location
     - Type of toolchain
- To Generate Code
  3. Select Generate Code button



life.augmented

Tasks lab

analysis of the code generated by STM32CubeMX

- Any component in FreeRTOS need to have handle, very similar to STM32CubeMX

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
osThreadId Task2Handle;
```

- Task function prototypes, names was taken from STM32CubeMX

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
void StartTask1(void const * argument);
void StartTask2(void const * argument);
```

- Before the scheduler is start we must create tasks

```
/* Create the thread(s) */
 /* definition and creation of Task1 */
 osThreadDef(Task1, StartTask1, osPriorityNormal, 0, 128);
 Task1Handle = osThreadCreate(osThread(Task1), NULL);

 /* definition and creation of Task2 */
 osThreadDef(Task2, StartTask2, osPriorityNormal, 0, 128);
 Task2Handle = osThreadCreate(osThread(Task2), NULL);
```

Define task parameters

Create task, allocate memory

life.augmented

printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

life.augmented

Tasks lab

some code modifications

- Start the scheduler. Its function should never ends *)

```
/* Start scheduler */
 osKernelStart();
```

- On first task run StartTask1 is called
- Task must have inside infinite loop in case we don't want to end the task

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task 1\n");
    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

Endless loop

osDelay will start context switch

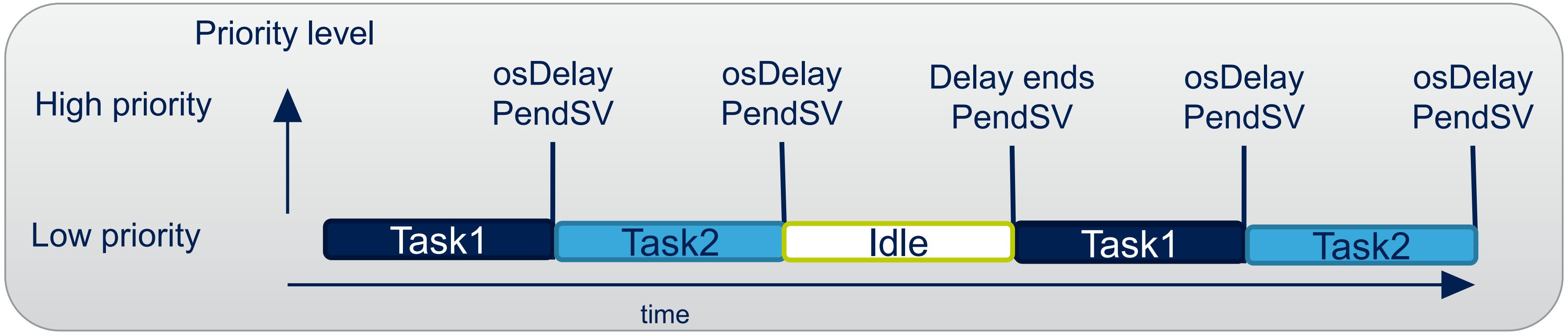
- Similar code prepare for Task2 function
- You can monitor both tasks output in debug (printf) viewer from the first lab
- Modify the code for both tasks in order to display a number of the task call, like: "Task2. Call no 12"

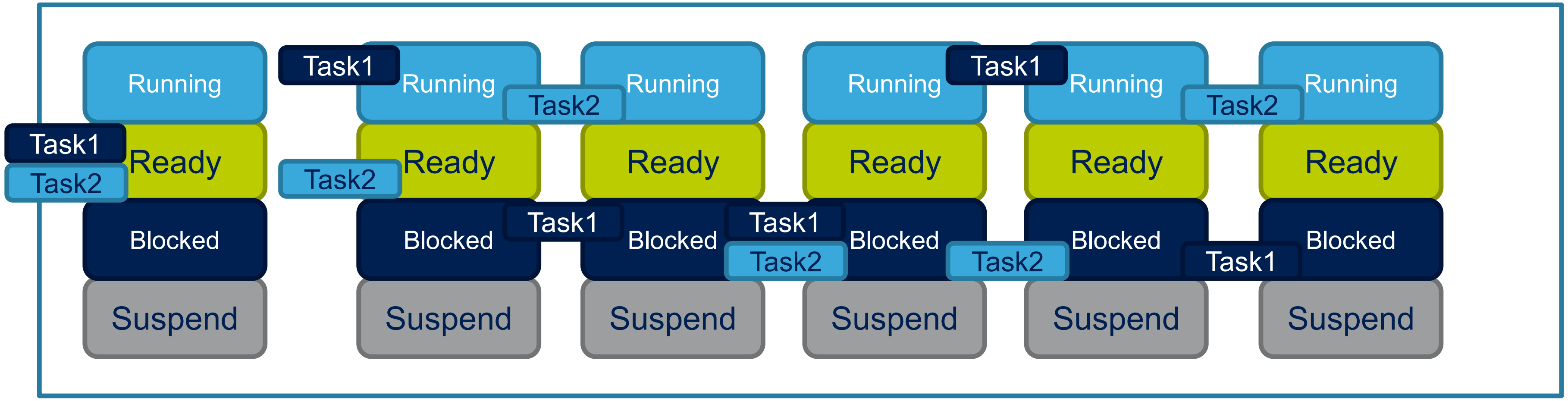
life.augmented

*) if it ends, it means that we are out of declared heap size and there was not enough memory space to create a new task

Tasks lab

- If both Delays are processed the FreeRTOS is in idle state

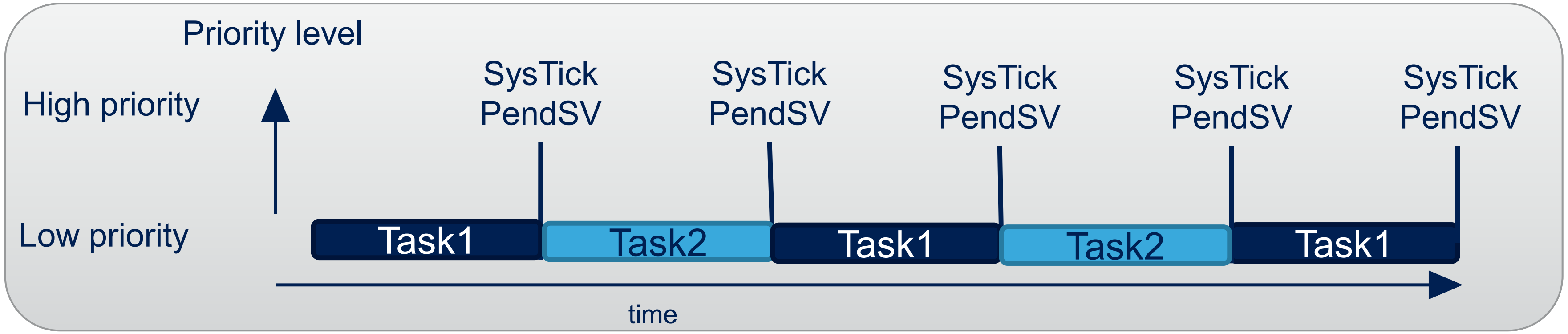


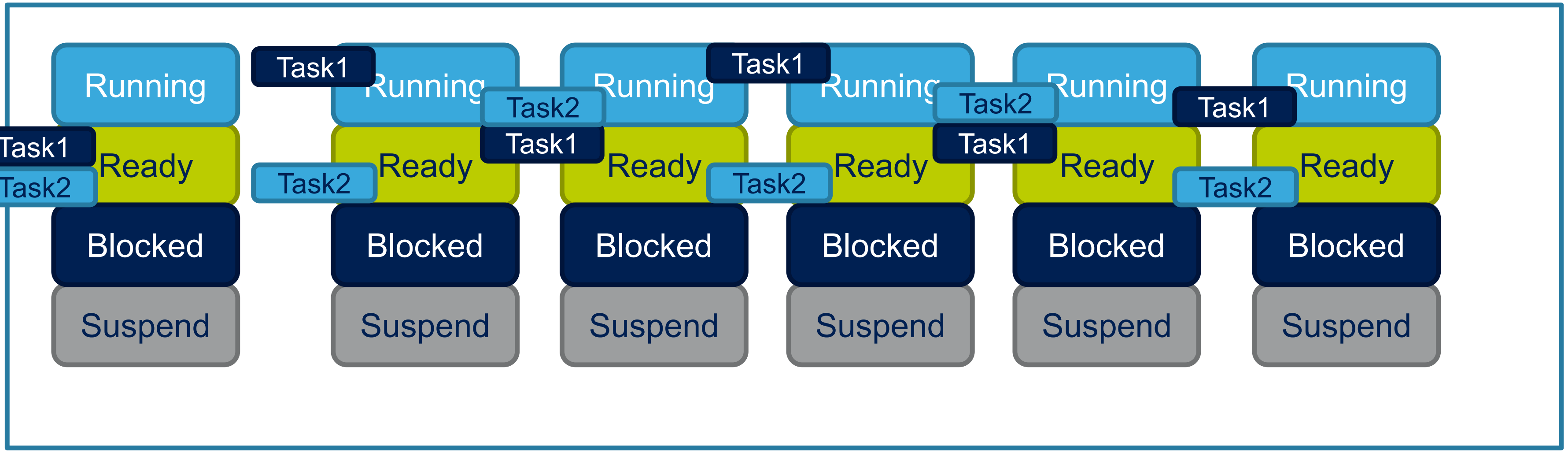


life.augmented

Tasks lab

- Without Delays the threads will be in Running state or in Ready state
- Use **`HAL_Delay()`**





Tasks lab

task priorities

- Increase the priority of Task1
- Double click on task for change
- Button OK
- Regenerate the code and compile it
- Is there any difference in the printf window during debug?
- What could be done to see the difference (Task1 more frequent occurrence)

Tasks and Queues | Timers and Semaphores | Mutexes | FreeRTOS Heap Usage
Config parameters | Include parameters | User Constants

Tasks

Task Name	Priority	Stack Size (Wor...	Entry Function	Code Generatio...	Parameter	Allocation	Buffer Name	Control Block N...
Task1	osPriorityRealti...	128	StartTask1	Default	NULL	Dynamic	NULL	NULL
Task2	osPriorityNormal	128	StartTask2	Default	NULL	Dynamic	NULL	NULL

Tasks lab

task priorities

- After we 5x times send text put task to block state
- Because task have high priority it allow to run lower priority task

```
/* USER CODE END 4 */
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  uint32_t i = 0;
  /* Infinite loop */
  for(;;)
  {
    for (i = 0; i < 5; i++){
      printf("Task 1\n");
      HAL_Delay(50);
    }
    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

Helps not spam terminal

Block task

Tasks lab

task priorities

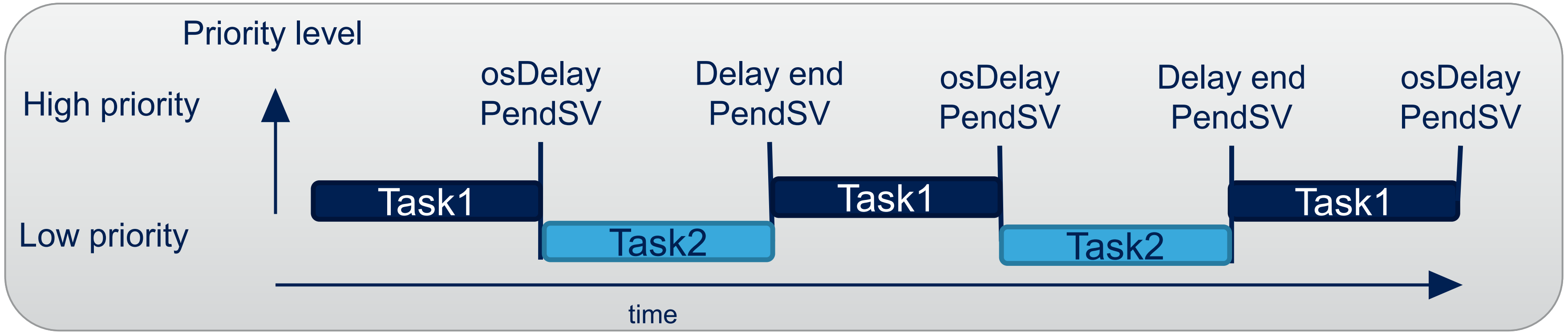
- If higher priority task is not running we can print text from this task

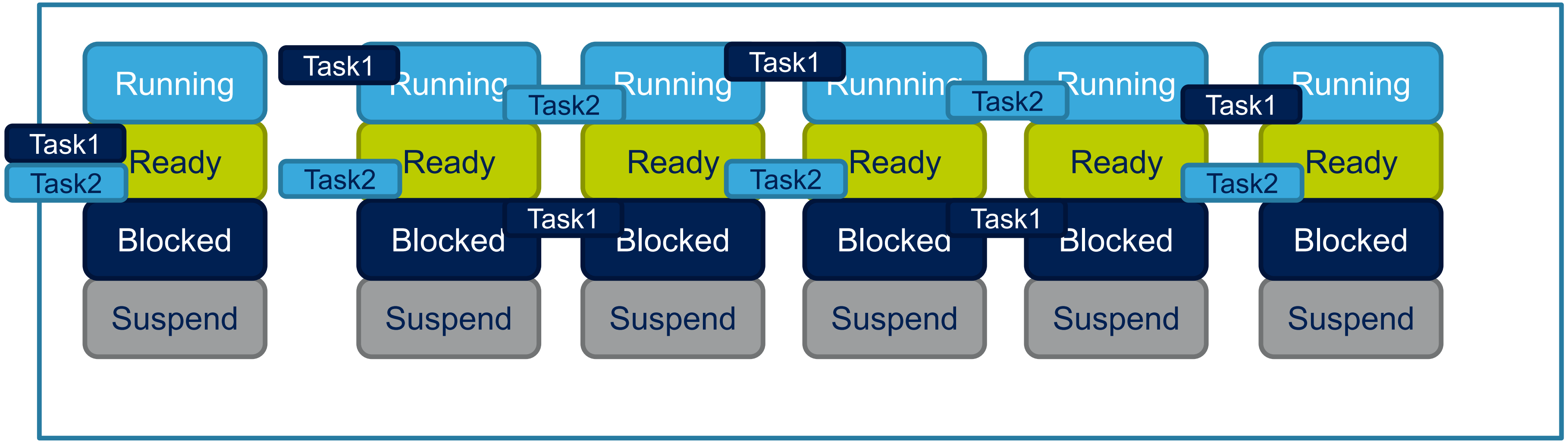
```
/* StartTask2 function */
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task 2\n");
    HAL_Delay(50);
  }
  /* USER CODE END StartTask2 */
}
```

Helps not spam terminal

Tasks lab

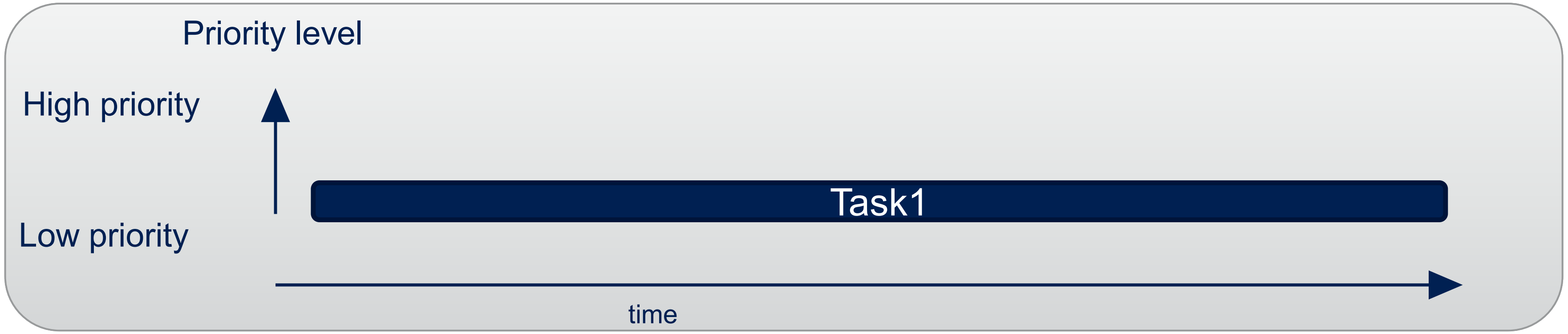
- What happen if Task1 not call `osDelay()` ?

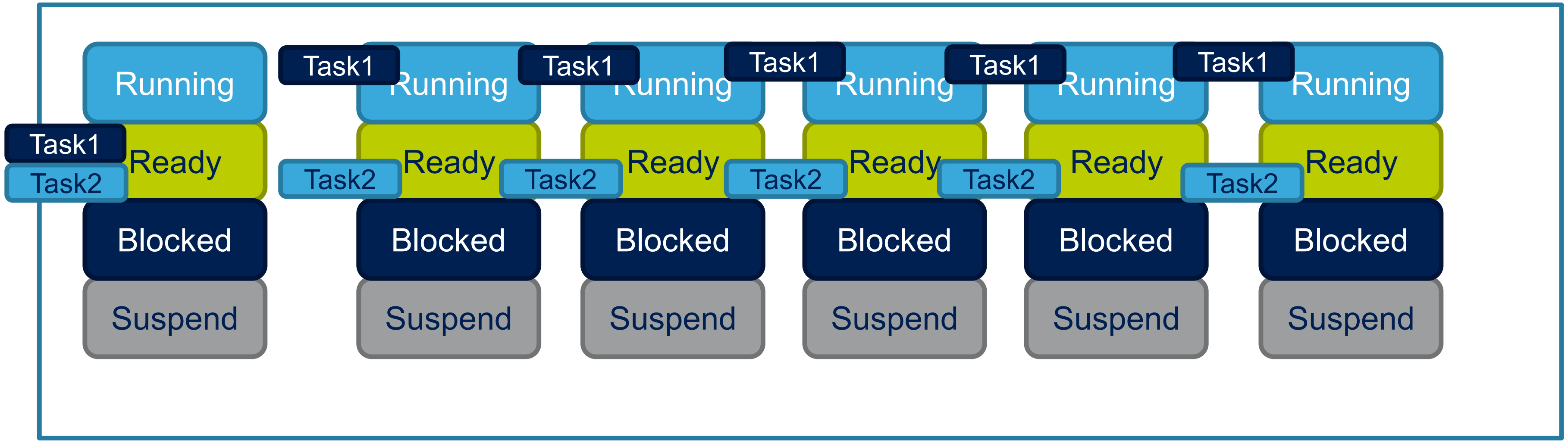




Tasks lab

- Task1 will be executed continuously





Include **vTaskDelayUntil**

Include **vTaskDelay**

osDelay API

- Delay function

```
osStatus osDelay (uint32_t millisec)
```

- Delay function which measure time from which is delay measured

```
osStatus osDelayUntil (uint32_t PreviousWakeTime, uint32_t millisec)
```

life.augmented

Include **vTaskDelay**

osDelay API

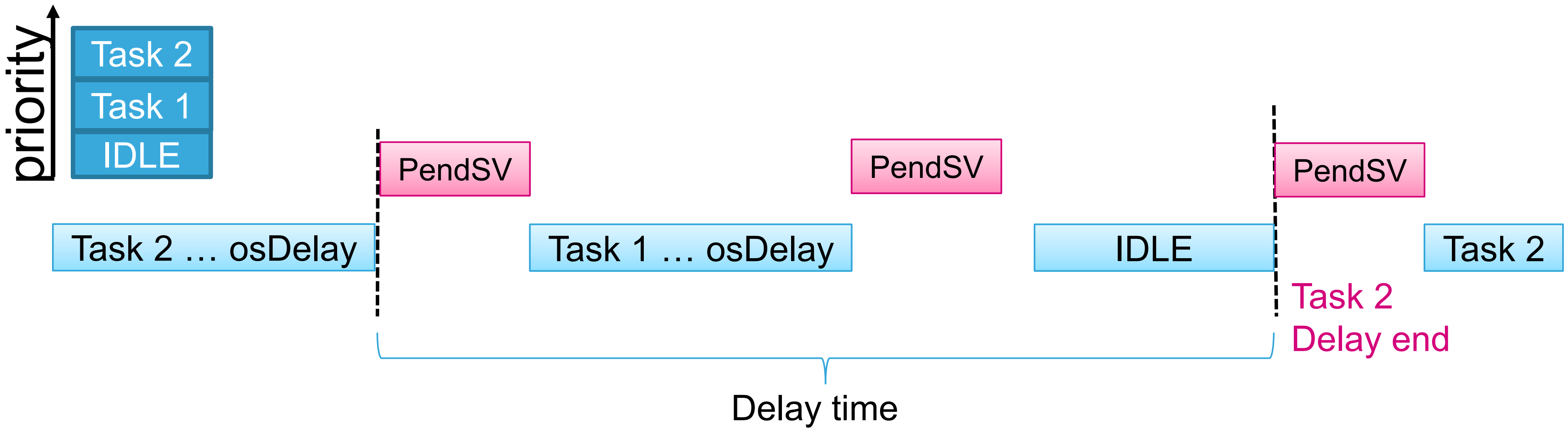
step by step

- `osDelay()` calls `vTaskDelay()` (**tasks.c** file)
- `vTaskDelay()` is performing the following list of operations:
  - Calls `vTaskSuspendAll()` to pause the scheduler without disabling interrupts. RTOS tick will be held pending until the scheduler has been resumed.
  - Remove task from event list (running tasks) and move it to delayed list with given delay value using the function `prvAddCurrentTaskToDelayedList()`
  - Resume the scheduler using `xTaskResumeAll()` function
  - Trigger PendSV interrupt (using `portYIELD_WITHIN_API()` macro) to switch the context

Include **vTaskDelay**

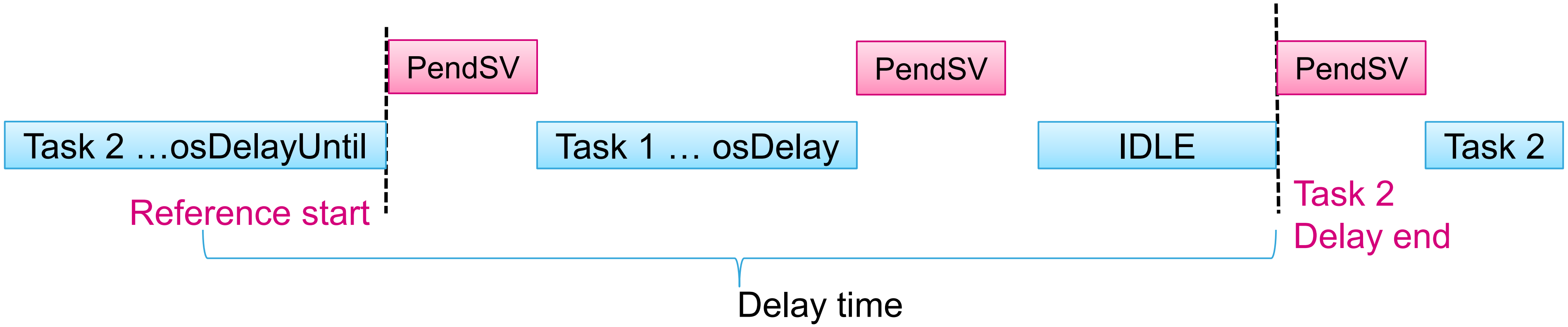
osDelay(), osDelayUntil functions

- `osDelay()` start measure time from osDelay call



Include **vTaskDelayUntil**

- `osDelayUntil()` starts measure time from point which we selected



Include **vTaskDelayUntil**

Include **vTaskDelay**

osDelay() and osDelayUntil()

- Enable **vTaskDelayUntil** in Include parameters
- Regenerate project, modify tasks to:

```
void StartTask1(void const * argument)
{

  /* USER CODE BEGIN 5 */
  uint32_t i = 0;
  /* Infinite loop */
  for(;;)
  {
    printf("Task 1\n");
    HAL_Delay(1000);
    osDelay(2000);
  }
  /* USER CODE END 5 */
}

/* StartTask2 function */
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task 2\n");
    HAL_Delay(200);
  }
  /* USER CODE END StartTask2 */
}
```

Delay between two run is 2s

- Mutexes
- FreeRTOS Heap Usage
- Tasks and Queues
- Timers and Semaphores
- Config parameters
- Include parameters
- User Constants

Configure the following parameters:

Search (Crtl+F)

Include definitions

vTaskPrioritySet	Enabled
uxTaskPriorityGet	Enabled
vTaskDelete	Enabled
vTaskCleanUpResources	Disabled
vTaskSuspend	Enabled
vTaskDelayUntil	Enabled
vTaskDelay	Enabled
xTaskGetSchedulerState	Enabled
xTaskResumeFromISR	Enabled
xQueueGetMutexHolder	Disabled
xSemaphoreGetMutexHolder	Disabled
pcTaskGetTaskName	Enabled
uxTaskGetStackHighWaterMark	Enabled
xTaskGetCurrentTaskHandle	Enabled
eTaskGetState	Enabled
xEventGroupSetBitFromISR	Disabled
xTimerPendFunctionCall	Disabled
xTaskAbortDelay	Enabled

Include **vTaskDelayUntil**

osDelay() and osDelayUntil()

- Enable **vTaskDelayUntil** in Include parameters
- Regenerate project, modify tasks to:

```
void StartTask1(void const * argument)
{

  /* USER CODE BEGIN 5 */
  uint32_t wakeuptime;
  wakeuptime=osKernelSysTick();
  /* Infinite loop */
  for(;;)
  {
    printf("Task 1\n");
    HAL_Delay(1000);
    osDelayUntil(wakeuptime,2000);
  }
  /* USER CODE END 5 */

