RTOS solution

life.augmented

RTOS object

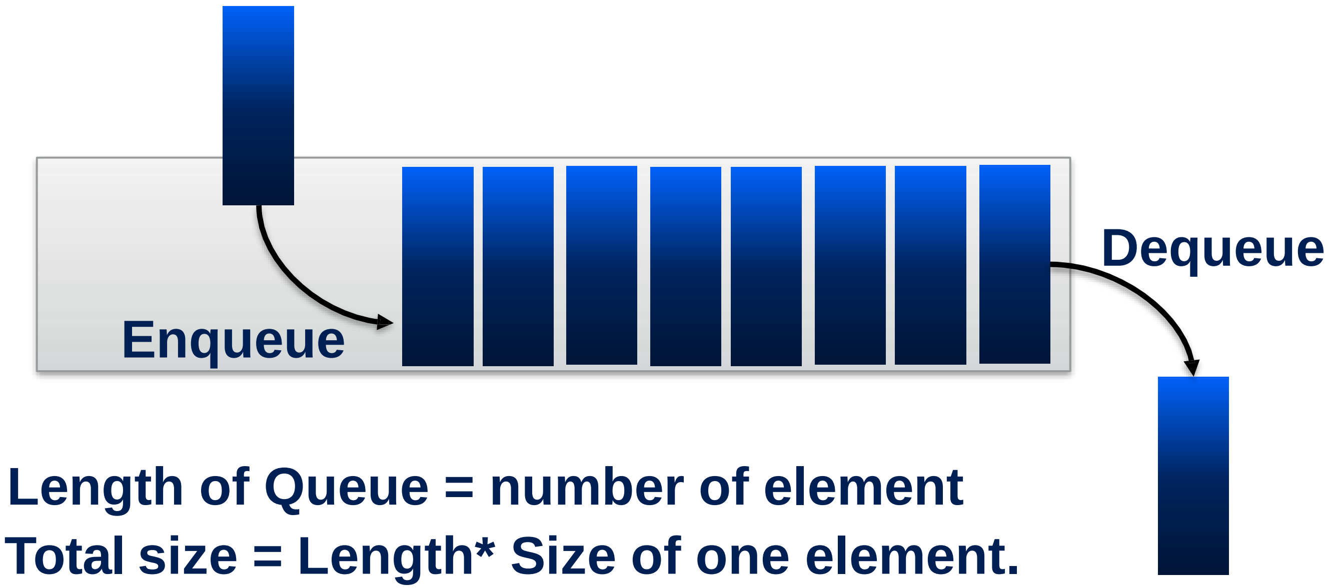
RTOS objects:

1. Task
2. Queue
3. Timer
4. EventGroup
5. SemaphoreBinary
6. SemaphoreCounting
7. SemphoreMutex
8. SemaphoreMutexRecursive

Queue

Queue : is a data structure, which can hold finite number of fixed size data elements

- They can send message between task as well as between interrupt and tasks
- Support appending data to the back of a queue or sendinf data to the head of a queue.
- Items are enqueud by copy not reference.