}
```

For osDelayUntil function we need mark wakeup time

Function will be executed every 2s

Time from which the delay is measured

Real delay time

life.augmented

Include **uxTaskPriorityGet**

Thread priority get API

step by step

- `osThreadGetPriority()` calls `uxTaskPriorityGet()` or `uxTaskPriorityGetFromISR()` (**tasks.c** file)

- `uxTaskPriorityGet()` is performing the following list of operations:
  - Entering into critical section (to avoid any parallel operations on OS) using `taskENTER_CRITICAL()` in case of executing from thread mode or `portSET_INTERRUPT_MASK_FROM_ISR()` in case of interrupt mode
  - Read priority value from TCB of the given task using function `prvGetTCBFromHandle(TCB_t xTask)`
  - extract Priority value from the TCB structure (`uxPriority` field)
  - Exit from critical section using `taskEXIT_CRITICAL()` in case of executing from thread mode or `portCLEAR_INTERRUPT_MASK_FROM_ISR()` in case of interrupt mode

life.augmented

Include **vTaskPrioritySet**

Thread priority set API

step by step

- `osThreadSetPriority()` calls `vTaskPrioritySet()` (**tasks.c** file)
- `vTaskPrioritySet()` is performing the following list of operations:
  - Entering into critical section (to avoid any parallel operations on OS) using `taskENTER_CRITICAL()`
  - Set given priority value to TCB of the given task
  - Checks whether task should not be moved to different task list due to new priority
  - Exit from critical section using `taskEXIT_CRITICAL()`

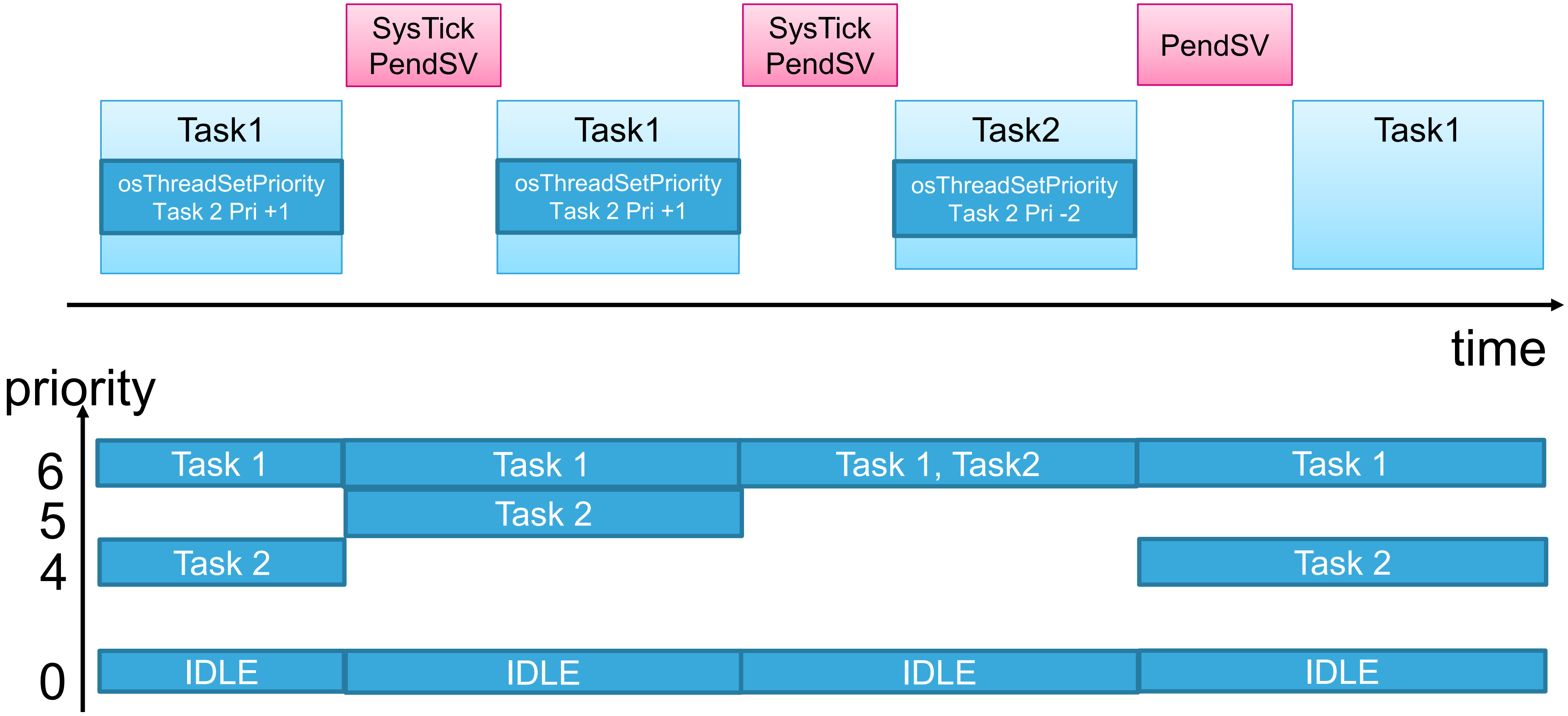
life.augmented

Include **vTaskPrioritySet**

Include **uxTaskPriorityGet**

Priority change lab

- How priorities are changed?

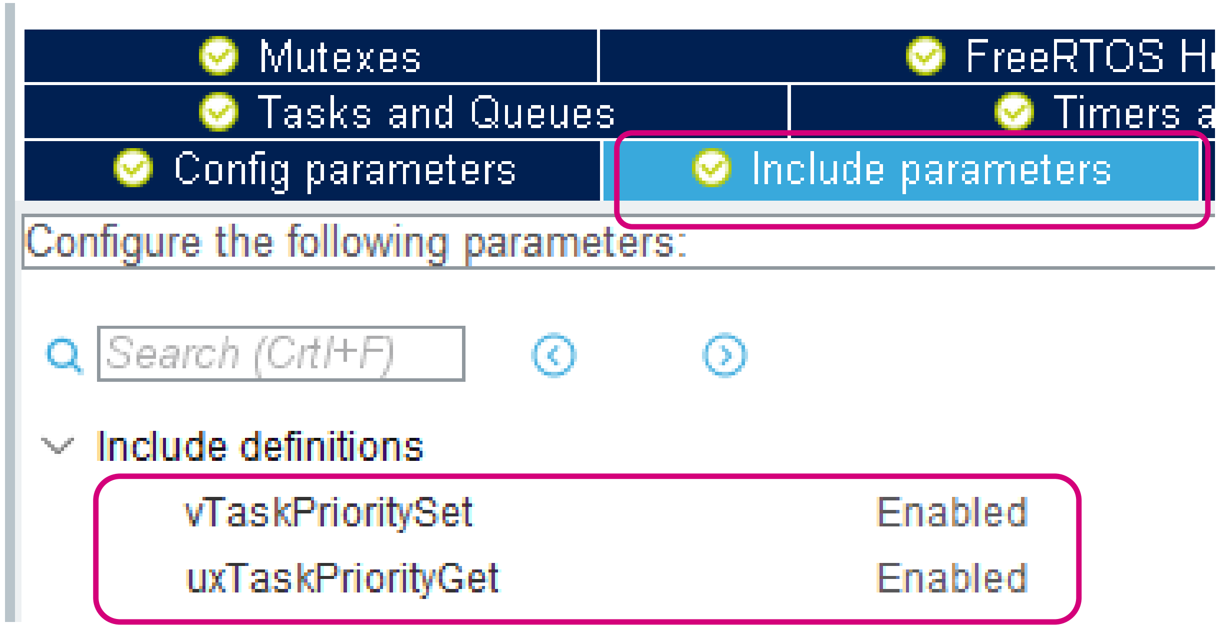


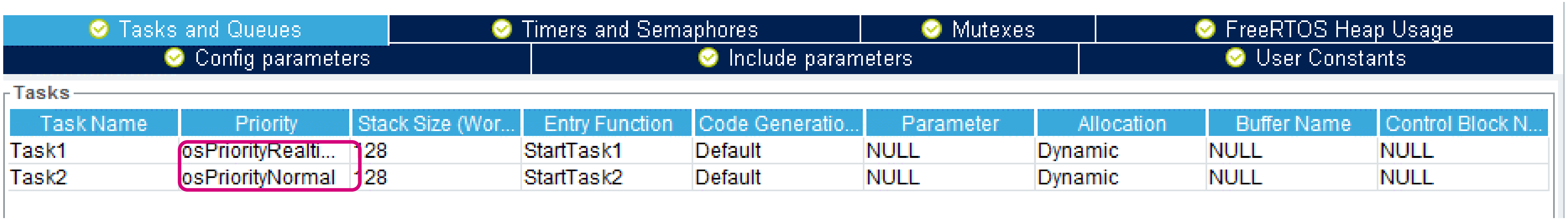
Include **vTaskPrioritySet**

Include **uxTaskPriorityGet**

Priority change lab

- Task1 has higher priority than Task2
- If not yet done, enable **vTaskPriorityGet** and **uxTaskPrioritySet** in IncludeParameters





Include **vTaskPrioritySet**

Include **uxTaskPriorityGet**

Priority change lab

- Modify Task1 to:

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  osPriority priority;
  /* Infinite loop */
  for(;;)
  {
    priority=osThreadGetPriority(Task2Handle);
    printf("Task 1\n");
    osThreadSetPriority(Task2Handle,priority+1);
    HAL_Delay(1000);
  }
  /* USER CODE END 5 */
}
```

Read Task2 priority

Increase Task2 priority

life.augmented

Include **vTaskPrioritySet**

Include **uxTaskPriorityGet**

Priority change lab

- Modify Task2 to:

```
/* StartTask2 function */
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  osPriority priority;
  /* Infinite loop */
  for(;;)
  {
    priority=osThreadGetPriority(NULL);
    printf("Task 2\n");
    osThreadSetPriority(NULL,priority-2);
  }
  /* USER CODE END StartTask2 */
}
```

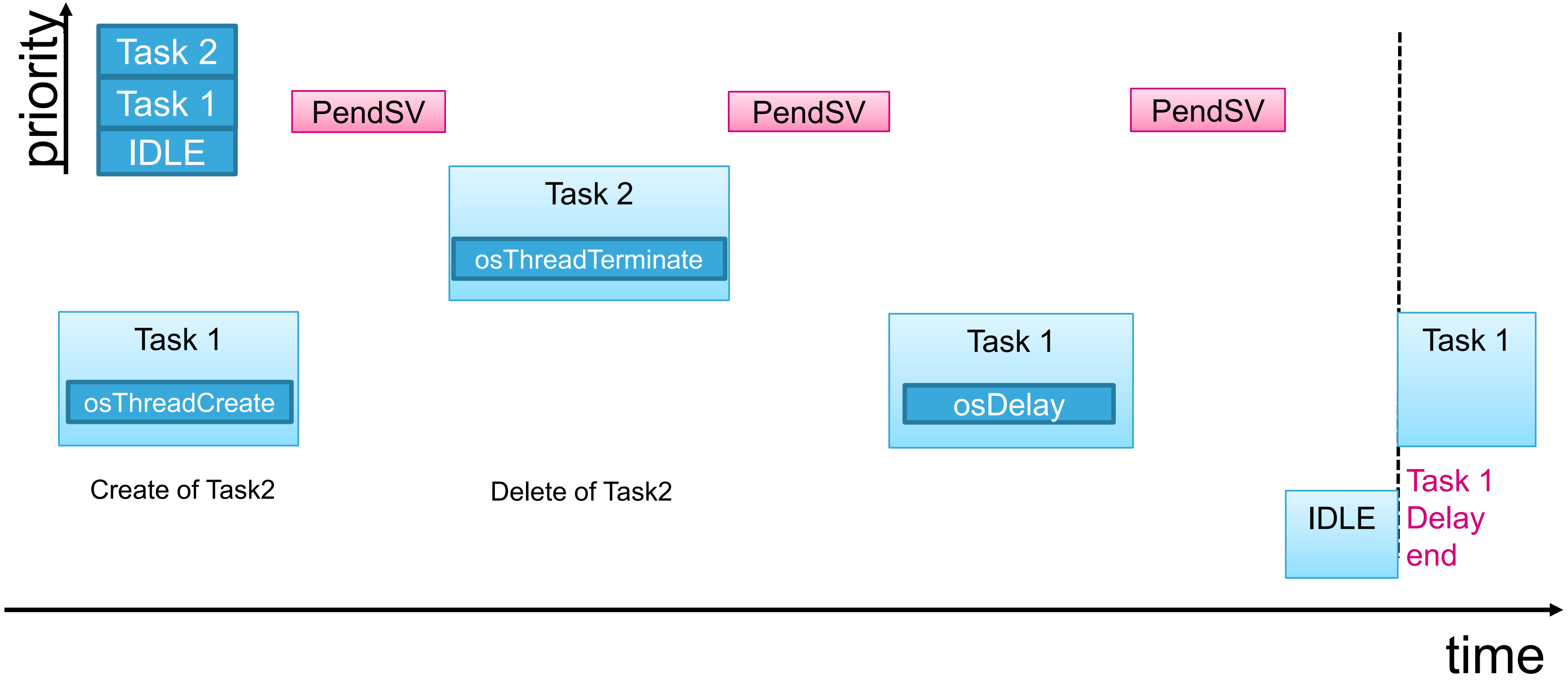
Read priority of current task

Decrease task priority

Include **vTaskDelete**

Creating and deleting tasks lab

- Example how to create and delete tasks



Creating and deleting tasks lab

- Example how to create tasks
- Comment Task2 creation part in **main.c**

```
 /* definition and creation of Task2 */
//  osThreadDef(Task2, StartTask2, osPriorityNormal, 0, 128);
//  Task2Handle = osThreadCreate(osThread(Task2), NULL);
```

- Modify Task1 to create task2

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    printf("Create task2");
    osThreadDef(Task2, StartTask2, osPriorityNormal, 0, 128);
    Task2Handle = osThreadCreate(osThread(Task2), NULL);
    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

Task 2 creation

life.augmented

Include **vTaskDelete**

Creating and deleting tasks lab

- Example how to delete tasks
- Modify Task2 to delete himself:

```
/* StartTask2 function */
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
    printf("Delete Task2\n");
    osThreadTerminate(Task2Handle);
  }
  /* USER CODE END StartTask2 */
}
```

Delete Task

life.augmented

Include **vTaskDelete**

osThreadTerminate API

step by step

- `osThreadTerminate()` calls `vTaskDelete()` (**cmsis_os.c** file)

- The only argument specifies the ID of the task to be deleted. NULL means that the calling task will be deleted.

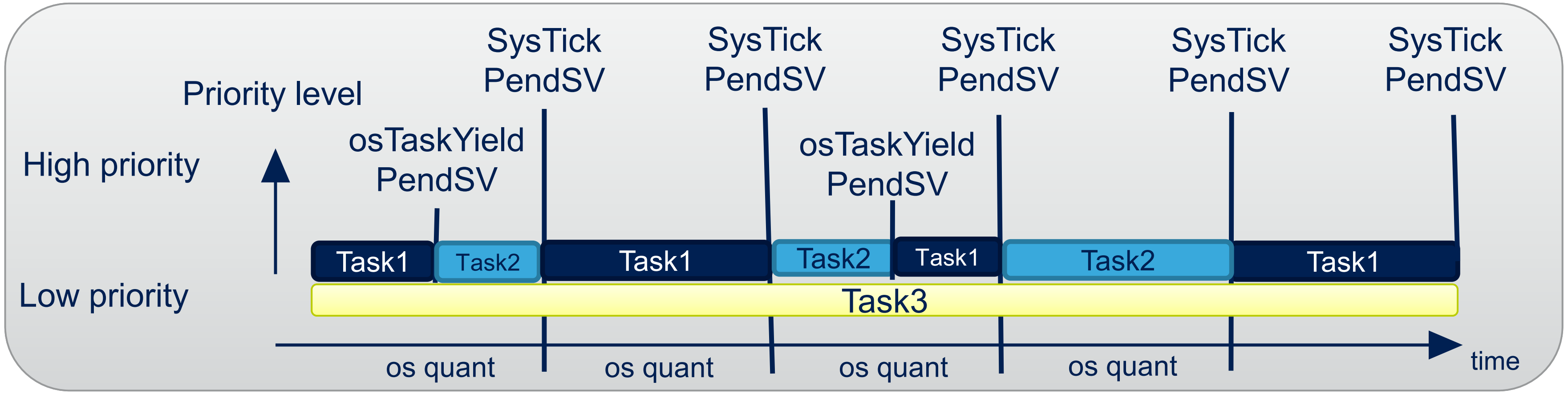
- `vTaskDelete()` function (**task.c** file):
  - Within critical section (started by `taskENTER_CRITICAL()` macro which is running `vPortEnterCritical()` defined in **port.c** file) removes the task from the ready list using function `uxListRemove()` and removes the task from waiting on an event tasks list.
  - In case the task is deleting itself function is switching execution to the next task calling function `portYIELD_WITHIN_API()` which could be in fact `portYIELD()` function (default setting, **FreeRTOS.h** file)

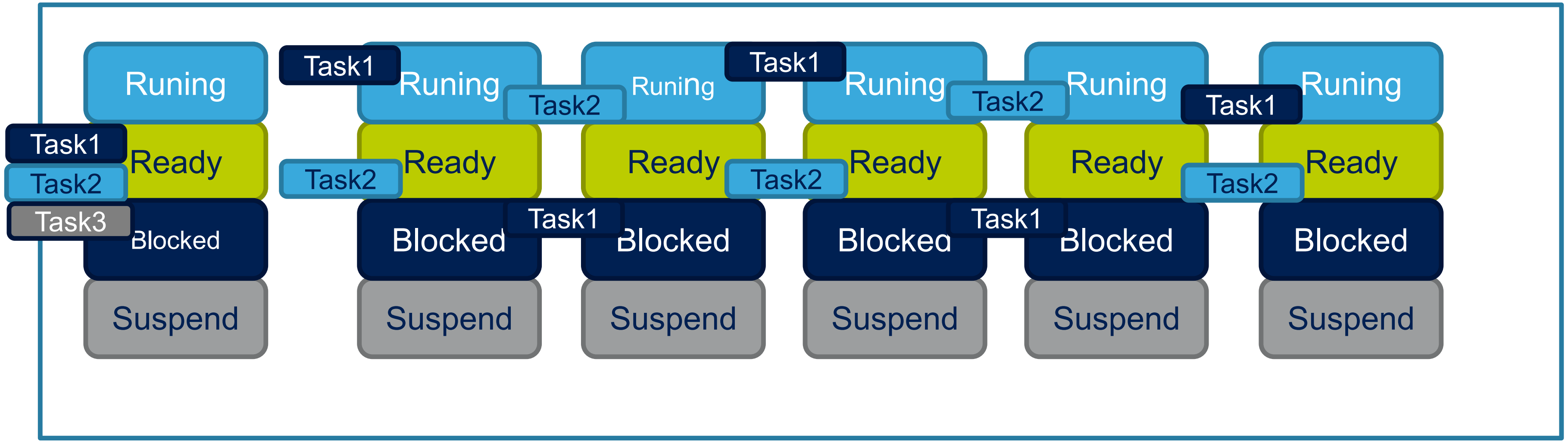
Memory allocated by the task code is not automatically freed and should be freed before the task is deleted, TCB and its original stack are freed by IDLE Task.

life.augmented

If the task has finished its job earlier…

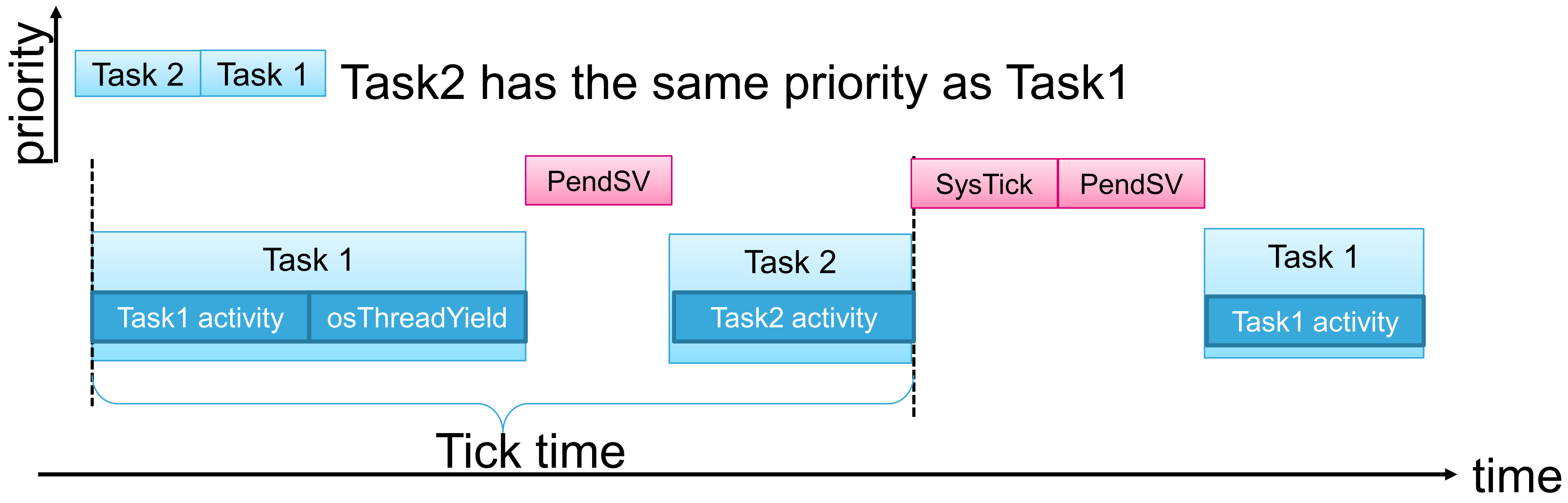
- **osThreadYield()** – move the task from Run to Ready state. Next task with the same priority will be executed.





osThreadYield() function

- **`osThreadYield()`** function is used to end task activity once the job is done to not wait for the tick.
- It moves task from **RUN** mode to **READY**
- It makes sense if we have few tasks on the same priority otherwise yielded task will be executed again



osThreadYield API

step by step

- `osThreadYield()` calls `taskYIELD()` (**cmsis_os.c** file) which is defined as `portYIELD()` (**task.h** file)
- `portYIELD()` function (**portmacro.h** file) triggers PendSV interrupt to request a context switch to the next task from ready list

An example (version for IAR C compiler):

```
#define portYIELD()
{
 /* Set a PendSV to request a context switch. */
 portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
 __DSB();
 __ISB();
}
```

Threads/Tasks APIs

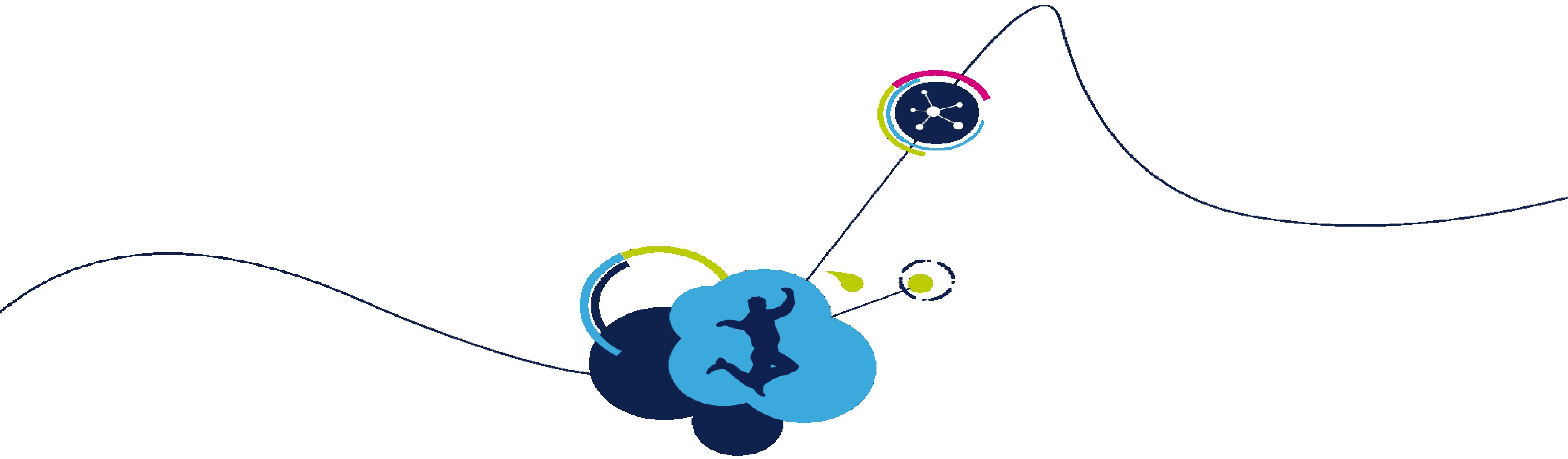
CMSIS_RTOS API	FreeRTOS API
osKernelInitialize() - empty	-
osKernelStart()	vTaskStartScheduler()
osKernelRunning()	xTaskGetSchedulerState()
osKernelSysTick()	xTaskGetTickCount() xTaskGetTickCountFromISR()
osThreadCreate()	xTaskCreate()
osThreadGetId()	xTaskGetCurrentTaskHandle()
osThreadTerminate()	vTaskDelete()
osThreadYield()	taskYIELD()
osThreadSetPriority()	vTaskPrioritySet()
osThreadGetPriority()	uxTaskPriorityGet() uxTaskPriorityGetFromISR()
osDelay()	vTaskDelay()

life.augmented

Threads/Tasks APIs

CMSIS_RTOS API	FreeRTOS API
osWait() – empty function	-
osThreadGetState()	eTaskGetState()
osThreadIsSuspended()	eTaskGetState()
osThreadSuspend()	vTaskSuspend()
osThreadSuspendAll()	vTaskSuspendAll()
osThreadResume()	vTaskResume() xTaskResumeFromISR()
osThreadResumeAll()	xTaskResumeAll()
osDelayUntil()	vTaskDelayUntil()
osAbortDelay()	xTaskAbortDelay()
osThreadList()	vTaskList()

life.augmented



Intertask communication

CMSIS OS inter-task communication

- **Queues**. Allows to pass more information between the tasks. Suspend task if tries to "put" to full queue or "get" from empty one.
- **Semaphores** are used to communication between the tasks without specifying the ID of the thread who can accept it. It allows counting multiple events and can be accepted by many threads.
- **Direct to task notifications** are used to precise communication between the tasks. It is necessary to specify within signal thread id.
- **Mutexes** are used to guard the shared resources. It must be taken and released always in that order by each task that uses the shared resource.
- **Event Groups** are used to synchronize task with multiple events (OR-ed together). There could be 8 or 24 bit value used here (depends on configUSE_16_BIT_TICKS settings) – not implemented in CMSIS_OS API

FreeRTOS Queues

life.augmented

Queues (1/2)

- Queues are pipes to transfer data between tasks in RTOS
- By default queue is behaving as FIFO (First In - First Out); can be redefined to perform as LIFO (Last In - First Out) structure by using **xQueueSendToFront()** function (not available in current CMSIS-RTOS API).
- All data send by queue must be of the same type, declared during queue creation phase. It can be simple variable or structure.
- Within CMSIS-RTOS API there are two types of queues:
  - **Message** where one can send only integer type data or a pointer
  - **Mail** where one can send memory blocks

Queues (2/2)

- Length of queue is declared during creation phase and is defined as a number of items which will be send via queue.
- Operations within queues are performed in **critical sections** (blocking interrupts by programming BASEPRI register for the time of operation on queue.
- Tasks can block on queue sending or receiving data with a timeout or infinitely.
- If multiple tasks are blocked waiting for receiving/Sending data from/To a queue then only the task with the highest priority will be unblocked when a data/space is available. If both tasks have equal priority the task that has been waiting the longest will be unblocked.

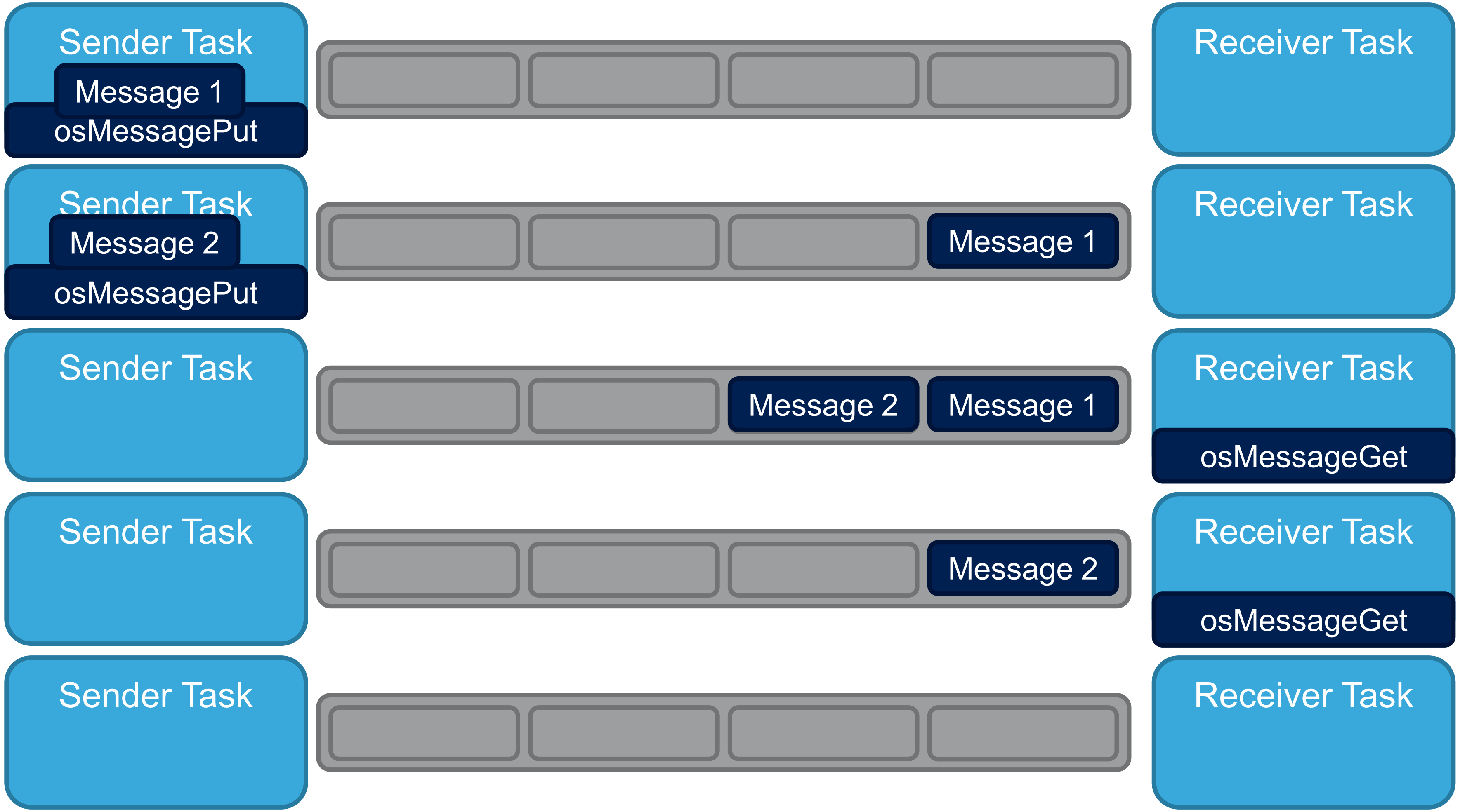
Queue structure management

queue.c

Name	Description	condition
*pcHead	Points to the beginning of the queue storage area	
*pcTail	Points to the byte at the end of the queue storage area. Once more byte is allocated than necessary to store the queue items, this is used as a marker	
*pcWriteTo	Points to the free next place in the storage area	
*pcReadFrom	Points to the last place that a queued item was read from when the structure is used as a queue	Use of a union is an exception to the coding standard to ensure two mutually exclusive structure members don't appear simultaneously (wasting RAM)
uxRecursiveCallCount	Maintains a count of the number of times a recursive mutex has been recursively 'taken' when the structure is used as a mutex	Use of a union is an exception to the coding standard to ensure two mutually exclusive structure members don't appear simultaneously (wasting RAM)
xTasksWaitingToSend	List of tasks that are blocked waiting to post onto this queue. Stored in priority order	
xTasksWaitingToReceive	List of tasks that are blocked waiting to read from this queue. Stored in priority order	
uxMessagesWaiting	The number of items currently in the queue	
uxLength	The length of the queue defined as the number of items it will hold, not the number of bytes.	
uxItemSize	The size of each items that the queue will hold.	
xRxLock	Stores the number of items received from the queue (removed from the queue) while the queue was locked. Set to queueUNLOCKED when the queue is not locked	
xTxLock	Stores the number of items transmitted to the queue (added to the queue) while the queue was locked. Set to queueUNLOCKED when the queue is not locked.	
uxQueueNumber		configUSE_TRACE_FACILITY == 1
ucQueueType		configUSE_TRACE_FACILITY == 1
*pxQueueSetContainer		configUSE_QUEUE_SETS == 1

life.augmented

Queue


Sender Task
Message 1
osMessagePut
Receiver Task
Sender Task
Message 2
osMessagePut
Message 1
Receiver Task
Sender Task
Message 2
Message 1
Receiver Task
osMessageGet
Sender Task
Message 2
Receiver Task
osMessageGet
Sender Task
Receiver Task

Queue

- Create Queue:

```
osMessageQId osMessageCreate (const osMessageQDef_t *queue_def, osThreadId thread_id)
```

Queue Handle

Create Queue

- Put data into Queue

```
osStatus osMessagePut (osMessageQId queue_id, uint32_t info, uint32_t millisec)
```

Queue handle

Item to send

Sending timeout

- Receive data from Queue

```
osEvent osMessageGet (osMessageQId queue_id, uint32_t millisec)
```

Structure with status and with received item

Queue handle

Receiving timeout

- Delete the queue

```
osStatus osMessageDelete (osMessageQId queue_id)
```

Queue handle

life.augmented

Queue

- Read an item from a Queue without removing the item from it:

```
osEvent osMessagePeek (osMessageQId queue_id, uint32_t millisec)
```

- osEvent – Structure with status and with received item
- osMessageQId queue_id – Queue handle
- uint32_t millisec – Receiving timeout

- Get the number of messages stored in a queue

```
uint32_t osMessageWaitingPeek (osMessageQId queue_id)
```

- uint32_t – Number of the messages stored in a queue
- osMessageQId queue_id – Queue handle

- Get the available space in a message queue

```
uint32_t osMessageAvailableSpace(osMessageQId queue_id)
```

- uint32_t – Available space in a message queue
- osMessageQId queue_id – Queue handle

Queue

- osEvent structure

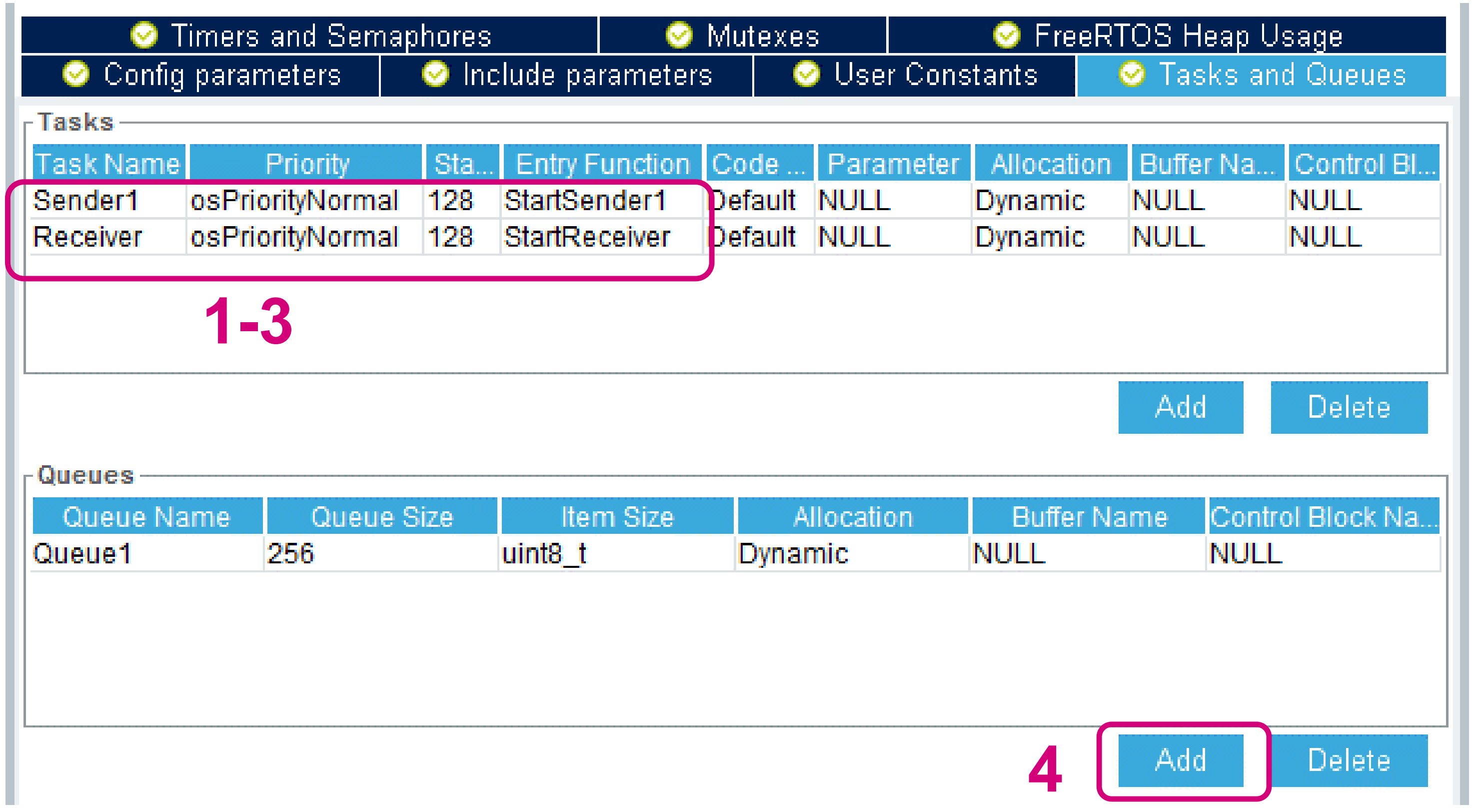
```
typedef struct  {
 osStatus        status;     ///< status code: event or error information
 union  {
   uint32_t      v;          ///< message as 32-bit value
   void          *p;         ///< message or mail as void pointer
   int32_t       signals;    ///< signal flags
 } value;                    ///< event value
 union  {
   osMailQId     mail_id;    ///< mail id obtained by \ref osMailCreate
   osMessageQId  message_id; ///< message id obtained by \ref osMessageCreate
 } def;                      ///< event definition
} osEvent;
```

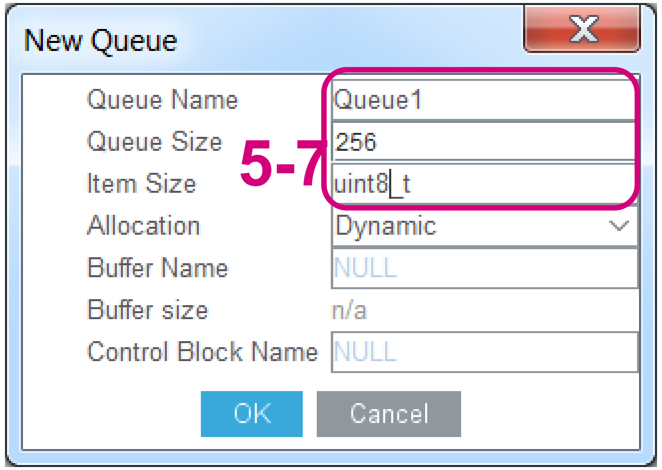
- If we want to get data from osEvent we must use:
  - `osEventName.value.v` if the value is 32bit message(or 8/16bit)
  - `osEventName.value.p` and retype on selected datatype

life.augmented

Queue lab

- Tasks part:
  1. Rename tasks to Sender1 and Receiver and its functions.
  2. If deleted, old tasks will be removed (with USER CODE !!!) from the code.
     To keep the user code, just rename the task.
  3. Set both tasks to normal priority
- Queue part
  4. Button Add
  5. Set queue size to **256**
  6. Queue type to **uint8_t**
  7. Button OK





life.augmented

printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

Queue lab

code processing

- Queue handle is now defined

```
/* Private variables ---------------------------------------------------------*/
osThreadId Sender1Handle;
osThreadId ReceiverHandle;
osMessageQId Queue1Handle;
```

- Queue item type initialization, length definition and create of queue and memory allocation

```
/* Create the queue(s) */
 /* definition and creation of Queue1 */
 osMessageQDef(Queue1, 256, uint8_t);
 Queue1Handle = osMessageCreate(osMessageQ(Queue1), NULL);
```

Queue item definition

Queue size

Queue lab

code processing

- Sender1 task

```
void StartSender1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task1\n");
    osMessagePut(Queue1Handle, 0x1, 200);
    printf("Task1 delay\n");
    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

Put value ‘1’ into queue

Item to send

Timeout for send

Queue handle

Queue lab

code processing

- Receiver task

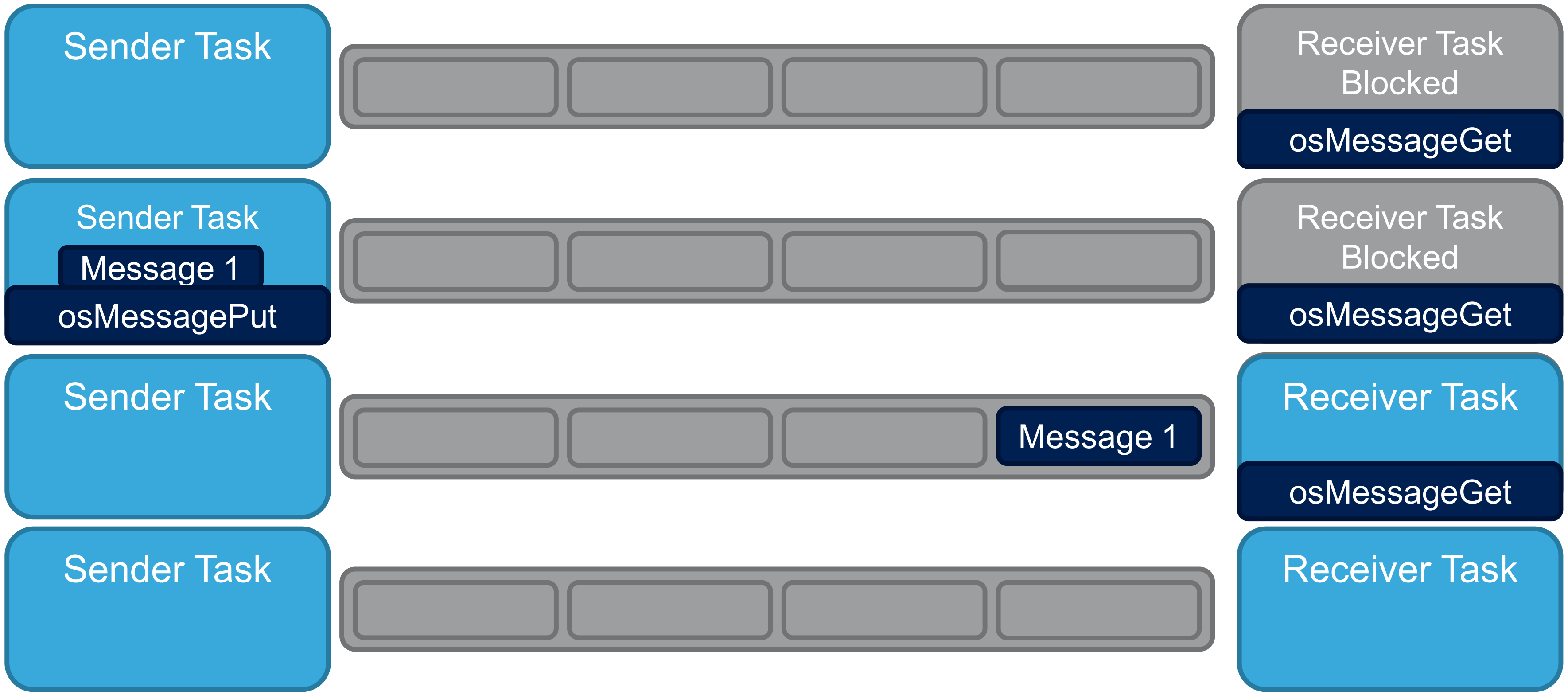
```
/* StartReceiver function */
void StartReceiver(void const * argument)
{
  /* USER CODE BEGIN StartReceiver */
  osEvent retvalue;
  /* Infinite loop */
  for(;;)
  {
    printf("Task2\n");
    retvalue=osMessageGet(Queue1Handl  000);
    printf(“%d \n",retvalue.value.
  }
  /* USER CODE END StartReceiver */
}
```

Get item from queue

Queue hand

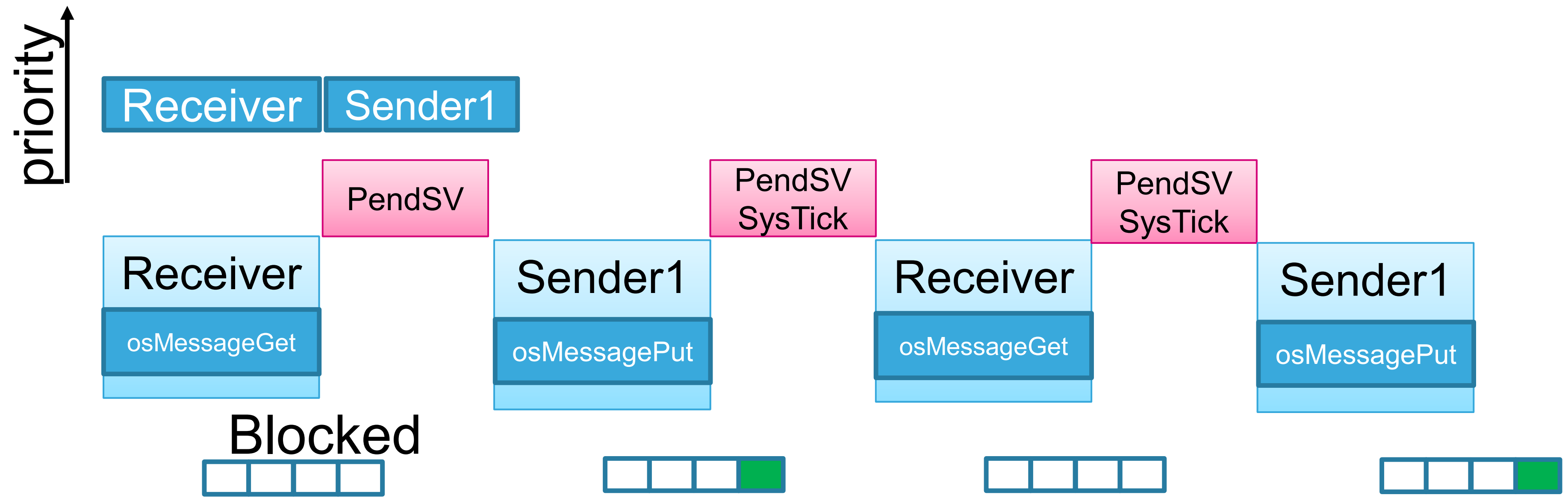
How long we wait on data in queue
It will block task

Queue Blocking



Queue Blocking

- After calling **osMessagePut()**
  - If there is no free space in queue the Sender task is blocked for settable time then it will continue (without sending the data)
  - If there is free space in queue the Sender task will continue just after data send
- After calling **osMessageGet()**
  - If any data are not in queue the Receiver task is blocked for settable time then it will continue (without data reception)
  - If the data are in queue the task will continue just after data reception



Two senders lab

- Let's create two sending tasks: Sender1, Sender2 and one Receiver task with the same priorities.

Timers and Semaphores | Mutexes | FreeRTOS Heap Usage

Config parameters | Include parameters | User Constants | Tasks and Queues

Tasks

Task Name	Priority	Sta...	Entry Function	Code ...	Parameter	Allocation	Buffer Na...	Control Bl...
Sender1	osPriorityNormal	128	StartSender1	Default	NULL	Dynamic	NULL	NULL
Receiver	osPriorityNormal	128	StartReceiver	Default	NULL	Dynamic	NULL	NULL
Sender2	osPriorityNormal	128	StartSender2	Default	NULL	Dynamic	NULL	NULL

Add Delete

Queues

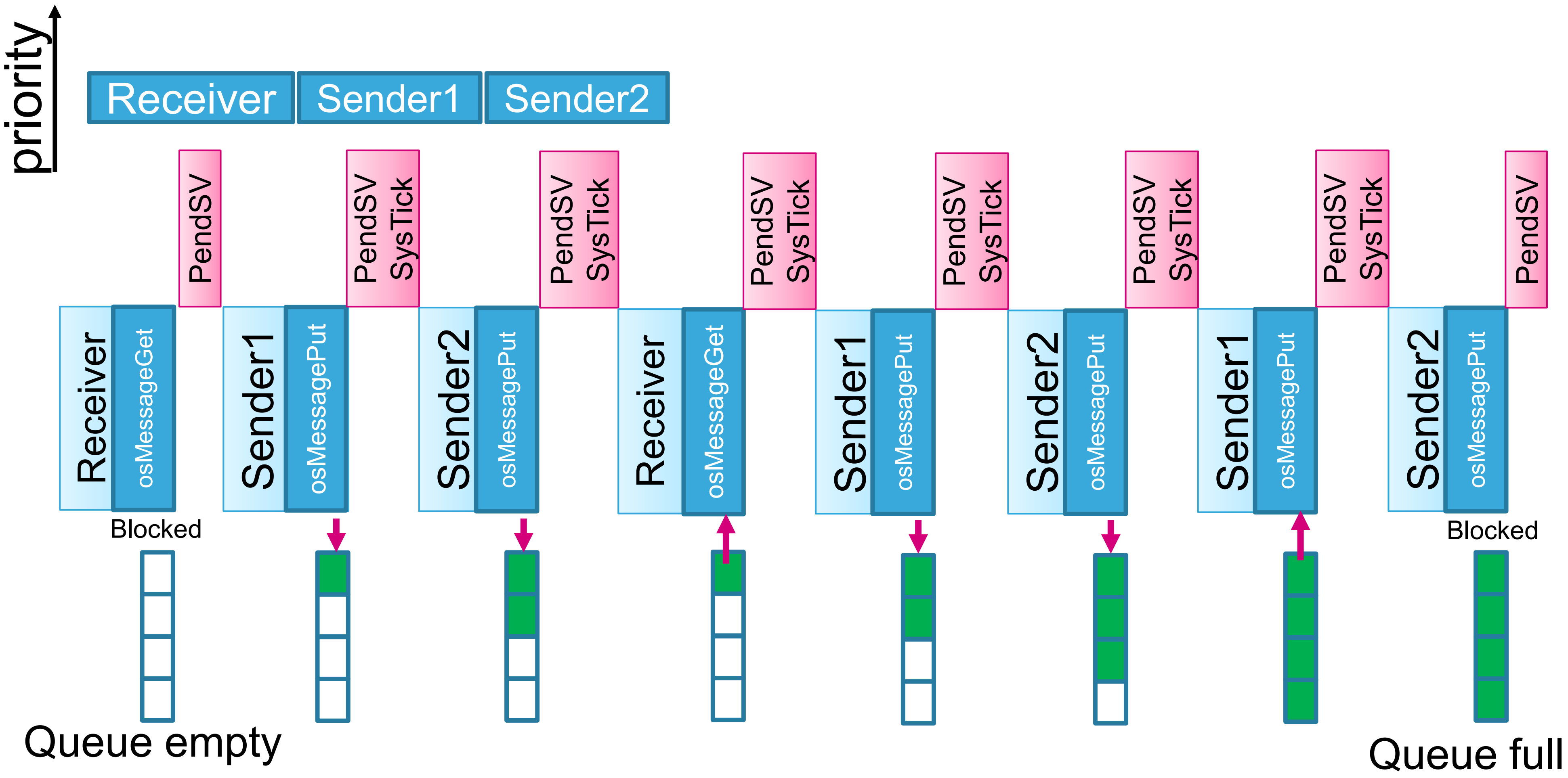
Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block Na...
Queue1	256	uint8_t	Dynamic	NULL	NULL

Add Delete

life.augmented

Multiple senders, one receiver

- Because tasks have same priority, receiver will get data from queue after both task put data into queue
- What would happened if will be more tasks?



Two senders lab

- Two sending tasks
- They are same no change necessary

```c
void StartSender1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task1\n");
    osMessagePut(Queue1Handle,0x1,200);
    printf("Task1 delay\n");
    osDelay(2000);
  }
  /* USER CODE END 5 */
}
```