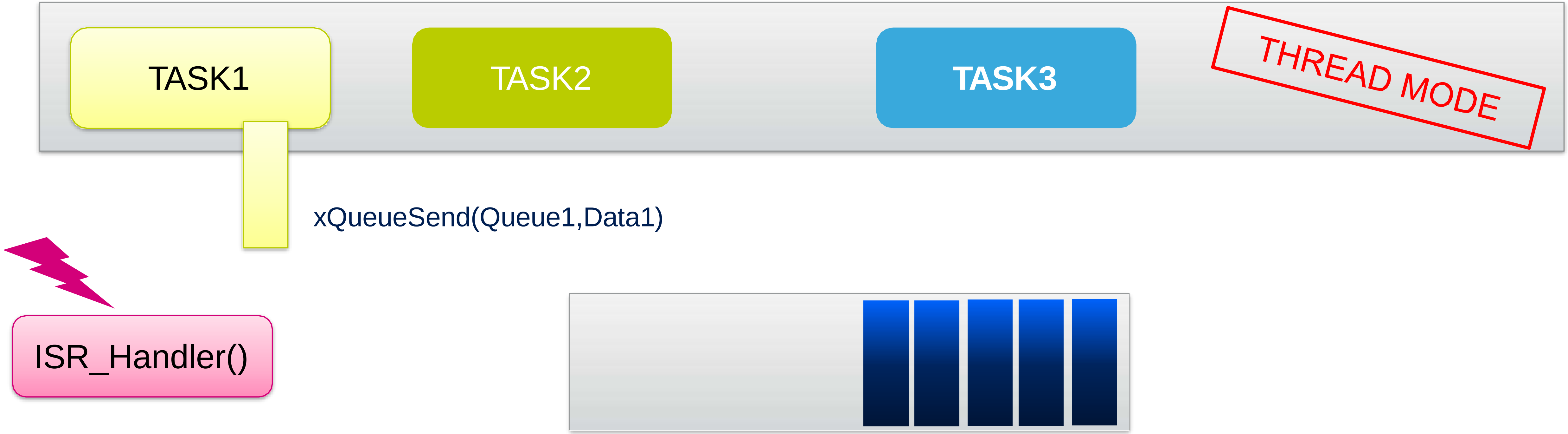


FIFO(First In First Out)

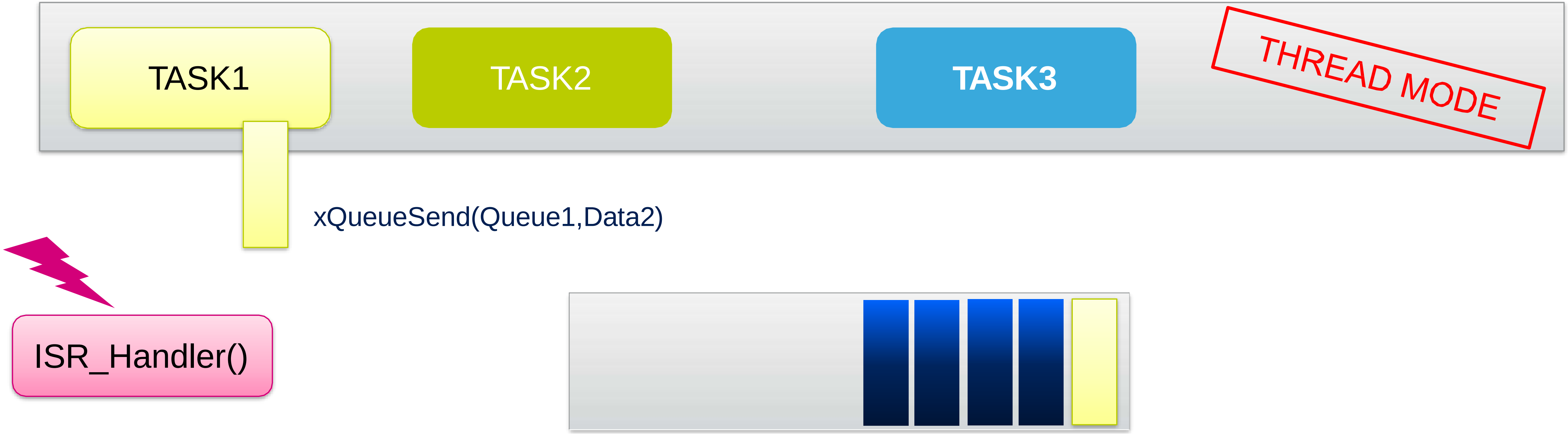
Length of Queue = number of element

Total size = Length* Size of one element.