```c
void StartSender2(void const * argument)
{
  /* USER CODE BEGIN StartSender2 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task2\n");
    osMessagePut(Queue1Handle,0x2,200);
    printf("Task2 delay\n");
    osDelay(2000);
  }
  /* USER CODE END StartSender2 */
}
```

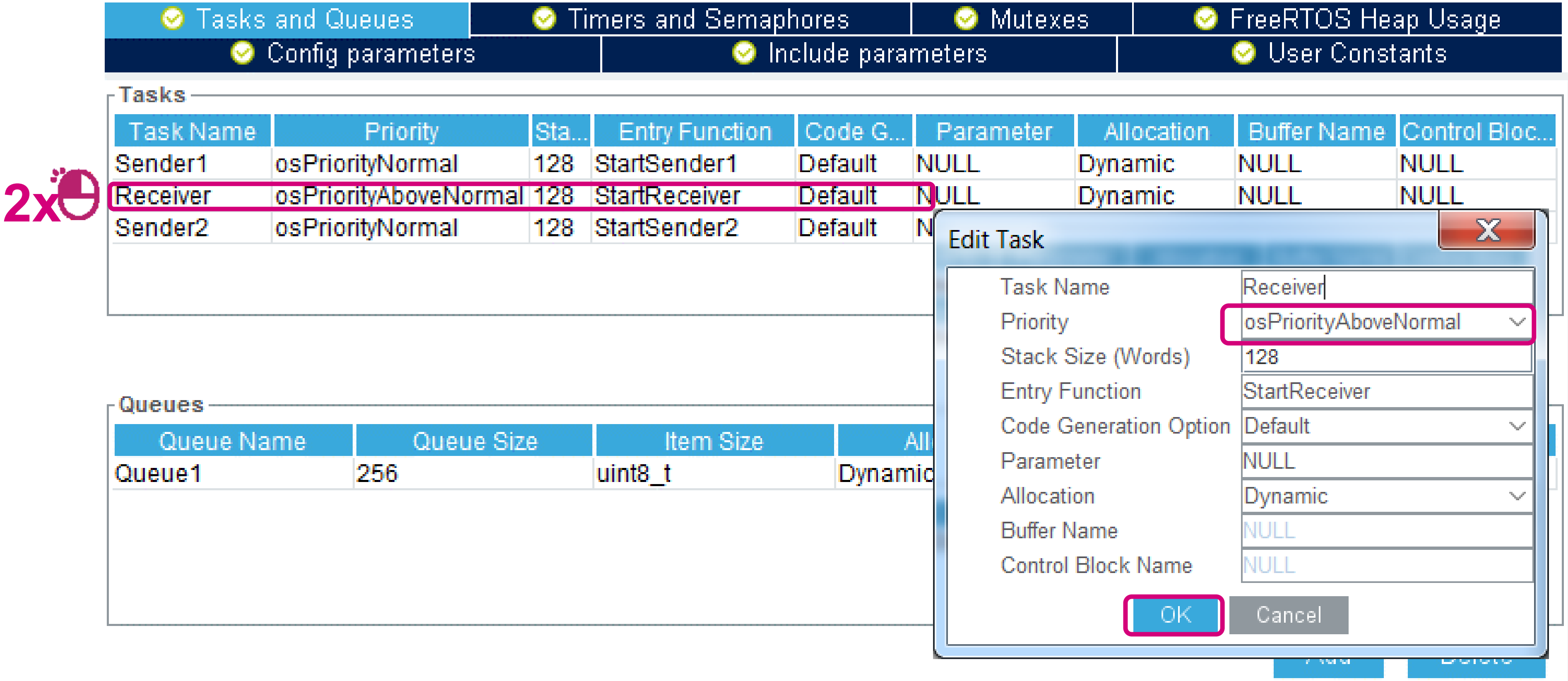
Two senders lab

- Simple receiver

```
/* StartReceiver function */
void StartReceiver(void const * argument)
{
  /* USER CODE BEGIN StartReceiver */
  osEvent retvalue;
  /* Infinite loop */
  for(;;)
  {
    retvalue=osMessageGet(Queue1Handle,4000);
    printf("Receiver\n");
    printf("%d \n",retvalue.value.p);
  }
  /* USER CODE END StartReceiver */
}
```

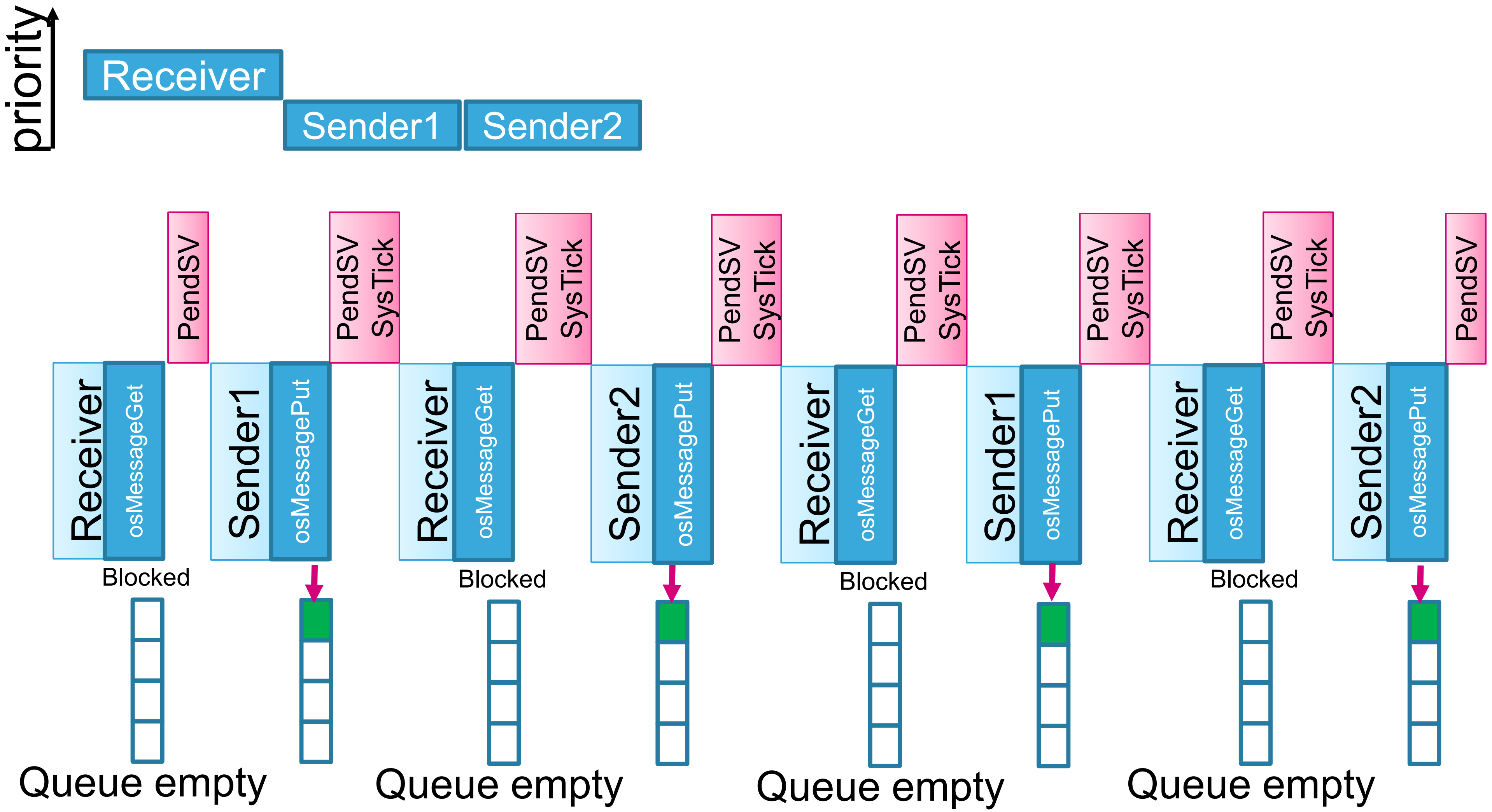
Receiver with higher priority lab

- Senders have same priority
- Receiver have higher priority than senders
- Please verify whether behavior is inline with expectations



Receiver with higher priority lab

- Receiver is now unblocked every time when sender tasks put data into queue



Single sender, two receivers

- Message from the queue is taken by the task with higher priority
- In case of equal priorities currently executed or first executed task will get the message. It is not deterministic.

Queue items lab

- Queues allow to define type (different variables or structures) which the queue use.
- Within Queue1 Item size put a structure called Data
- Regenerate project

Queues

Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block Name
Queue1	16	Data	Dynamic	NULL	NULL

life.augmented

Queue items lab

- Create new structure type for data

```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
  uint16_t Value;
  uint8_t Source;
} Data;
```

- Define Structures which will be sent from sender task

```
/* Declare two variables of type Data that will be passed on the queue. */
Data DataToSend1={0x2018,1};
Data DataToSend2={0x2019,2};
```

Queue items lab

- Sent data from Sender1 task

```
void StartSender1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    printf("Task1\n");
    osMessagePut(Queue1Handle,(uint32_t)&DataToSend1,200);
    printf("Task1 delay\n");
    osDelay(2000);
  }
  /* USER CODE END 5 */
}
```

Put data into queue

- **Prepare similar code for Sender2**

life.augmented

Queue

- osEvent structure

```
typedef struct  {
 osStatus        status;     ///< status code: event or error information
 union  {
   uint32_t      v;          ///< message as 32-bit value
   void          *p;         ///< message or mail as void pointer
   int32_t       signals;    ///< signal flags
 } value;                    ///< event value
 union  {
   osMailQId     mail_id;    ///< mail id obtained by \ref osMailCreate
   osMessageQId  message_id; ///< message id obtained by \ref osMessageCreate
 } def;                      ///< event definition
} osEvent;
```

- If we want to get data from osEvent we must use:
  - `osEventName.value.v` if the value is 32bit message(or 8/16bit)
  - `osEventName.value.p` and retype on selected datatype

life.augmented

Queue items lab

- Receiver data from sender task

```c
/* StartReceiver function */
void StartReceiver(void const * argument)
{
  /* USER CODE BEGIN StartReceiver */
  osEvent retvalue;
  /* Infinite loop */
  for(;;)
  {
    retvalue=osMessageGet(Queue1Handle,4000);
    if(((Data*)retvalue.value.p)->Source==1){
      printf("Receiver Receive message from Sender 1\n");
    }else{
      printf("Receiver Receive message from Sender 2\n");
    }
    printf("Data: %d \n",((Data*)retvalue.value.p)->Value);
  }
  /* USER CODE END StartReceiver */
}
```

Get data from queue

Decode data from osEvent structure

Mail Queue

- In mail queue we are transferring memory blocks which needs to be allocated (before put the data there) and freed (after taking data out)
- Mail queue passes pointers to allocated memory blocks within the message queue, so there is no big data transfers. It is an advantage to message queues.

Mail Queue

- Create Mail Queue:

```
osMailQId osMailCreate (const osMailQDef_t *queue_def, osThreadId thread_id)
```

Mail Queue Handle

Create Mail Queue

- Put a mail to a Queue

```
osStatus osMailPut (osMailQId queue_id, void * mail)
```

Status of the operation

Mail Queue handle

- Receive mail from a Queue

```
osEvent osMessageGet (osMailQId queue_id, uint32_t millisec)
```

Structure with status and with received item

Mail Queue handle

- Free a memory block from a mail

```
osStatus osMailFree (osMailQId queue_id, void *mail)
```

Status of the operation

Mail Queue handle

life.augmented

Mail Queue

- Allocate a memory block from a mail

```
void * osMailAlloc (osMailQId queue_id, uint32_t millisec)
```

Mail Queue handle

- Allocate a memory block from a mail and set memory block to zero

```
void * osMailCAlloc (osMailQId queue_id, uint32_t millisec)
```

Mail Queue handle

Queues APIs

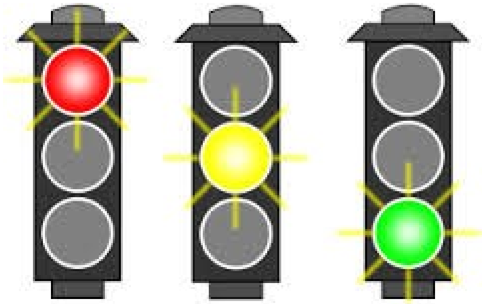
CMSIS_RTOS API	FreeRTOS API
osMessageCreate()	xQueueCreateStatic() xQueueCreate()
osMessagePut()	xQueueSend() xQueueSendFromISR()
osMessageGet()	xQueueReceive() xQueueReceiveFromISR()
osMessageDelete()	vQueueDelete(queue_handler)
osMessageWaiting()	**uxQueueMessagesWaiting(queue_handler)** **uxQueueMessagesWaitingFromISR(queue_handler)**
-	**xQueueSendToBack**(queue_handle,*to_queue,block_time) **xQueueSendToBackFromISR**(queue_handle,*to_queue,block_time)
-	**xQueueSendToFront**(queue_handle,*to_queue,block_time) **xQueueSendToFrontFromISR**(queue_handle,*to_queue,block_time)
osMessagePeek()	**xQueuePeek**(queue_handle,*to_queue,block_time)
osMessageAvailableSpace()	Returns uxQueueSpacesAvailable

life.augmented

Mail Queue APIs

CMSIS_RTOS API	FreeRTOS API
osMailCreate()	pvPortMalloc(), xQueueCreate(), osPoolCreate()
osMailAlloc()	osPoolAlloc()
osMailCAlloc()	osMailAlloc(),
osMailPut()	xQueueSendFromISR() xQueueSend()
osMailGet()	xQueueReceiveFromISR() xQueueReceive()
osMailFree()	osPoolFree()

FreeRTOS Semaphores

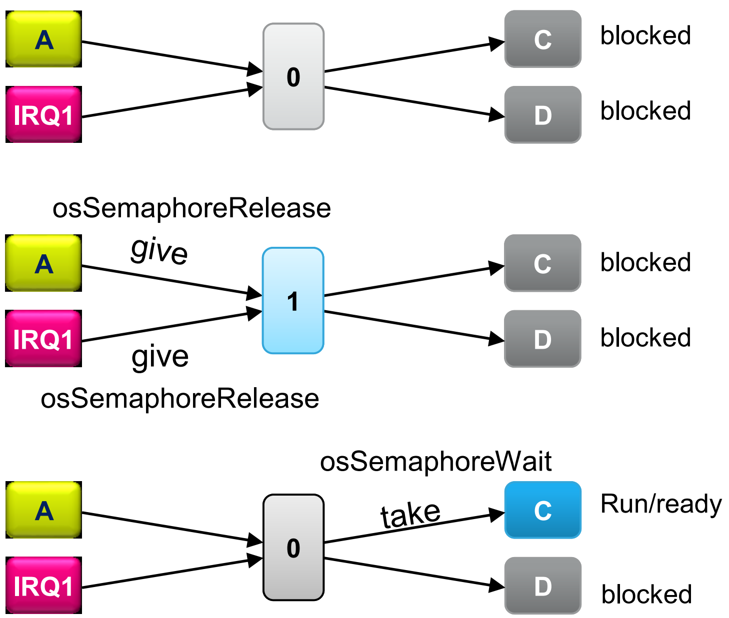


Semaphores

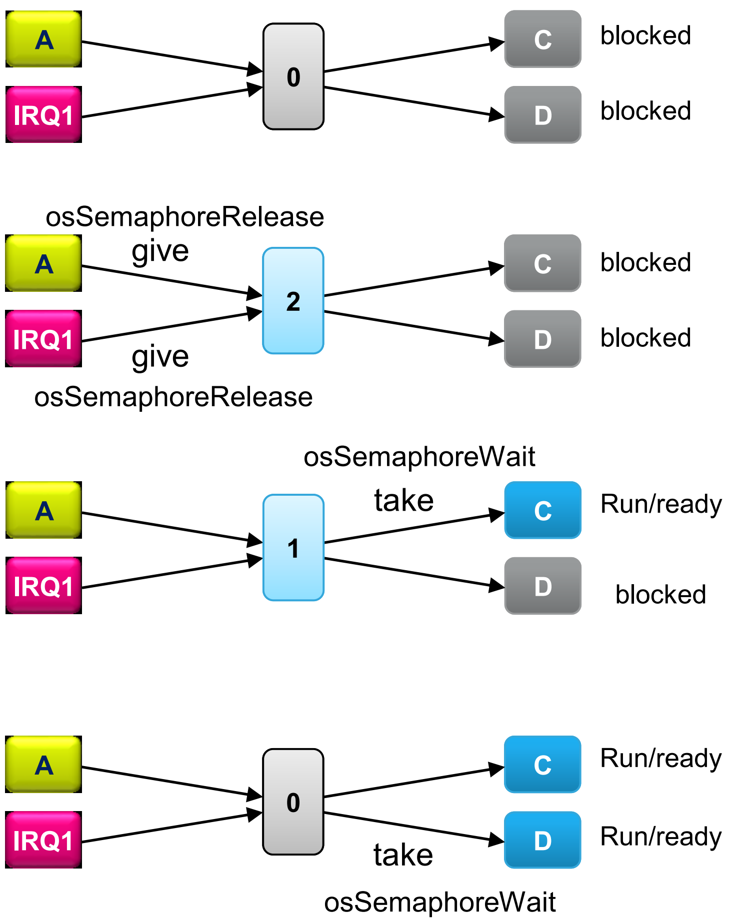
- Semaphores are used to synchronize tasks with other events in the system (especially IRQs)
- Waiting for semaphore is equal to wait() procedure, task is in blocked state not taking CPU time
- Semaphore should be created before usage
- In FreeRTOS implementation semaphores are based on queue mechanism
- In fact those are queues with length 1 and data size 0
- There are following types of semaphores in FreeRTOS:
  - **Binary** – simple on/off mechanism
  - **Counting** – counts multiple ***give*** and multiple ***take***
  - **Mutex** – Mutual Exclusion type semaphores (explained later on)
  - **Recursive** (in CMSIS FreeRTOS used only for Mutexes)
- **Turn on** semaphore = **give** a semaphore can be done from other task or from interrupt subroutine (function `osSemaphoreRelease()`)
- **Turn off** semaphore = **take** a semaphore can be done from the task (function `osSemaphoreWait()`)

life.augmented

Semaphores: binary vs counting

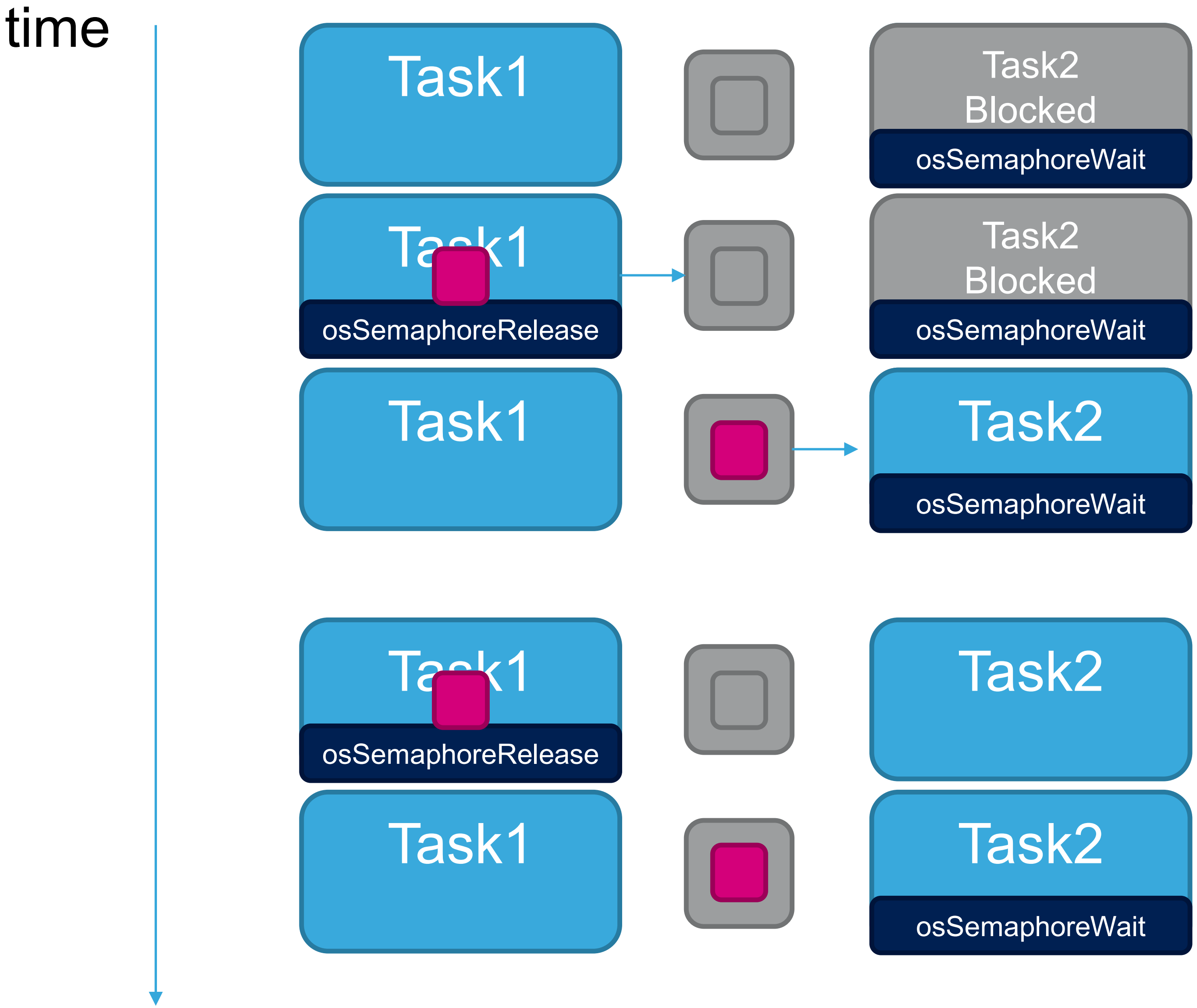


Binary



Counting

Binary Semaphore



Binary Semaphore

- Semaphore creation

```
osSemaphoreId osSemaphoreCreate (const osSemaphoreDef_t *semaphore_def, int32_t count)
```

Semaphore handle

Semaphore definition

Semaphore 'tokens'
For binary semaphore is 1

- Wait for Semaphore release

```
int32_t osSemaphoreWait (osSemaphoreId semaphore_id, uint32_t millisec)
```

Should be: Number of 'tokens in semaphore', but it is status osStatus (in v 24.12.2014) like for Mutex

Semaphore handle

How long wait for semaphore release
- 0 – no wait
- >0 – delay in ms
- 0xFFFFFFFF - forever

- Semaphore release

```
osStatus osSemaphoreRelease (osSemaphoreId semaphore_id)
```

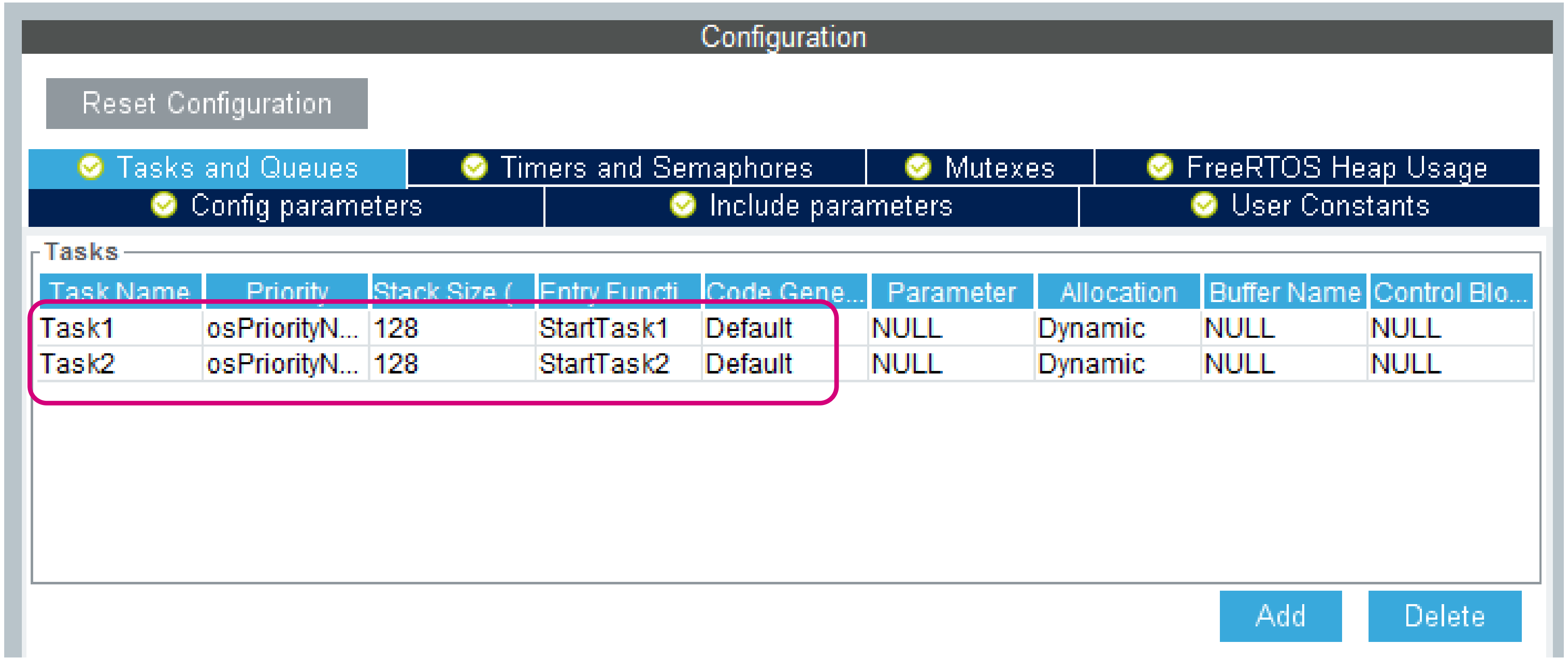
Return status

Semaphore handle

life.augmented

Binary Semaphore lab

- Create two tasks Task1, Task2 with the same priorities

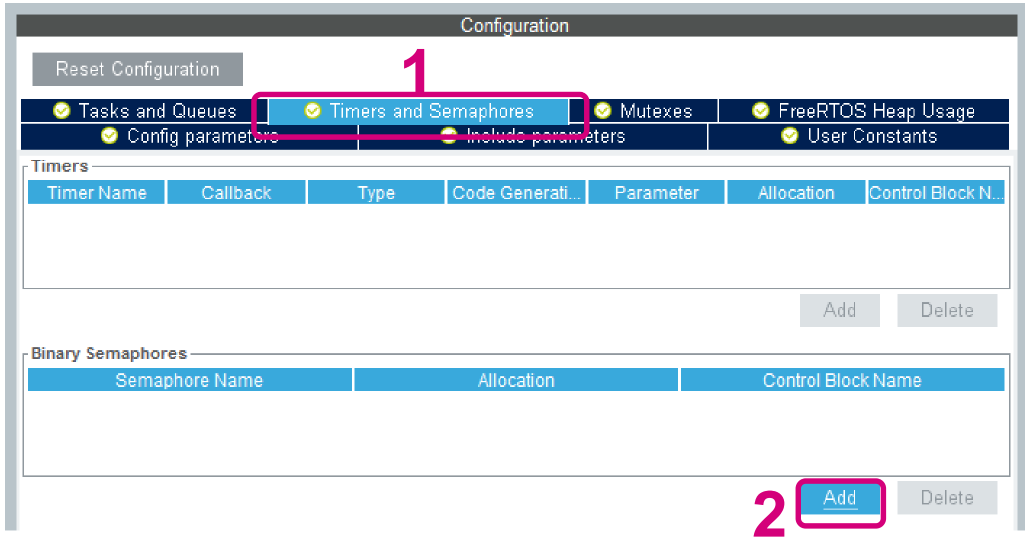


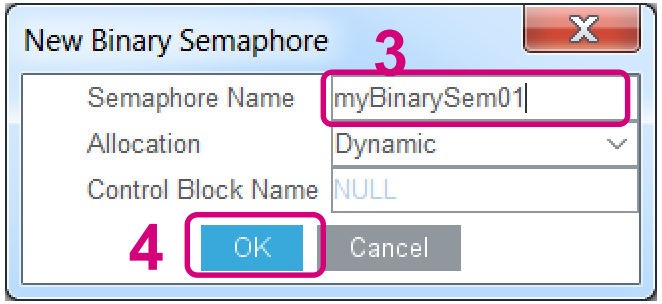
life.augmented

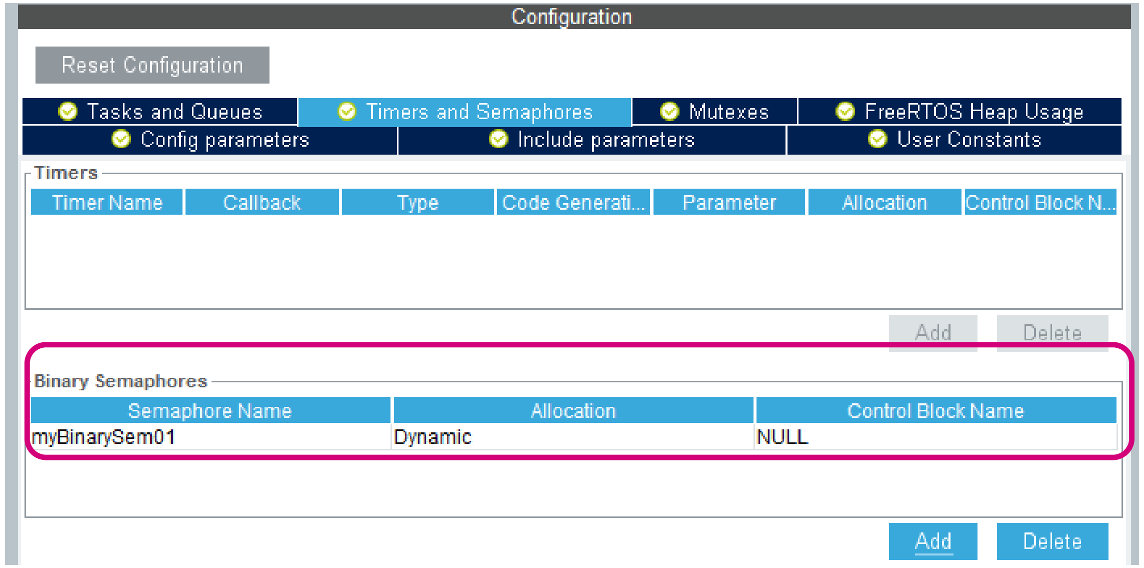
Binary Semaphore lab

Create binary semaphore

1. Select Timers and Semaphore tab
2. Click **Add** button in Binary Semaphores section
3. Set name: **myBinarySem01**
4. Click **OK** button



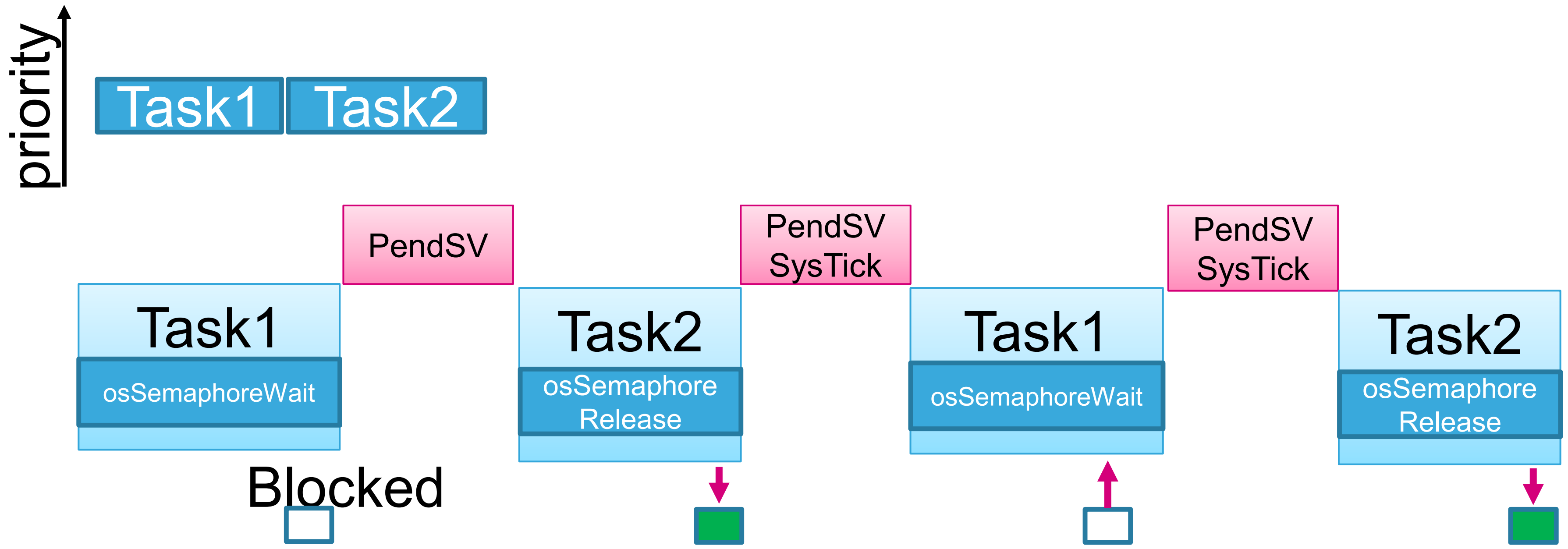




life.augmented

Binary Semaphore lab

- Task1 is synchronized with Task2
- Both tasks have the same priorities
- Task1 is waiting for semaphore (with 4sec delay)
- Task2 is releasing the semaphore



printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

life.augmented

Binary Semaphore lab

code processing

- Semaphore handle definition

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
osThreadId Task2Handle;
osSemaphoreId myBinarySem01Handle;
```

- Semaphore creation

```
/* Create the semaphores(s) */
/* definition and creation of myBinarySem01 */
osSemaphoreDef(myBinarySem01);
myBinarySem01Handle = osSemaphoreCreate(osSemaphore(myBinarySem01), 1);
```

life.augmented

Binary Semaphore lab

code processing

- Semaphore release usage
- If tasks/interrupt is done the semaphore is released

```
void StartTask1(void const * argument)
{

  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(2000);
    printf("Task1 Release semaphore\n");
    osSemaphoreRelease(myBinarySem01Handle);
  }
  /* USER CODE END 5 */
}
```

Binary Semaphore lab

code processing

- Semaphore wait usage
- Second task waits on semaphore release
  After release task is unblocked and continue in work

```
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
    osSemaphoreWait(myBinarySem01Handle,4000);
    printf("Task2 synchronized\n");
  }
  /* USER CODE END StartTask2 */
}
```

Binary Semaphore lab

code processing

- Semaphore can be released from interrupt (if interrupt priority is below – higher number in CortexM cores - configured `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`)
- Using HAL libraries we can release semaphore in the callback (JOY_CENTER button press):

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  osSemaphoreRelease(myBinarySem01Handle);
}
/* USER CODE END 4 */
```

configUSE_COUNTING_SEMAPHORE

Counting semaphores

- Counting semaphores can be seen as a as queues of length greater than one. users of the semaphore (Tasks, IT) are not interested in the data that is stored in the queue, just whether the queue is empty or not.

- Counting semaphores are typically used for two purposes:
  - **Counting events** : an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the count value to be zero when the semaphore is created.
  - **Resource management** : the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it releases (gives) the semaphore back incrementing the semaphore count value. In this case it is desirable for the count value to be equal the maximum count value when the semaphore is created.

life.augmented

Counting semaphores

- API is the same as for Binary semaphore
- Semaphore creation

```
osSemaphoreId osSemaphoreCreate (const osSemaphoreDef_t *semaphore_def, int32_t count)
```

- Wait for Semaphore release

```
int32_t osSemaphoreWait (osSemaphoreId semaphore_id, uint32_t milisec)
```

Return value (osStatus):

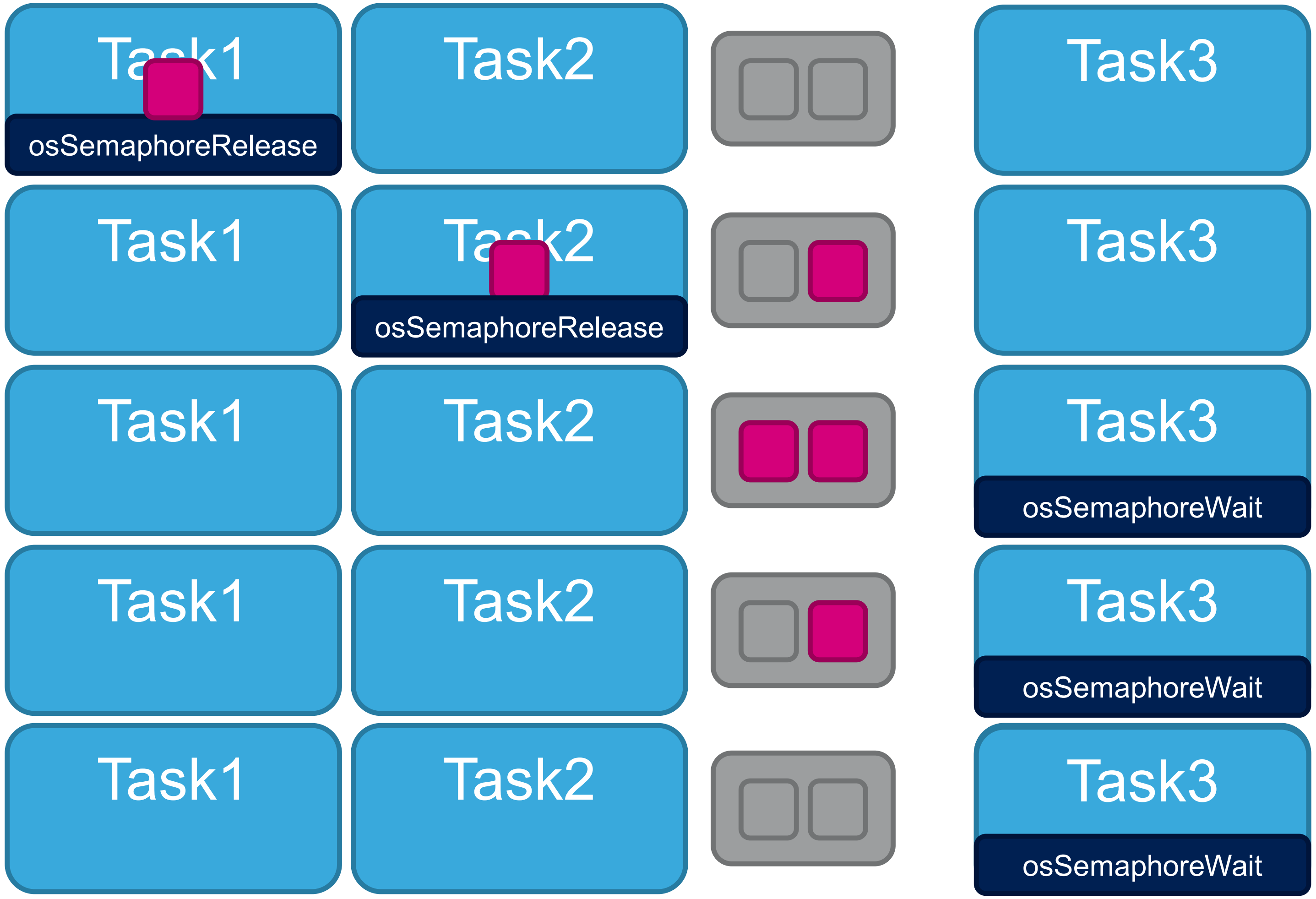
- 0 – semaphore released within given timeout (milisec)
- 0xFF – semaphore not released

0 – no delay
>0 – delay in ms
0xFFFFFFFF – wait forever

- Semaphore release

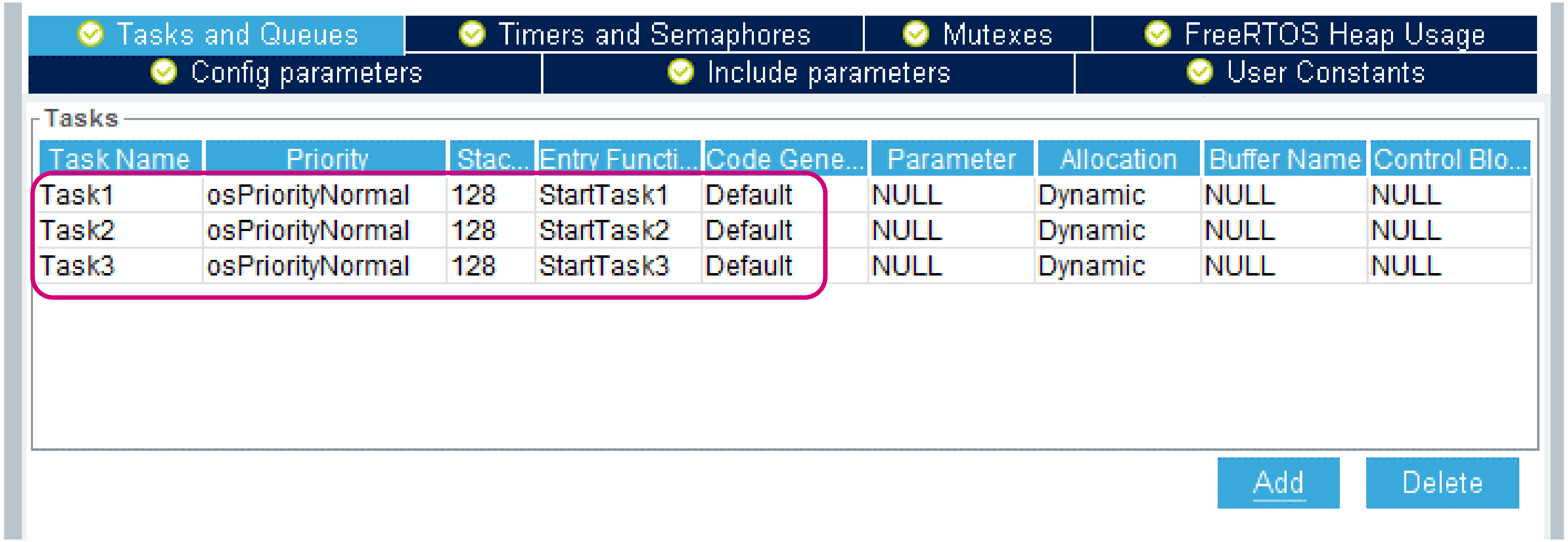
```
osStatus osSemaphoreRelease (osSemaphoreId semaphore_id)
```

Counting Semaphore



Counting Semaphore lab

- Create three tasks (Task1, Task2, Task3) with same priority
- Set entry function o StartTask1,2,3 respectively
- Keep all other parameters in default value



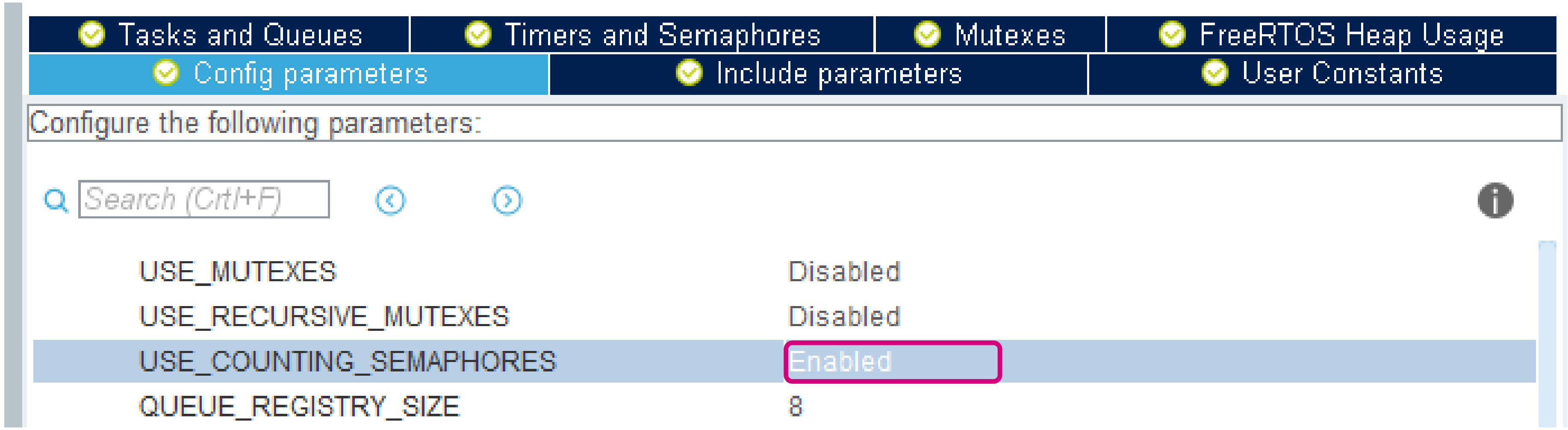
Task Name	Priority	Stac...	Entry Functi...	Code Gene...	Parameter	Allocation	Buffer Name	Control Blo...
Task1	osPriorityNormal	128	StartTask1	Default	NULL	Dynamic	NULL	NULL
Task2	osPriorityNormal	128	StartTask2	Default	NULL	Dynamic	NULL	NULL
Task3	osPriorityNormal	128	StartTask3	Default	NULL	Dynamic	NULL	NULL

life.augmented

Counting Semaphore lab

Enable Counting semaphore

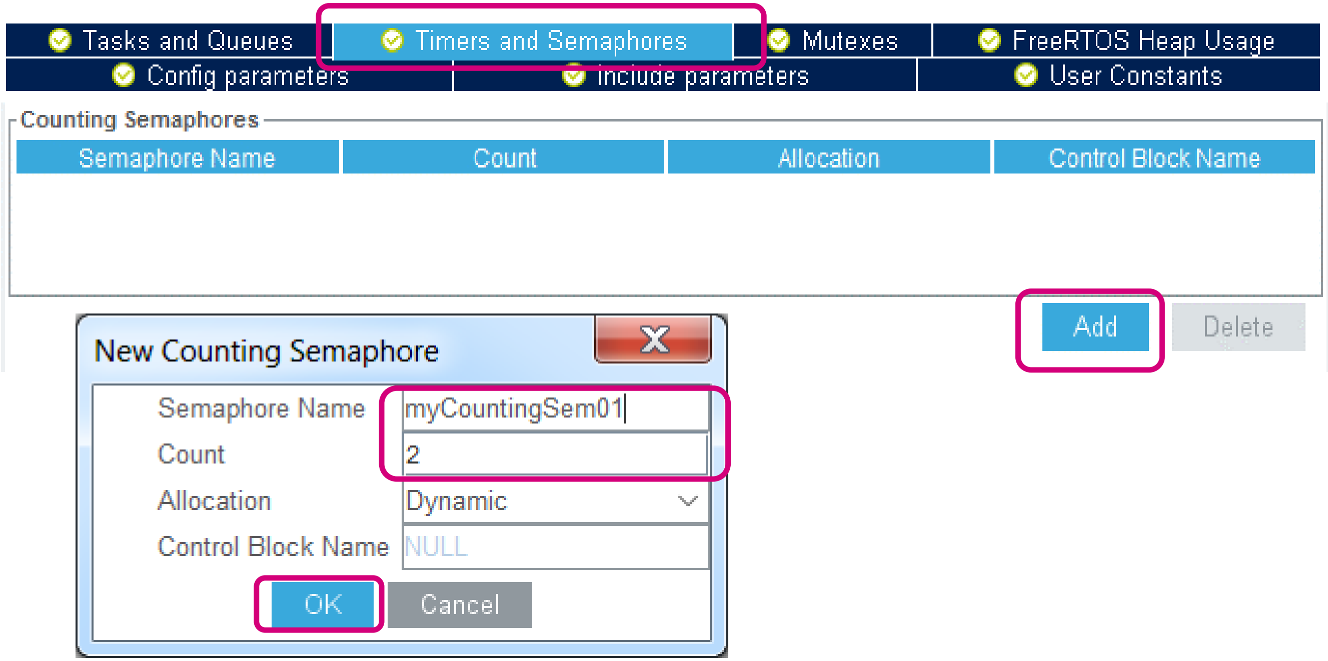
1. Select Config parameters tab
2. Change “USE_COUNTING_SEMAPHORES” to **Enabled**

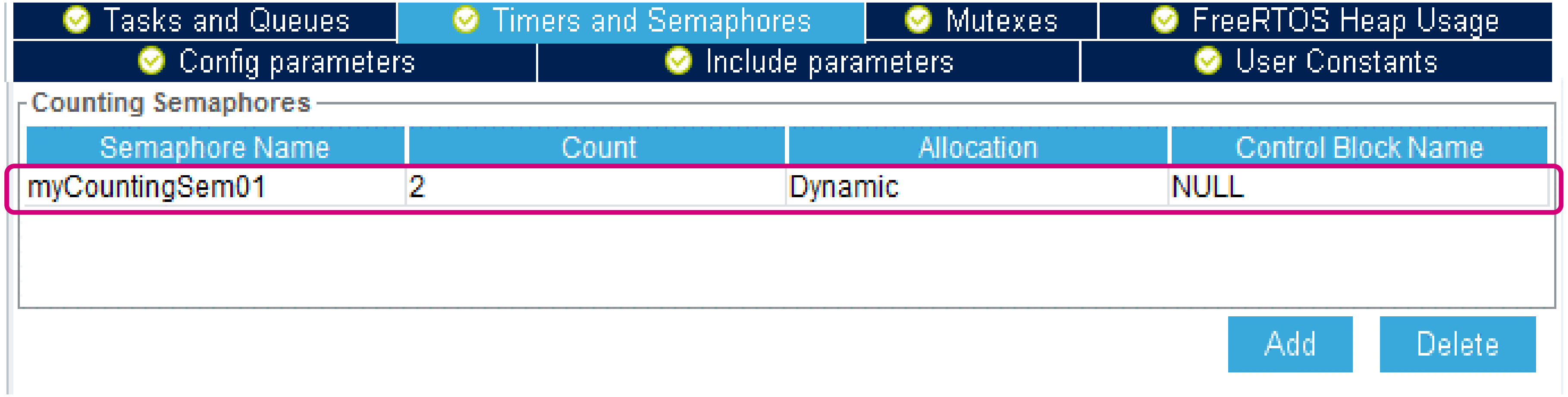


Counting Semaphore lab

Create Counting semaphore

1. Select Timers and Semaphores tab
2. Click Add button in Counting Semaphores section
3. Set name to myCountingSem01
4. Set count of tokens to 2
5. Click OK button

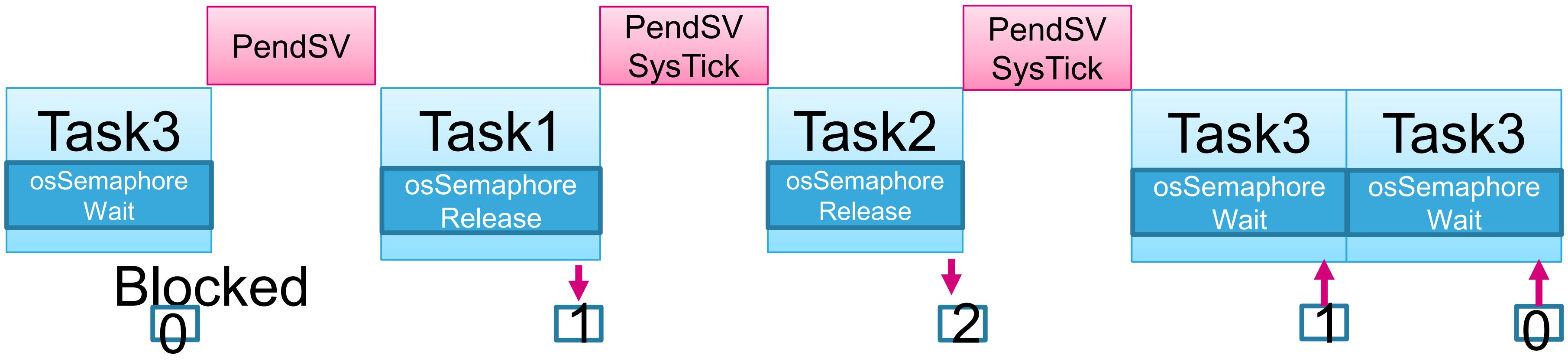




life.augmented

Counting Semaphore lab

- Task1 and Task2 release semaphore
- Task 3 wait for two tokens



printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

Counting Semaphore lab

code processing

- Create Counting semaphore

```
/* Create the semaphores(s) */
 /* definition and creation of myCountingSem01 */
 osSemaphoreDef(myCountingSem01);
 myCountingSem01Handle = osSemaphoreCreate(osSemaphore(myCountingSem01), 2);
```

- Task1 and Task2 will be same

```
void StartTask1(void const * argument)
{

  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(2000);
    printf("Task1 Release counting semaphore\n");
    osSemaphoreRelease(myCountingSem01Handle);
  }
  /* USER CODE END 5 */
}
```

```
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
     osDelay(2000);
     printf("Task2 Release counting semaphore\n");
     osSemaphoreRelease(myCountingSem01Handle);
  }
  /* USER CODE END StartTask2 */
}
```

Counting Semaphore lab

code processing

- Task3 will wait until semaphore will be 2 times released

```
void StartTask3(void const * argument)
{
  /* USER CODE BEGIN StartTask3 */
  /* Infinite loop */
  for(;;)
  {
  osSemaphoreWait(myCountingSem01Handle, 4000);
  osSemaphoreWait(myCountingSem01Handle, 4000);
  printf("Task3 synchronized\n");
  }
  /* USER CODE END StartTask3 */
}
```

Semaphores APIs

CMSIS_RTOS API	FreeRTOS API
osSemaphoreCreate()	vSemaphoreCreateBinaryStatic() vSemaphoreCreateCountingStatic() vSemaphoreCreateBinary() xSemaphoreCreateCounting()
osSemaphoreWait()	xSemaphoreTake() xSemaphoreTakeFromISR()
osSemaphoreRelease()	xSemaphoreGive() xSemaphoreGiveFromISR()
osSemaphoreDelete()	vSemaphoreDelete()

Direct to task notification

CMSIS-OS – Signals

FreeRTOS – Task Notification

Direct to task notification

- FreeRTOS Direct Task notifications feature is available starting from release 8.2.0.
- Within CMSIS_OS it is covered by less featured Signals.
- Each FreeRTOS task has a 32-bit *notification value*. An *RTOS task notification* is an event sent directly to a task that can unblock the receiving task.
- Task notifications can be used where previously it would have been necessary to create a separate queue, binary semaphore, counting semaphore or event group. Unblocking an RTOS task with a direct notification is **45% faster** and **uses less RAM** than unblocking a task with a binary semaphore.
- Task notification RAM footprint and speed advantage over other FreeRTOS feature (performing equivalent functionalities). Nevertheless It presents following limitations:
  - Task notifications can only be used to notify only one Task at a time : i.e only one task can be the recipient of the event. This condition is however met in the majority of real world applications.
  - If Task notification is used in place of a message queue then the receiving task (waiting for the notification) is set to the blocked state. However The sending task (sending the notification) cannot wait in the Blocked state for a send to complete if the send cannot complete immediately

Signals

- Signals are used to trigger execution states between the threads and from IRQ to thread.
- Each thread has up to 31 assigned signal flags.
- The maximum number of signal flags is defined in **cmsis_os.h** (`osFeature_Signals`). It is set to 8. It is not possible to configure signals from STM32CubeMX.
- Main functions:
  - `osSignalSet()` - set specified signal flags of an active thread

```
int32_t osSignalSet (osThreadId thread_id, int32_t signals)
```

  - `osSignalWait()` - wait for one or more signal flags for running thread

```
osEvent osSignalWait (int32_t signals, uint32_t milisec)
```

Signals

example

- We can reuse existing Tasks_lab
- Let’s define any signal

```
#define SIGNAL_BUTTON_PRESS      1  /* USER CODE BEGIN PD */
```

- Task1 is waiting (being in blocked mode) for an external interrupt occurrence.

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
   osSignalSet(Task1Handle,SIGNAL_BUTTON_PRESS);
}
```

- Within external interrupt callback SIGNAL_BUTTON_PRESS is send to Task1

```
void StartTask1(void const * argument)
{
 for(;;)
  {
   osSignalWait(SIGNAL_BUTTON_PRESS,osWaitForever); /* USER CODE BEGIN 5 */
   HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
  }
}
```

- RED LED will be toggled on each button press.

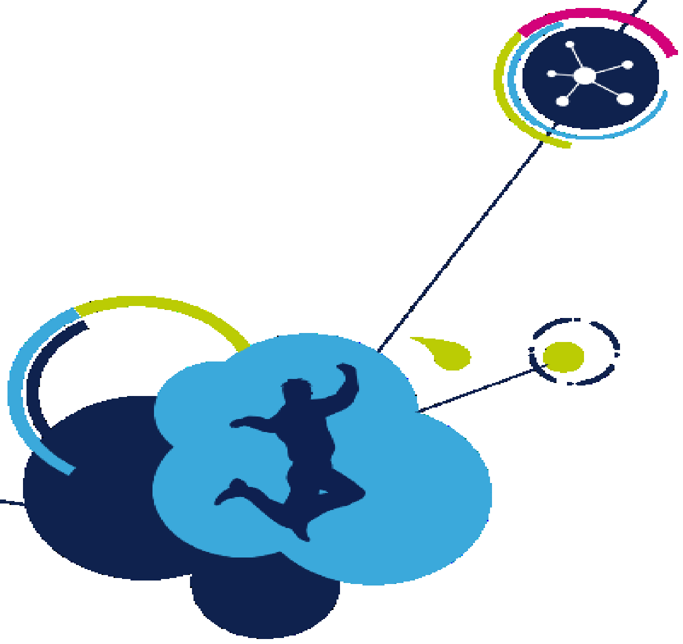
Signals

CMSIS_RTOS API	FreeRTOS API
osSignalSet() (given value is OR-ed with current notification value of given task – eSetBits action set in cmsis_os.c)	**xTaskGenericNotify()** **xTaskGenericNotifyFromISR()**
osSignalClear() (empty declaration)	Not available
osSignalWait() (it is clearing notification value)	**xTaskNotifyWait()**
osSignalGet() (removed in CMSIS_OS v1.02)	Not available
-	xTaskNotifyGive() vTaskNotifyGiveFromISR()
-	ulTaskNotifyTake()
-	xTaskNotifyStateClear()

Signals (task notifications) cannot be used:

- To send an event or data to IRQ
- To communicate with more than one task (thread)
- To buffer multiple data items

life.augmented



FreeRTOS
Resources management

Resource management

- **Critical sections** – when it is necessary to block small piece of code inside the task against task switching or interrupts. This section should start with macro `taskENTER_CRITICAL()`, and should end with macro `taskEXIT_CRITICAL()`

- **Suspendig the scheduler** – when waiting on interrupt and no task switching allowed. Function `vTaskSuspendAll()` block context switching with interrupts enabled. Unblock the tasks is done by `xTaskResumeAll()` function.

  It is not allowed to run any FreeRTOS API function when scheduler is suspended.

- **Gatekeeper task**
  - Dedicated procedure managing selected resource (i.e. peripheral)
  - No risk of priority inversion and deadlock
  - It has ownership of a resource and can access it directly
  - Other tasks can access protected resource indirectly via gatekeeper task
  - Example: standard out access

- **Mutexes**
  - Kind of binary semaphore shared between tasks
  - Require set `configUSE_MUTEXES` at 1 in *FreeRTOSConfig.h*

life.augmented

Critical sections

- Critical section mechanism allows to block all the interrupts during sensitive/atomic operation execution (like operations on queues)
- To enter into critical section `portENTER_CRITICAL()` should be used
- To exit from critical section `portEXIT_CRITICAL()` should be used

```
void function(void)
{
          portENTER_CRITICAL();

          ..sensitive code
execution

          portEXIT_CRITICAL();
}
```

Storing current BASEPRI value (current level of interrupt masking) and programming BASEPRI with MASK value *)
It means masking of all interrupts below this value

restoring BASEPRI value and masking of all interrupts below this value

*) MASK = configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY<<(8- configPRIO_BITS)

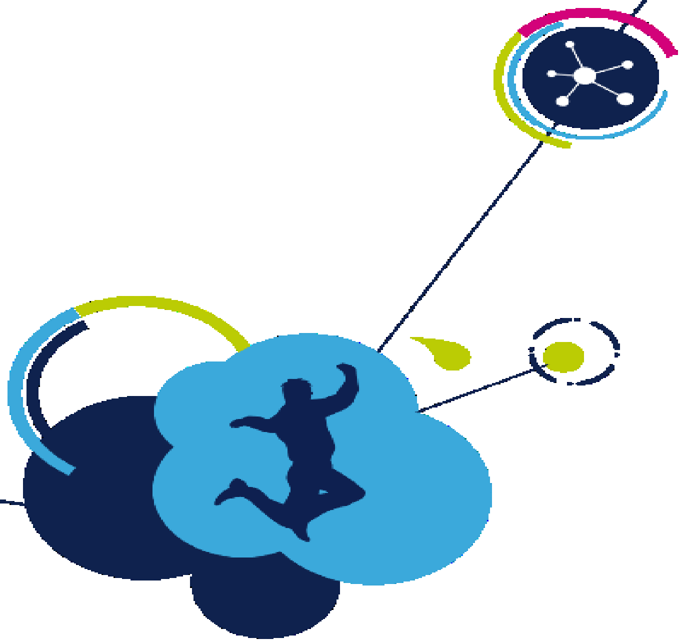
defined in FreeRTOSConfig.h

= 4 for CortexM3, CortexM4 based STM32

life.augmented

Gatekeeper task

- Gatekeeper is a task being the only allowed to access certain resources (i.e. peripheral).
  - It owns selected resource and only it can access it directly; other tasks can do it indirectly by using services provided by the gatekeeper task.
    - There is nothing physically preventing other tasks from accessing the resource → it is on the designer side to program it proper way
  - It is providing clean method to implement mutual exclusion without risk of priority inversion or deadlock.
  - It spends most of the time in the blocked state waiting for the requests on the owned resources
  - It is up to the designer to set the priority of the gatekeeper and its name.



FreeRTOS
Mutex

Mutex 1/2

- Mutex is a binary semaphore that include a priority inheritance mechanism.
  - binary semaphore is the better choice for implementing synchronization (between tasks or between tasks and an interrupt),
  - mutex is the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

- When used for mutual exclusion the mutex acts like a token that is used to guard a resource.
  - When a task wishes to access the resource it must first obtain ('take') the token.
  - When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.
  - In case of **recursive mutex** it should be given as many times as it was successfully taken (like counting semaphores) to release it for another task.

Mutex 2/2

- Mutexes use the same access API functions as semaphores – this permits a block time to be specified.
- The block time indicates the maximum number of 'ticks' that a task should enter the Blocked state when attempting to 'take' a mutex if the mutex is not available immediately.
- Unlike binary semaphores however - mutexes employ priority inheritance. This means that if a high priority task is blocked while attempting to obtain a mutex (token) that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocked task.
- Mutex Management functions cannot be called from interrupt service routines (ISR).
- A task must not be deleted while it is controlling a Mutex. Otherwise, the Mutex resource will be locked out to all other tasks

Mutex, Semaphore – threats 1/3

Priority inversion

- This is the situation where a higher priority task is waiting for a lower priority task to give a control of the mutex and low priority task is not able to execute.

Mutex, Semaphore – threats 2/3

Priority inheritance

- It is temporary raise of the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex. The low priority task that holds the mutex inherits the priority of the task waiting for the mutex. The priority of the mutex holder is reset automatically to its original value when it gives the mutex back.
- It is a mechanism that minimizes the negative effects of priority inversion
- It is complicating system timing analysis and it is not a good practice to rely on it for correct system operation

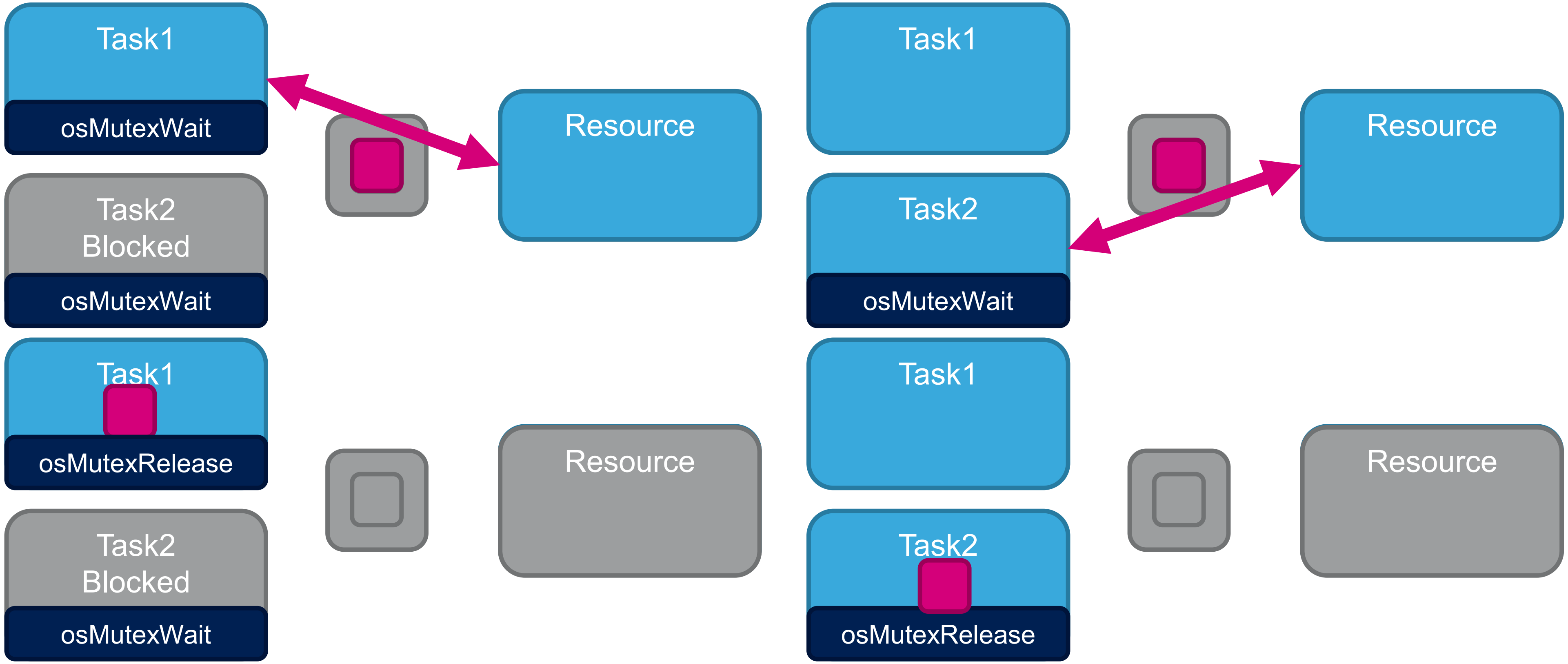
Mutex, Semaphore – threats 3/3

Deadlock (Deadly Embrace)

- It occurs when two tasks cannot work because they are both waiting for a resource held by each other
- The best way to avoid deadlock is to consider them at design time and design the system to be sure that the deadlock cannot occur.

Mutex

- Used to guard access to limited recourses
- Works very similar as semaphores



configUSE_MUTEXES

Mutex

- Mutex creation

```
osMutexId osMutexCreate (const osMutexDef_t *mutex_def)
```

osMutexId: Mutex handle

const osMutexDef_t *mutex_def: Mutex definition

- Wait for Mutex release

```
osStatus osMutexWait (osMutexId mutex_id, uint32_t millisec)
```

osStatus: Return status

osMutexId mutex_id: Mutex handle

uint32_t millisec: How long wait for mutex release

- Mutex release

```
osStatus osMutexRelease (osMutexId mutex_id)
```

osStatus: Return status

osMutexId mutex_id: Mutex handle

life.augmented

Recursive mutex

- Recursive mutex creation

```
osMutexId osRecursiveMutexCreate (const osMutexDef_t *mutex_def)
```

osMutexId: Mutex handle
const osMutexDef_t *mutex_def: Mutex definition

- Wait for Recursive mutex release

```
osStatus osRecursiveMutexWait (osMutexId mutex_id, uint32_t millisec)
```

osStatus: Return status
osMutexId mutex_id: Mutex handle
uint32_t millisec: How long wait for mutex release

- Recursive mutex release

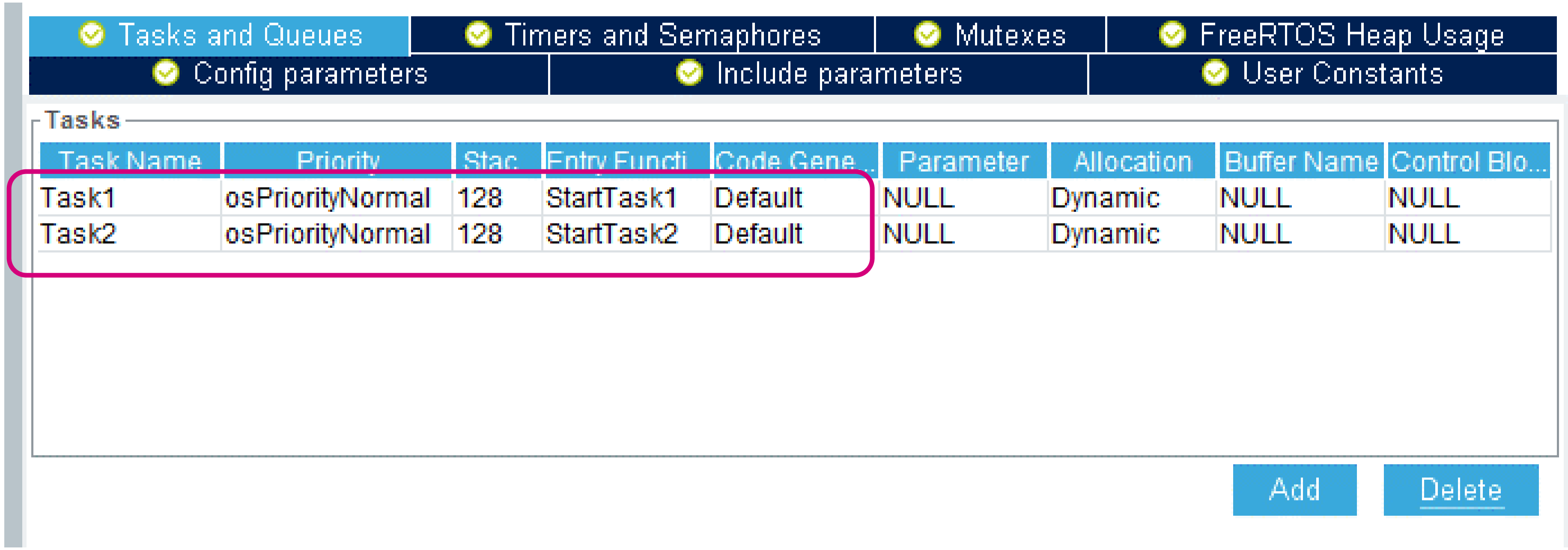
```
osStatus osRecursiveMutexRelease (osMutexId mutex_id)
```

osStatus: Return status
osMutexId mutex_id: Mutex handle

Mutex lab

Create two tasks: Task1, Task2 with same priorities

- Click Add button in Tasks section
- Set parameters (entry functions, stack size)
- Click OK button

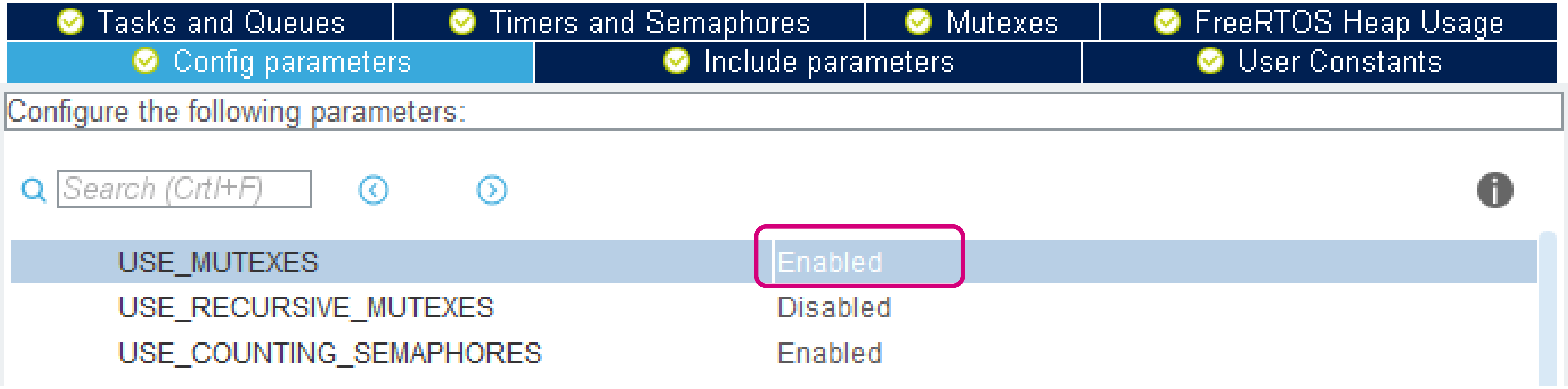


configUSE_MUTEXES

Mutex lab

Enable Counting semaphore

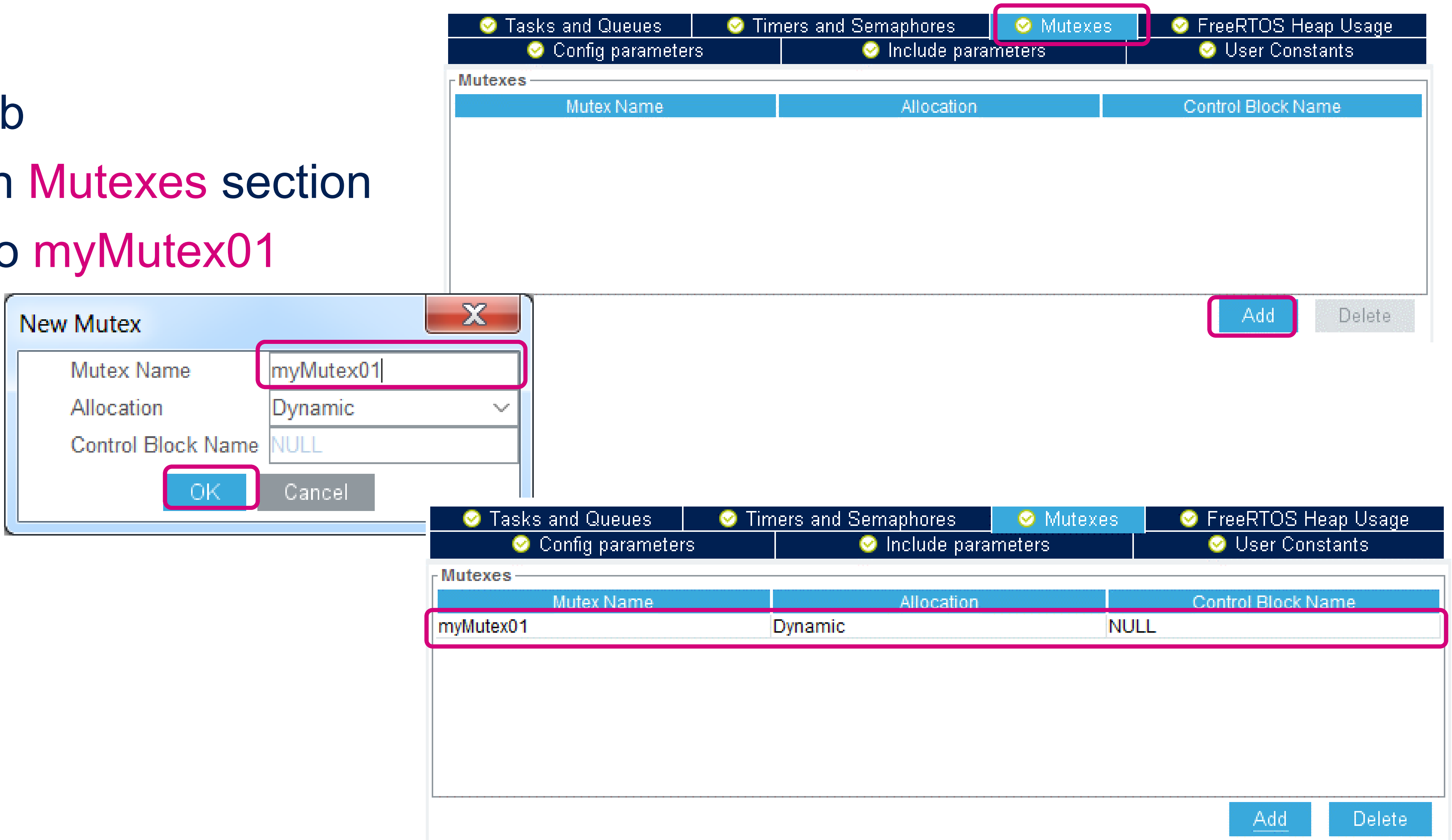
1. Select Config parameters tab
2. Change “USE_MUTEXES” to **Enabled**



Mutex lab

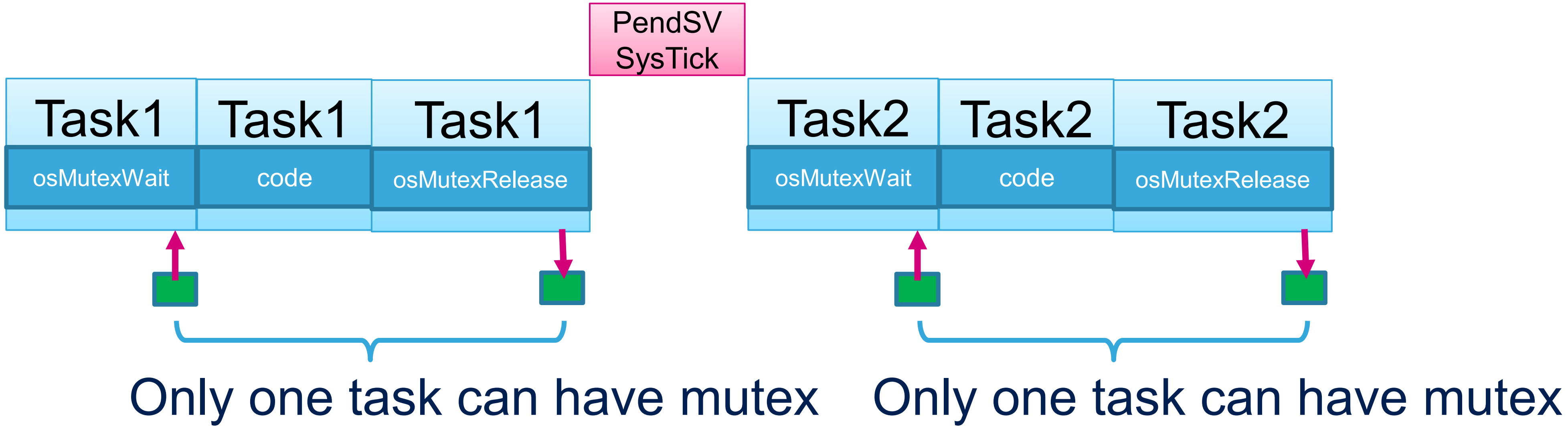
Add Mutex

- Select Mutexes tab
- Click Add button in Mutexes section
- Set Mutex name to myMutex01
- Click OK button



Mutex lab

- Both tasks use printf function.
- Mutex is used to avoid collisions



printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

Mutex lab

- Mutex handle definition

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
osThreadId Task2Handle;
osMutexId myMutex01Handle;
```

- Mutex creation

```
/* Create the mutex(es) */
 /* definition and creation of myMutex01 */
 osMutexDef(myMutex01);
 myMutex01Handle = osMutexCreate(osMutex(myMutex01));
```

Mutex lab

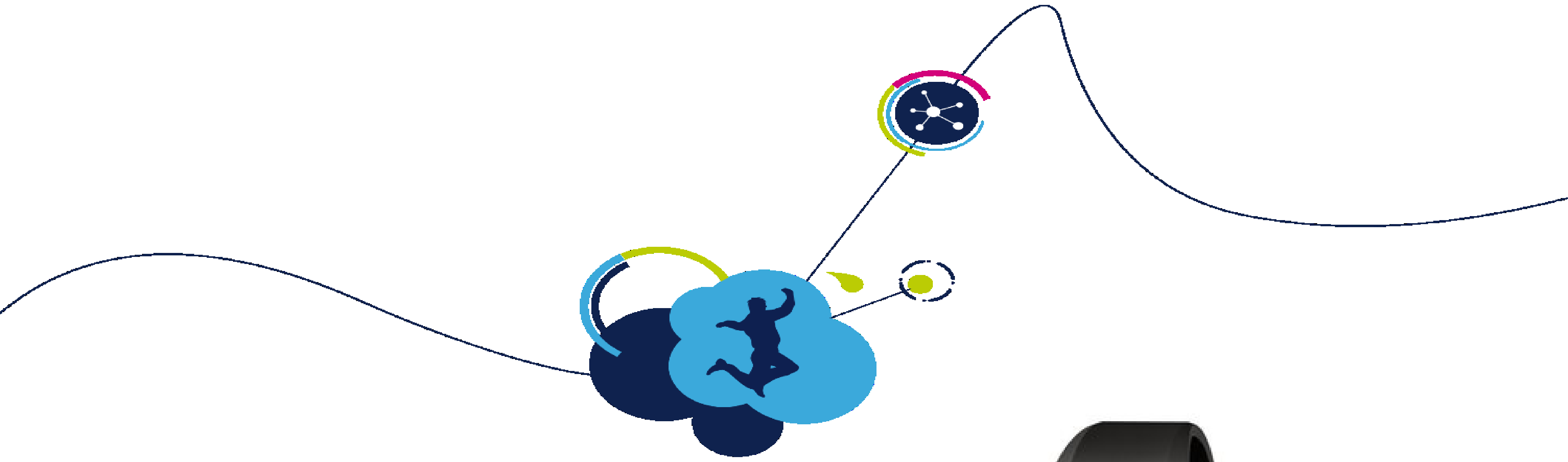
- Task1 and Task2 using of Mutex

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(2000);
    osMutexWait(myMutex01Handle,1000);
    printf("Task1 Print\n");
    osMutexRelease(myMutex01Handle);
  }
  /* USER CODE END 5 */
}
```

```
void StartTask2(void const * argument)
{
  /* USER CODE BEGIN StartTask2 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(2000);
    osMutexWait(myMutex01Handle,1000);
    printf("Task2 Print\n");
    osMutexRelease(myMutex01Handle);
  }
  /* USER CODE END StartTask2 */
}
```

Mutex APIs

CMSIS_RTOS API	FreeRTOS API
osMutexCreate()	xSemaphoreCreateMutexStatic() xSemaphoreCreateMutex()
osMutexRelease()	xSemaphoreGive() xSemaphoreGiveFromISR()
osMutexWait()	xSemaphoreTake() xSemaphoreTakeFromISR()
osMutexDelete()	vQueueDelete()
osRecursiveMutexCreate()	xSemaphoreCreateRecursiveMutexStatic() xSemaphoreCreateRecursiveMutex()
osRecursiveMutexRelease()	xSemaphoreGiveRecursive()
osRecursiveMutexWait()	xSemaphoreTakeRecursive()



FreeRTOS
Software Timers

life.augmented

Software Timers (1/3)

- Software timer is one of standard component of every RTOS
- FreeRTOS “software” Timers allows to execute a callback at a set of time (timer period). Timer callback functions execute in the context of the timer service task.
- It is therefore essential that timer callback functions never attempt to block. For example, a timer callback function must not call vTaskDelay(), vTaskDelayUntil(), or specify a non zero block time when accessing a queue or a semaphore.

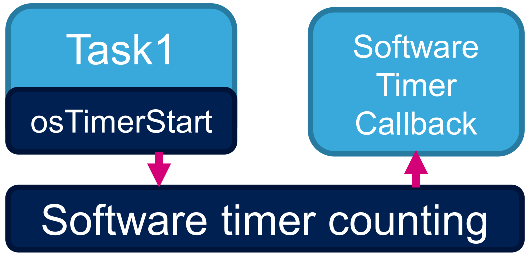
life.augmented

Software Timers (2/3)

- It is not precise, intended to handle periodic actions and delay generation
  - Can be conditionally used to extend number of hardware timers in STM32

- Two types od software timers are available:
  - **Periodic** (execute its callback periodically with autoreload functionality)

  - **One Pulse** (execute its callback only once with an option of manual re-trigger)



Software Timers (3/3)

- When Timers are enabled (configUSE_TIMERS enabled) , the scheduler creates automatically the timers service task (daemon) when started (calling `xTimerCreateTimerTask()` function).
- The timers service task is used to control and monitor (internally) all timers that the user will create.
- The timers task parameters are set through the fowling defines (in FreeRTOSConfig.h):
  - configTIMER_TASK_PRIORITY : priority of the timers task
  - configTIMER_TASK_STACK_DEPTH : timers task stack size (in words)
- The scheduler also creates automatically a message queue used to send commands to the timers task (timer start, timer stop ...) .
- The number of elements of this queue (number of messages that can be hold) are configurable through the define:
  - configTIMER_QUEUE_LENGTH.

Software Timers

configuration

- Software timer is one of standard component of every RTOS

Config field (default value)	description
configUSE_TIMERS (0 – disabled)	1 – includes software timers functionality and automatically creates timer service task on scheduler start 0 – disabled, no timer service task
configTIMER_TASK_PRIORITY ()	Priority for timer service task from the range between IDLE task priority and configMAX_PRIORITIES-1
configTIMER_QUEUE_LENGTH ()	This sets the maximum number of unprocessed commands that the timer command queue can hold at any one time.
configTIMER_TASK_STACK_DEPTH ()	Sets the size of the stack (in words, not bytes) allocated to the timer service task.

Software Timers

- Software timer creation

```
osTimerId osTimerCreate (const osTimerDef_t *timer_def, os_timer_type type, void *argument)
```

Timer handle

Timer definition

Repeat timing or onetime timing

- Software timer start

```
osStatus osTimerStart (osTimerId timer_id, uint32_t millisec)
```

Return status

Timer handle

Timer period

- Software timer stop

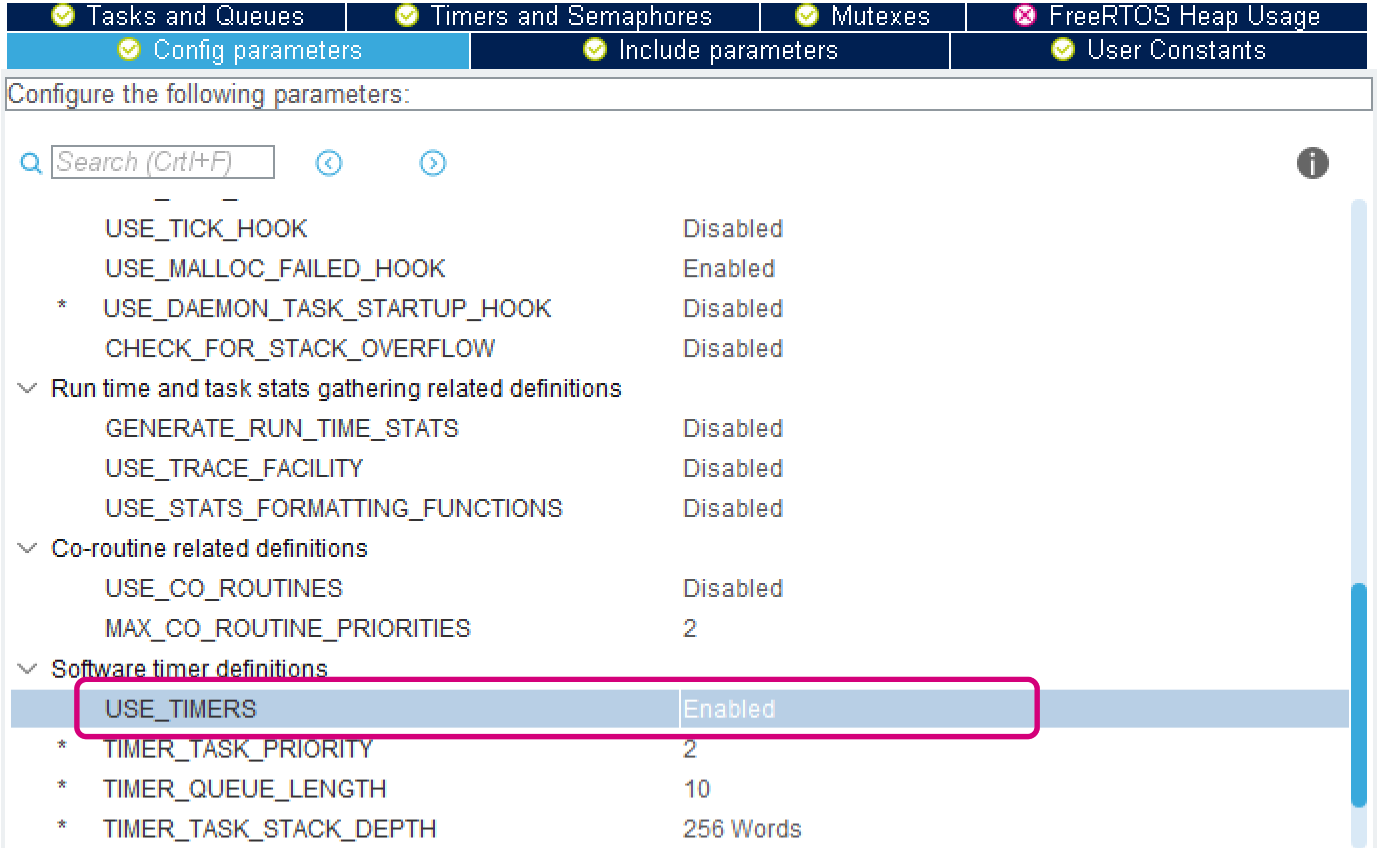
```
osStatus osTimerStop (osTimerId timer_id)
```

Return status

Timer handle

Software Timers lab

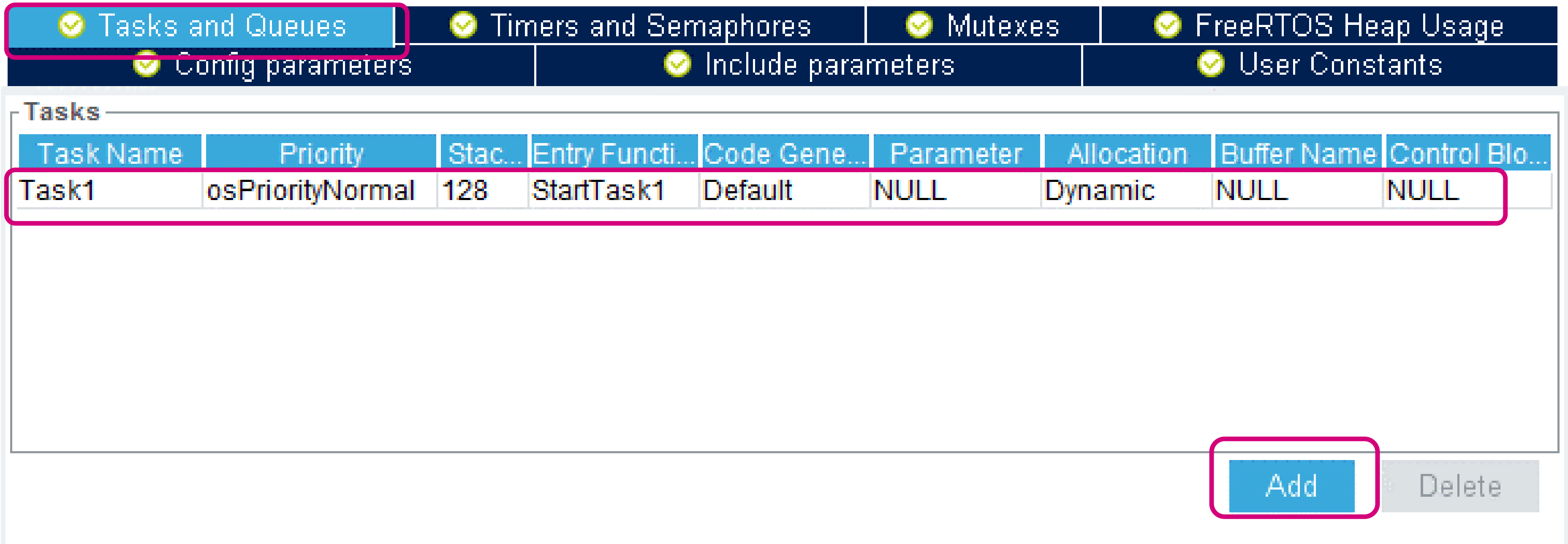
- Software timers are disabled by default in STM32CubeMX
- To enable them:
  - Select Config parameters tab
  - Set USE_TIMERS value to Enabled
  - Other software timers parameters we will keep in default configuration



life.augmented

Software Timers lab

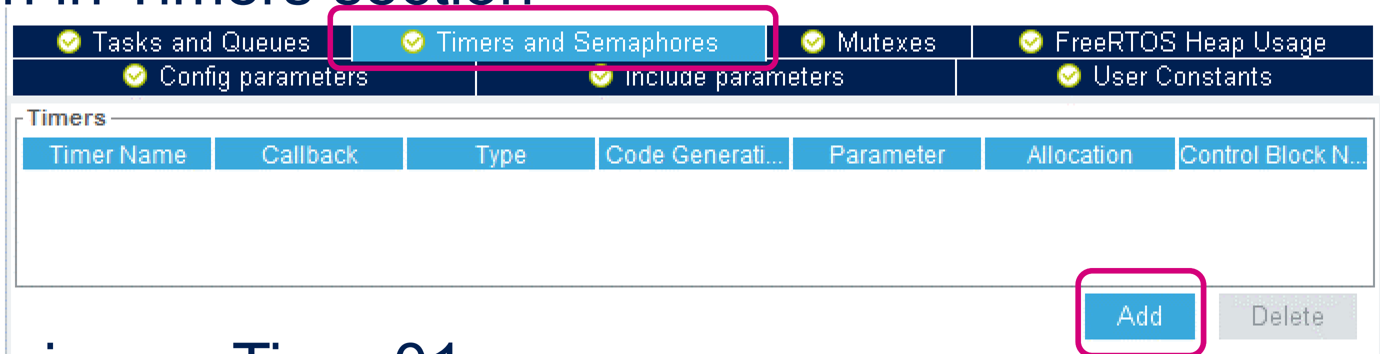
- Create one task, Task1 with entry function StartTask1 and normal priority



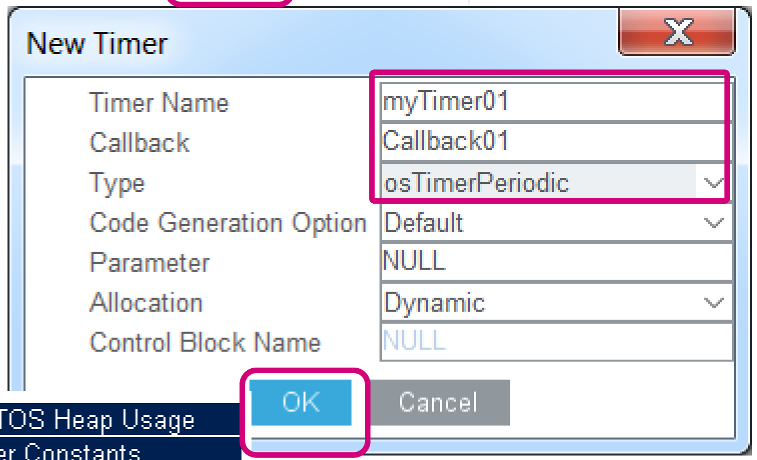
Software Timers lab

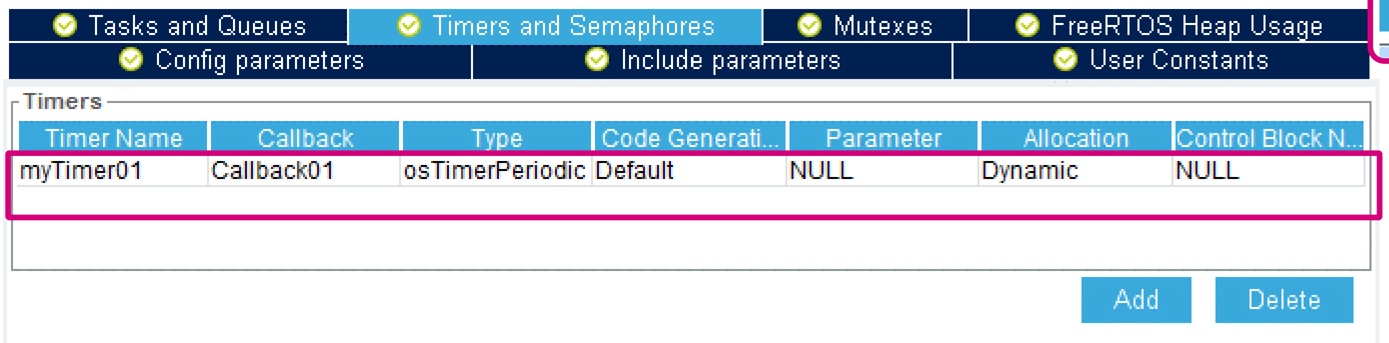
Create a new timer

- Select Timers and Semaphores tab
- Click Add button in Timers section



- Set timer name: i.e. myTimer01
- Timer callback name: i.e. Callback01
- Type: Periodic
- Click OK button





life.augmented

printf redirection to USART2

- The following code should be included into ***main.c*** file to redirect printf output stream to UART2

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len)
{
  HAL_UART_Transmit(&huart2,(uint8_t *)ptr,len,10);
  return len;
}
/* USER CODE END 0 */
```

Software Timers lab

- Software timer handle definition

```
/* Private variables ---------------------------------------------------------*/
osThreadId Task1Handle;
osTimerId myTimer01Handle;
```

- Software timer creation

```
 /* Create the timer(s) */
  /* definition and creation of myTimer01 */
  osTimerDef(myTimer01, Callback01);
  myTimer01Handle = osTimerCreate(osTimer(myTimer01), osTimerPeriodic, NULL);
```

- Software timer start

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  osTimerStart(myTimer01Handle,1000);
  /* Infinite loop */
  for(;;)
  {
    osDelay(2000);
    printf("Task1 Print\n");
  }
  /* USER CODE END 5 */
}
```

Software Timers lab

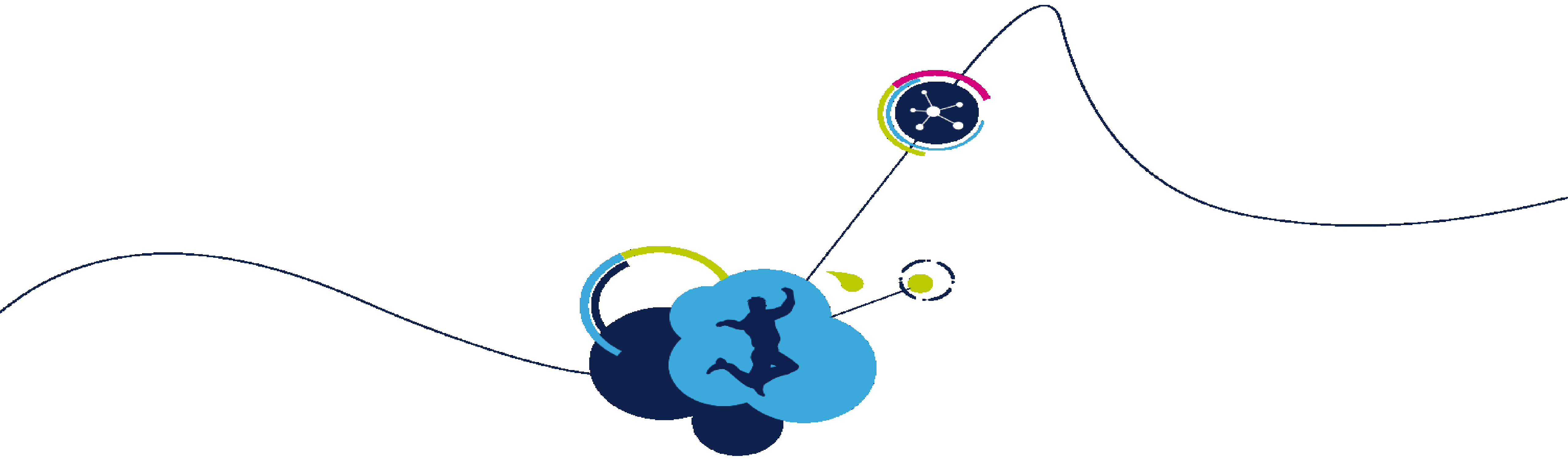
- Timer callback functions execute in the context of the timer service task.
- Timer callbacks are not called from interrupt context.
- There should be no blocking functions inside (like in hooks)

```c
/* Callback01 function */
void Callback01(void const * argument)
{
  /* USER CODE BEGIN Callback01 */
  printf("Timer Print\n");
  /* USER CODE END Callback01 */
}
```

```c
/* Callback01 function */
void Callback01(void const * argument)
{
  /* USER CODE BEGIN Callback01 */
  printf("Timer Print\n");
  osDelay(100);
  /* USER CODE END Callback01 */
}
```

Software Timers APIs

CMSIS_RTOS API	FreeRTOS API
osTimerCreate()	xTimerCreateStatic() xTimerCreate()
osTimerStart()	xTimerChangePeriod() xTimerChangePeriodFromISR()
osTimerStop()	xTimerStop() xTimerStopFromISR()
osTimerDelete()	xTimerDelete()
-	xTimerGetTimerDaemonTaskHandle()
-	xTimerGetPeriod()
-	xTimerGetExpiryTime()
-	pcTimerGetName()
-	xTimerGenericCommand()



FreeRTOS advanced topics

life.augmented

Hooks

- Hooks are the callbacks supported by FreeRTOS core
- Those can help with FreeRTOS fault handling
- Type of hooks:
  - Idle Hook
  - Tick Hook
  - Malloc Failed Hook
  - Stack Overflow Hook
- STM32CubeMX creates hook functions in freertos.c file

Idle task and "idle task hook"

- Idle task is automatically created by scheduler within **osKernelStart()** function
- It has the lowest possible priority
- It runs only if there are no tasks in ready state
- It can share same priority with other tasks
- Specific function (called idle task hook function) can be called automatically from idle task. Its prototype is strictly defined:
  - **void vApplicationIdleHook(void); // [weak] version in freertos.c file**
  - **configUSE_IDLE_HOOK** must be set to **1** in FreeRTOSConfig.h to get it called
  - it must never attempt to block or suspend
  - it is responsible to cleanup resources after deletion of other task
  - it is executed every iteration of the idle task loop, do not put any endless loop inside

```
void vApplicationIdleHook(void)
{
		tick_IDLE++;
}
```

Correct

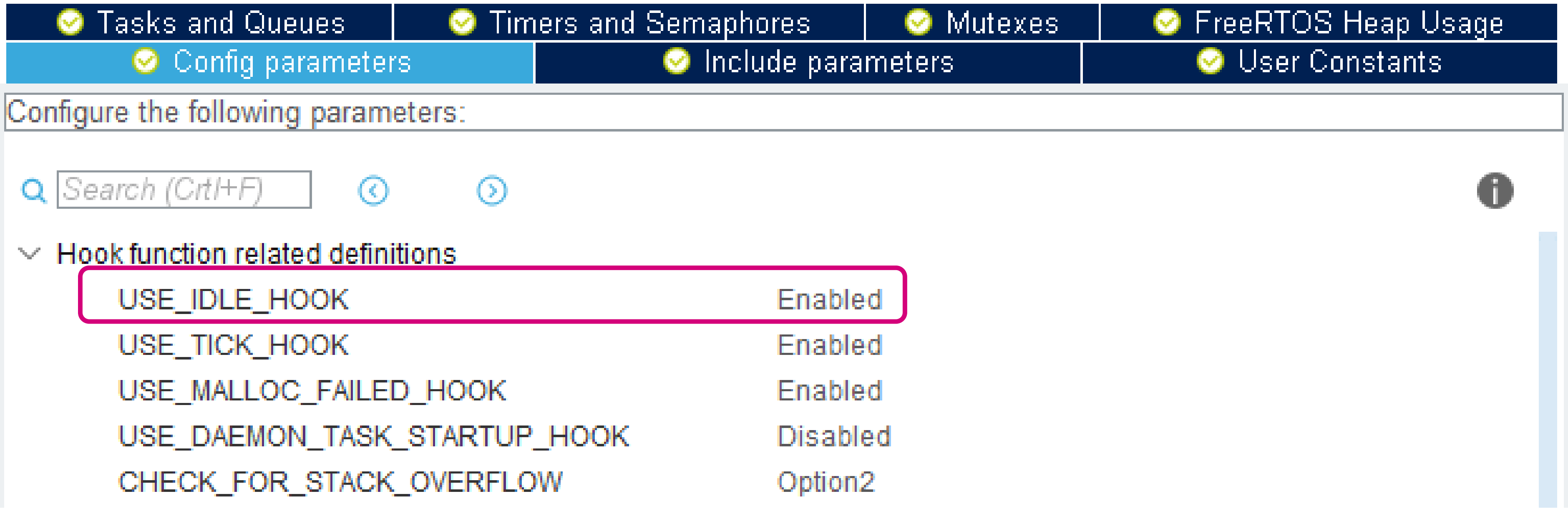
```
void vApplicationIdleHook(void)
{
		while(1)
		{
				tick_IDLE++;
		}
}
```

Wrong

configUSE_IDLE_HOOK

Idle Hook

- If the scheduler cannot run any task it goes into idle mode
- Idle hook is callback from idle mode
- Within this task is possible to put power saving function
- It is necessary to enable it within Config parameters (part of FreeRTOSConfig.h configuration file)



Idle Hook

- Idle hook callback in freertos.c created by STM32CubeMX

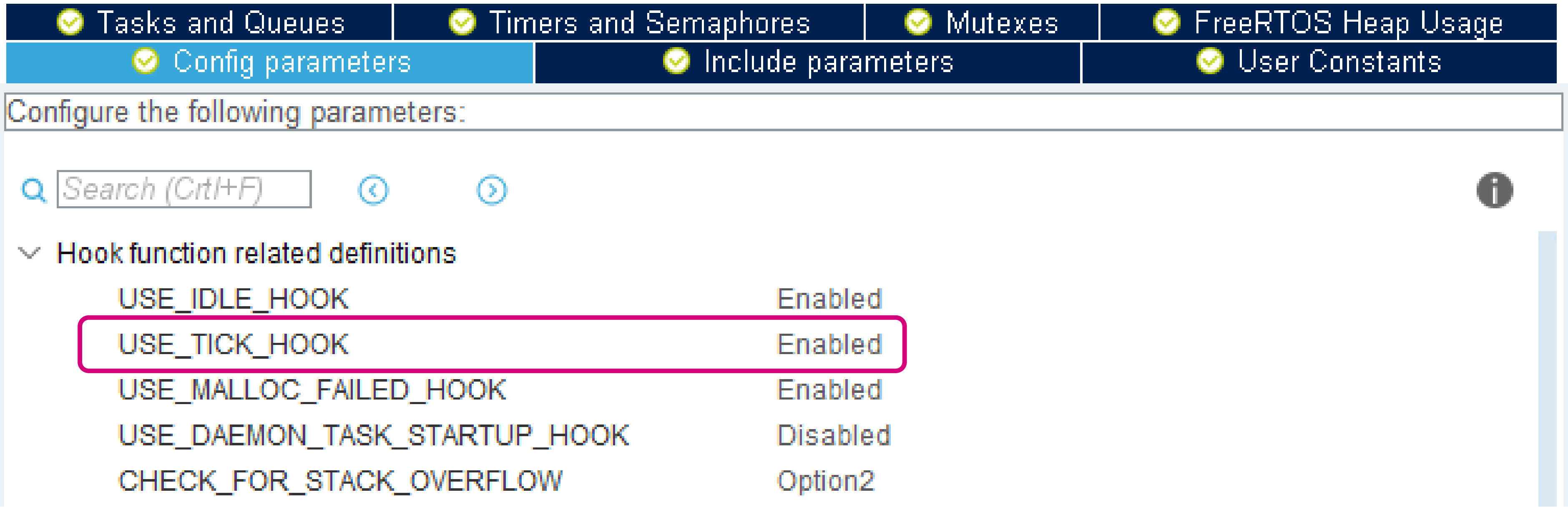
```
/* USER CODE END FunctionPrototypes */
/* Hook prototypes */
void vApplicationIdleHook(void);

/* USER CODE BEGIN 2 */
__weak void vApplicationIdleHook( void )
{
   /* vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set
   to 1 in FreeRTOSConfig.h. It will be called on each iteration of the idle
   task. It is essential that code added to this hook function never attempts
   to block in any way (for example, call xQueueReceive() with a block time
   specified, or call vTaskDelay()). If the application makes use of the
   vTaskDelete() API function (as this demo application does) then it is also
   important that vApplicationIdleHook() is permitted to return to its calling
   function, because it is the responsibility of the idle task to clean up
   memory allocated by the kernel to any task that has since been deleted. */
}
/* USER CODE END 2 */
```

- Do not use blocking functions (`osDelay()`, …) in this function or while(1)

Tick Hook

- Every time the SysTick interrupt is trigger the TickHook is called
- Is possible use TickHook for periodic events like watchdog refresh
- It is necessary to enable it within Config parameters (part of FreeRTOSConfig.h configuration file)



configUSE_TICK_HOOK

Tick Hook

- Tick hook callback in freertos.c created by STM32CubeMX

```
/* Hook prototypes */
void vApplicationTickHook(void);

/* USER CODE BEGIN 3 */
__weak void vApplicationTickHook( void )
{
   /* This function will be called by each tick interrupt if
   configUSE_TICK_HOOK is set to 1 in FreeRTOSConfig.h. User code can be
   added here, but the tick hook is called from an interrupt context, so
   code must not attempt to block, and only the interrupt safe FreeRTOS API
   functions can be used (those that end in FromISR()). */
}
/* USER CODE END 3 */
```

- Do not use blocking functions (`osDelay`, …) in this function or while(1)
- Use only the interrupt safe FreeRTOS functions (with suffix FromISR()).

life.augmented

Memory management models - monitoring

Malloc Failed Hook Function

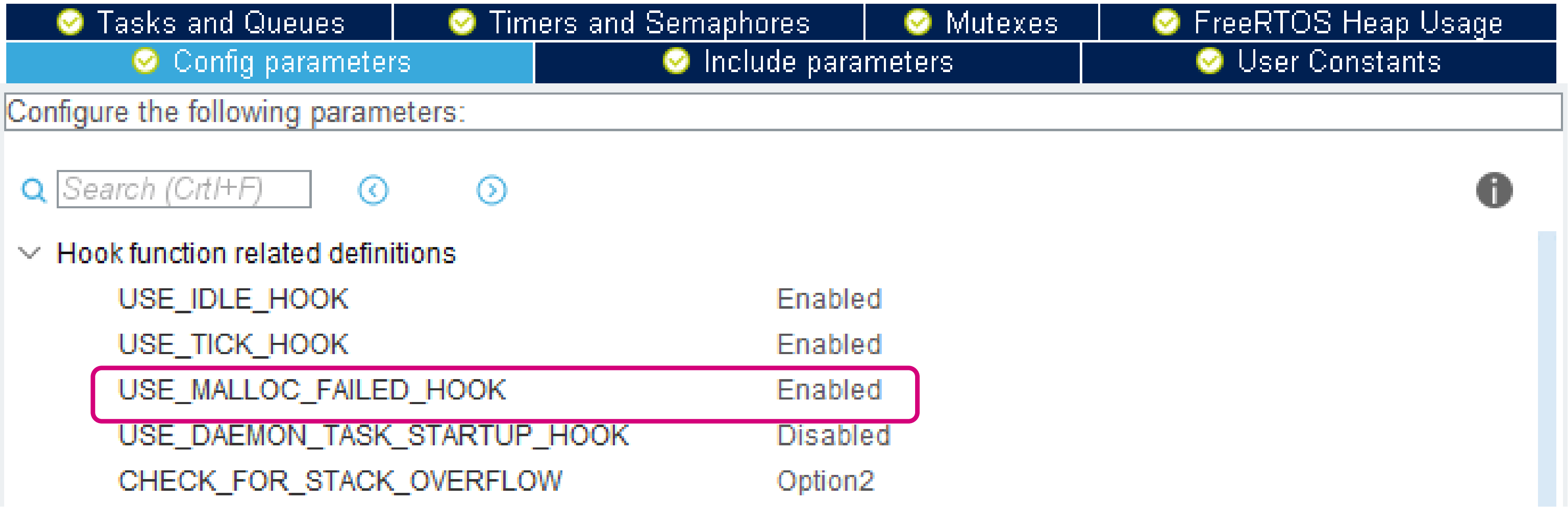
- Memory allocation schemes implemented by heap_1.c, heap_2.c, heap_3.c, and heap_4 and heap_5.c can optionally include **`malloc()`** failure hook (or callback) function that can be configured to get called on **`pvPortMalloc()`** returning NULL.
- Defining **`malloc()`** failure hook will help to identify problems caused by lack of heap memory; especially when call to **`pvPortMalloc()`** fails within an API function.
- Malloc failed hook will only get called if **configUSE_MALLOC_FAILED_HOOK** is set to 1 in FreeRTOSConfig.h. When it is set, an application must provide hook function with the following prototype:

```
void vApplicationMallocFailedHook( void )
```

life.augmented

Malloc Failed Hook

- This callback is called if the memory allocation process fails (pvPortMalloc() returns NULL)
- Helps to react on malloc problems, when function return is not handled
- It is necessary to enable it within **Config parameters** (part of FreeRTOSConfig.h configuration file)



Malloc Failed Hook

- Malloc Failed hook callback skeleton is present in freertos.c created by STM32CubeMX

```
/* Hook prototypes */
void vApplicationMallocFailedHook(void);

/* USER CODE BEGIN 5 */
__weak void vApplicationMallocFailedHook(void)
{
   /* vApplicationMallocFailedHook() will only be called if
   configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h. It is a hook
   function that will get called if a call to pvPortMalloc() fails.
   pvPortMalloc() is called internally by the kernel whenever a task, queue,
   timer or semaphore is created. It is also called by various parts of the
   demo application. If heap_1.c or heap_2.c are used, then the size of the
   heap available to pvPortMalloc() is defined by configTOTAL_HEAP_SIZE in
   FreeRTOSConfig.h, and the xPortGetFreeHeapSize() API function can be used
   to query the size of free heap space that remains (although it does not
   provide information on how the remaining heap might be fragmented). */
}
/* USER CODE END 5 */
```

- Do not use blocking functions (`osDelay()` , …) in this function or while(1) configUSE_MALLOC_FAILED_HOOK

Malloc Failed Hook

- Let’s try to implement and test Malloc Failed hook mechanism
- Simple example of Malloc Failed hook (main.c):

```
/* USER CODE BEGIN 5 */
void vApplicationMallocFailedHook(void)
{
  printf("malloc fails\n");
}
/* USER CODE END 5 */
```

- Do impossible memory allocation within one of our tasks

```
/* Private variables -------------*/
osThreadId Task1Handle;
osPoolId PoolHandle;
```

```
void StartTask1(void const * argument)
{
  /* USER CODE BEGIN 5 */
  osPoolDef(Memory,0x10000000,uint8_t);
  /* Infinite loop */
  for(;;)
  {
    PoolHandle = osPoolCreate(osPool(Memory));
    osDelay(5000);
  }
  /* USER CODE END 5 */
}
```

Impossible memory allocation

life.augmented

Stack overflow protection

check of stack ‘high watermark

- During task creation, its stack memory space is filled with 0xA5 data
- During run time we can check how much stack is used by task – stack ‘high water mark’
- To turn on this mechanism, some additional configuration of FreeRTOS is required (*FreeRTOSConfig.h* file or STM32CubeMX FreeRTOS configuration window):
  - `configUSE_TRACE_FACILITY` should be defined to 1
  - `INCLUDE_uxTaskGetStackHighWaterMark` should be defined to 1
- There is a dedicated function to perform this operation:

```
uxTaskGetStackHighWaterMark(xTaskHandle xTask);
```

- After call it with task handle as an argument returns the minimum amount of remaining stack for xTask is presented (NULL means task which is currently in RUN mode).
- Additional configuration within *FreeRTOSConfig.h* is required

life.augmented

Stack overflow protection

runtime stack check mechanism

Stack Overflow Detection - Option 1

- Stack can reach its deepest value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point RTOS kernel can check whether processor stack pointer remains within valid stack space. Stack overflow hook function is called, if the stack pointer contains value outside of the valid stack range.
- This method is quick but it can't guarantee catching all stack overflows.
  To use this option only set **configCHECK_FOR_STACK_OVERFLOW** to 1.

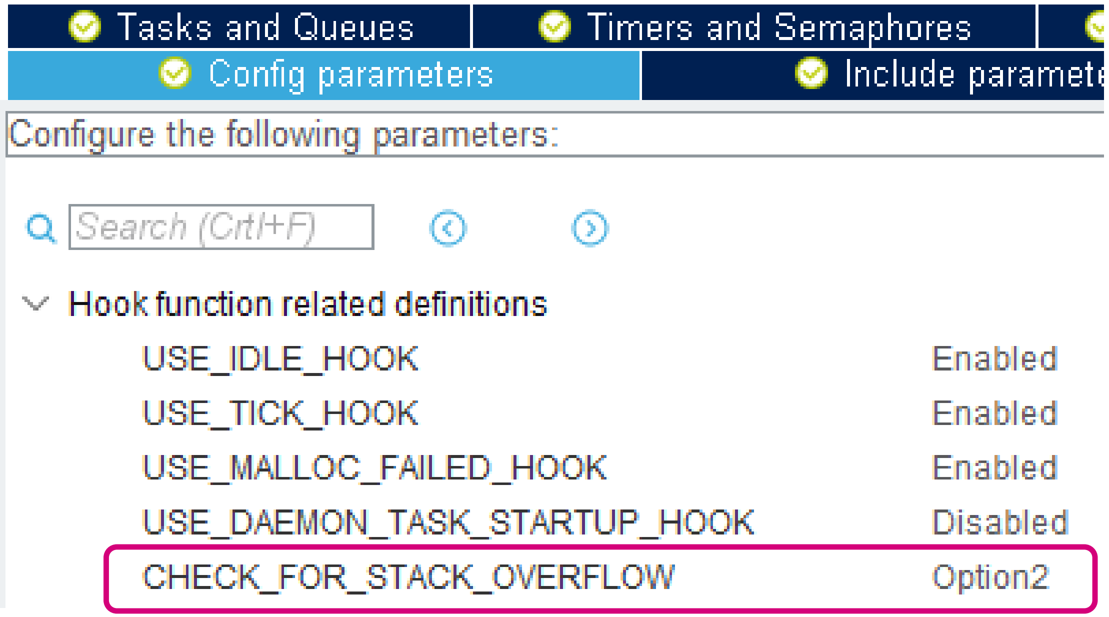
Stack Overflow Detection - Option 2

- When task is first created, its stack is filled with a known value. When swapping task out of the Running state, RTOS kernel can check last 16 bytes within valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. Stack overflow hook function is called should any of these 16 bytes not remain at their initial value.
- This method is less efficient than method one, but still fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.
- To use this method in combination with option 1 set **configCHECK_FOR_STACK_OVERFLOW** to 2 (this is not possible to use only this option).

Stack overflow protection

runtime stack check mechanism in STM32CubeMX

- FreeRTOS is able to check stack against overflow
- Two options are available (to be configured within Config parameters (FreeRTOSConfig.h file):
  - Option 1
  - Option 2



Stack overflow protection

stack overflow hook implementation

- Stack overflow hook function is a function called by the kernel at detected stack overflow
- It should be implemented by the user. Its declaration should look like:
- **vApplicationStackOverflowHook(xTaskHandle *pxTask, signed char *pcName);**
- Its skeleton is generated by STM32CubeMX in freertos.c file

```
/* Hook prototypes */
void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName);

/* USER CODE BEGIN 4 */
__weak void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName)
{
   /* Run time stack overflow checking is performed if
   configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2. This hook function is
   called if a stack overflow is detected. */
}
/* USER CODE END 4 */
```

- Do not use blocking functions (`osDelay()` , …) or while(1) in this function

life.augmented

Statistics

- To collect runtime statistics of OS components, there is dedicated function:

  `osThreadList()`

- This function is calling **`vTaskList()`** within FreeRTOS API and is collecting information about all tasks and put them to the table
- Function triggering and data formatting should be implemented by the user
- To run this function you need to set two definitions (define its values to 1):
  - **configUSE_TRACE_FACILITY**
  - **configUSE_STATS_FORMATTING_FUNCTIONS** → it should be added manually to FreeRTOSConfig.h or within STM32CubeMX configuration window for FreeRTOS

FreeRTOS – debug support

TrueStudio

life.augmented

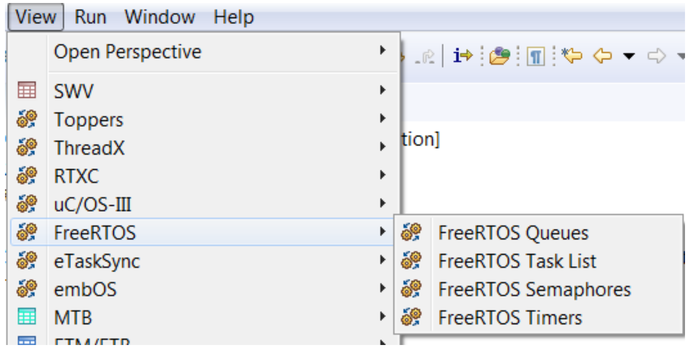
FreeRTOS debug support

STLink and JLink

- True Studio provides a FreeRTOS plugin that can be used to display a snapshot of tasks, queues, semaphores and timers each time the debugger is paused or single stepped.

- It can be enabled within debug session.

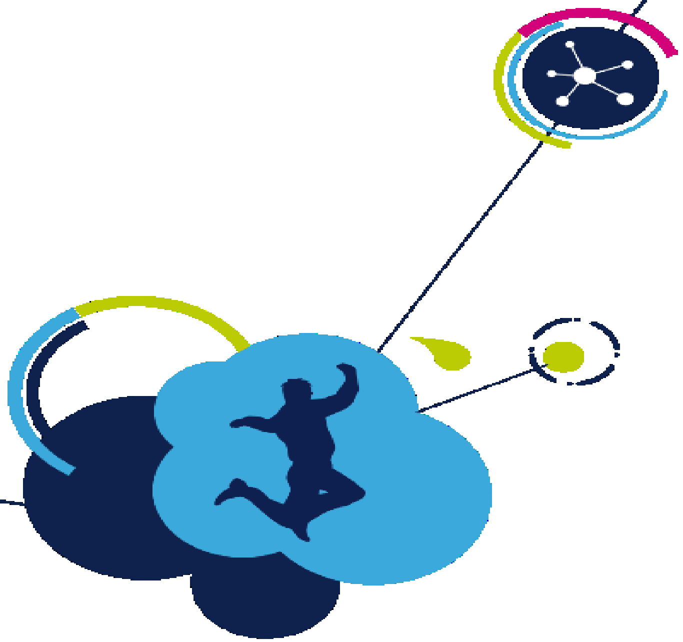
  View ->FreeRTOS



- After selection of i.e. FreeRTOS Task List there will be an additional window present, then after run and pause of the code, the list of task till be displayed

Console | FreeRTOS Task List

	Name	Priority (Bas...	Start of Stack	Top of Stack	State	Event Object	Min Free St...	Run Time (%)
⇨	Gyro_Task	N/A/3	0x20000098	0x20000234	RUNNING		Disabled	>99%
	IDLE	N/A/0	0x20000b80	0x20000d34	READY		Disabled	0%
	LCD_Task	N/A/3	0x20000300	0x2000041c	BLOCKED	0x20000b34	Disabled	0%
	LED_Task	N/A/3	0x200007d0	0x200008ec	BLOCKED	0x20000ad4	Disabled	<1%
	UART_Task	N/A/3	0x20000568	0x20000684	BLOCKED	0x20000a5c	Disabled	0%



FreeRTOS
low power modes

FreeRTOS and low power modes

Tickless idle mode operation

- Kernel can stop tick interrupt and place MCU in low power mode, on exit from this mode tick counter is updated
- Enabled when setting configUSE_TICKLESS_IDLE as 1
- The kernel will call a macro (tasks.c) `portSUPPRESS_TICKS_AND_SLEEP()` when the Idle task is the only task able to run (and no other task is scheduled to exit from blocked state after n* ticks)
- FreeRTOS implementation of `portSUPRESS_TICKS_AND_SLEEP` for cortexM3/M4 enters MCU in sleep low power mode
- Wakeup from sleep mode can be from a system interrupt/event
- User implementation can be done by setting configUSE_TICKLESS_IDLE above 1 (to avoid usage of kernel macros)
- Lowest power consumption can be achieved by replacing default SysTick by LowPower timers (LPTIM or RTC) as tick timer

*) n value is defined in FreeRTOS.h file

Idle task code

- Idle task code is generated automatically when the scheduler is started
- It is portTASK_FUNCTION() function within task.c file
- It is performing the following operations (in endless loop):
  - Check for deleted tasks to clean the memory
  - taskYIELD() if we are not using preemption (configUSE_PREEMPTION=0)
  - Get yield if there is another task waiting and we set configIDLE_SHOULD_YIELD=1
  - Executes vApplicationIdleHook() if configUSE_IDLE_HOOK=1
  - Perform low power entrance if configUSE_TICKLESS_IDLE!=0) -> let’s look closer on this

Perform low power entrance

idle task code

- Check expected idle time and if it is bigger than configEXPECTED_IDLE_TIME_BEFORE_SLEEP (set to 2 in FreeRTOS.h) then continue
- Suspend all tasks (stop scheduler)
- Check again expected idle time by prvGetExpectedIdleTime()
- execute configPRE_SUPPRESS_TICKS_AND_SLEEP_PROCESSING with expected idle time and if is bigger than configEXPECTED_IDLE_TIME_BEFORE_SLEEP (set to 2 in FreeRTOS.h) then continue
- Execute portSUPPRESS_TICKS_AND_SLEEP() with expected idle time and enter into low power mode

Low power mode

- Wakeup from low power mode and resume all tasks (start scheduler)

Perform low power entrance

configPRE_SUPPRESS_TICKS_AND_SLEEP_PROCESSING

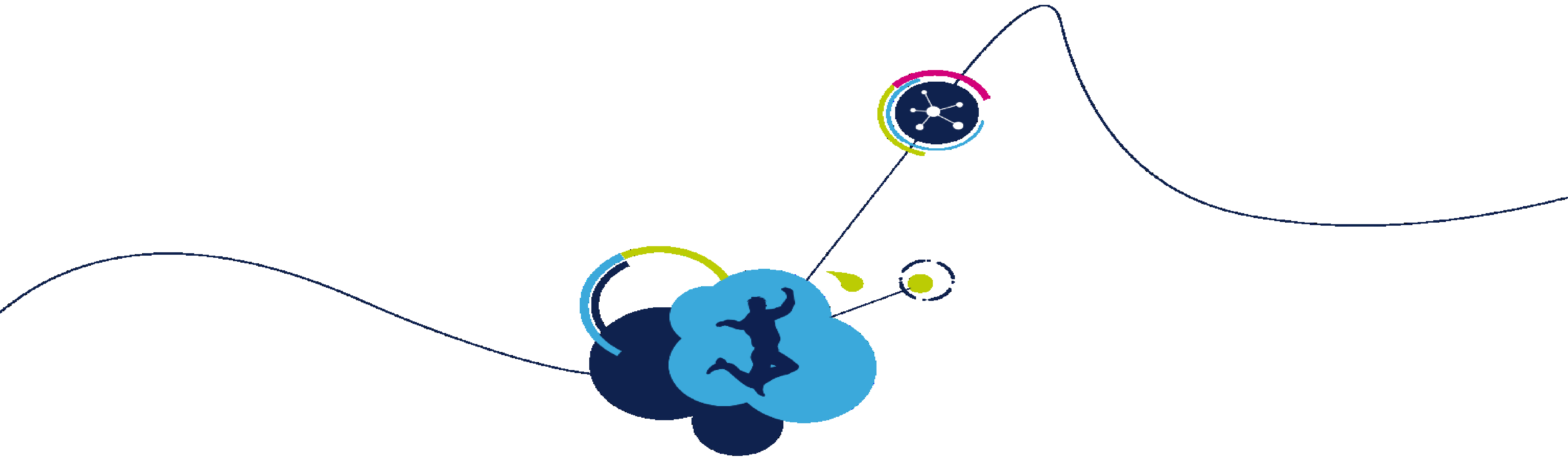
- It is an empty macro defined in FreeRTOS.h file, needs to be defined by the user
- We should define this macro to set xExpectedIdleTime to 0 if the application does not want portSUPRESS_TICKS_AND_SLEEP() to be called

life.augmented

Perform low power entrance

portSUPPRESS_TICKS_AND_SLEEP

- It is an empty macro defined in FreeRTOS.h file, needs to be defined by the user
- It is usually done in port functions (i.e. portmacro.h for gcc)
- There is an assignment to function i.e. vPortSuppressTicksAndSleep() which is defined as “weak” within port.c
- This function is called with the scheduler suspended



FreeRTOS footprint

life.augmented

RTOS'es ported to STM32 - comparison

Features / RTOS	Multitasking	Round-robin scheduling	priority	Number of tasks	Compiler supported	Footprint (kernel size in kB)
CMX-RTX	Preemptive or cooperative	Yes	255	255	IAR/Keil	ROM: 3.904 RAM: 0.748
FreeRTOS	Preemptive or cooperative	Yes	unlimited	unlimited	IAR/Keil /gcc	ROM: 2.7-3.6 RAM: 0.19
µC/OSII	Preemptive	Yes	256	255	IAR/Keil	ROM: 2 RAM: 0.2
Keil-RTX	Preemptive	Yes	256	Unlimited (tasks defined) 256 (tasks active)	ARM/Keil	ROM:1.5-3 RAM < 0.5
embOS	Preemptive	Yes	256	unlimited	IAR	ROM:1.7 RAM :0.06

Thank you

life.augmented