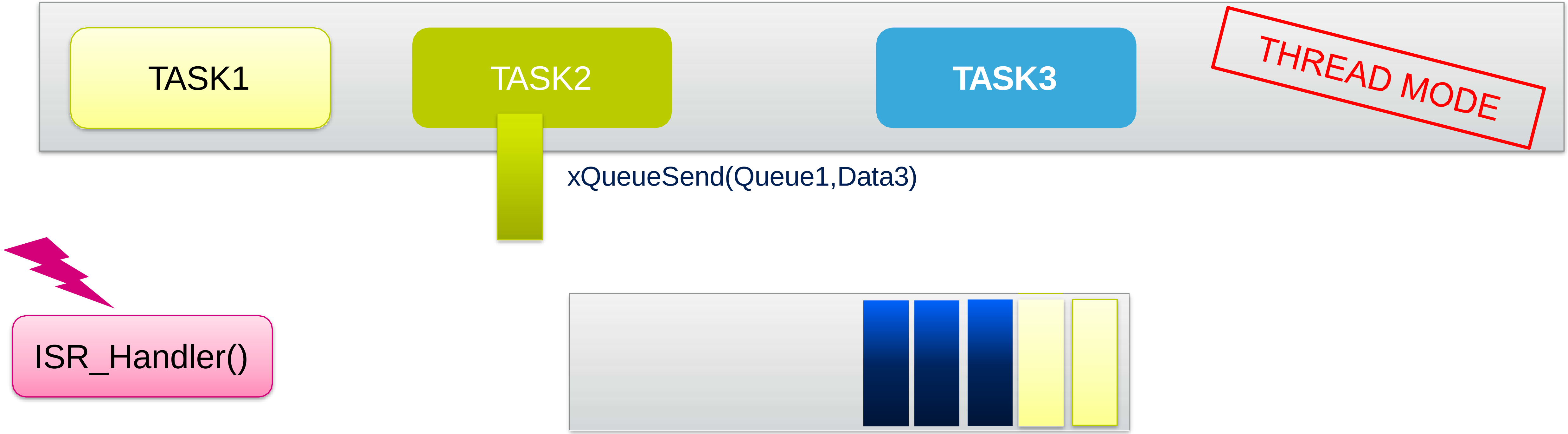
Queue Simulation



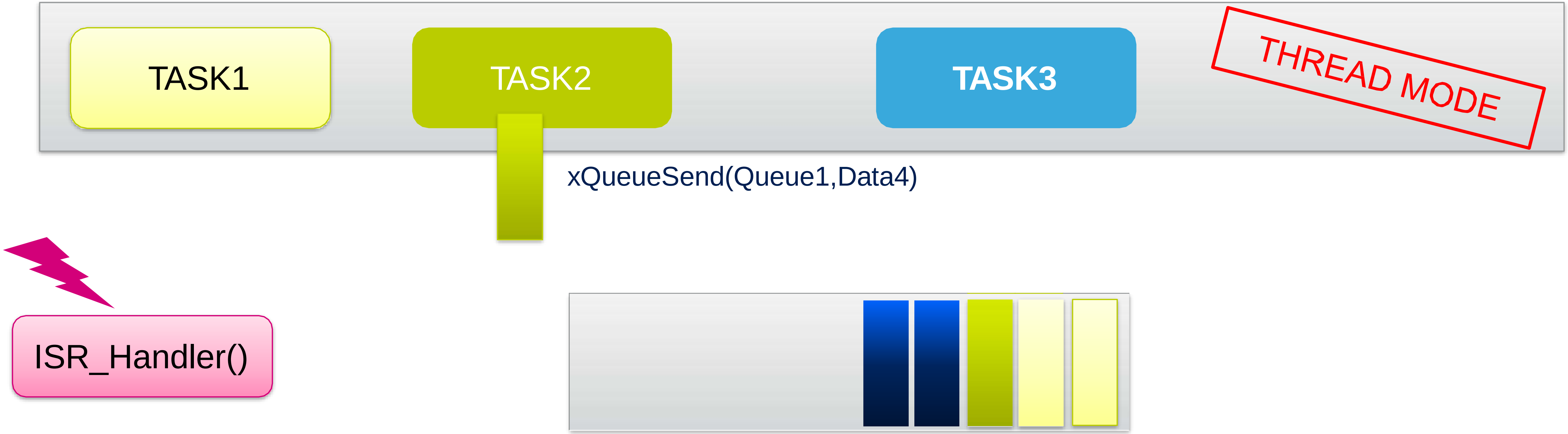
Queue Simulation



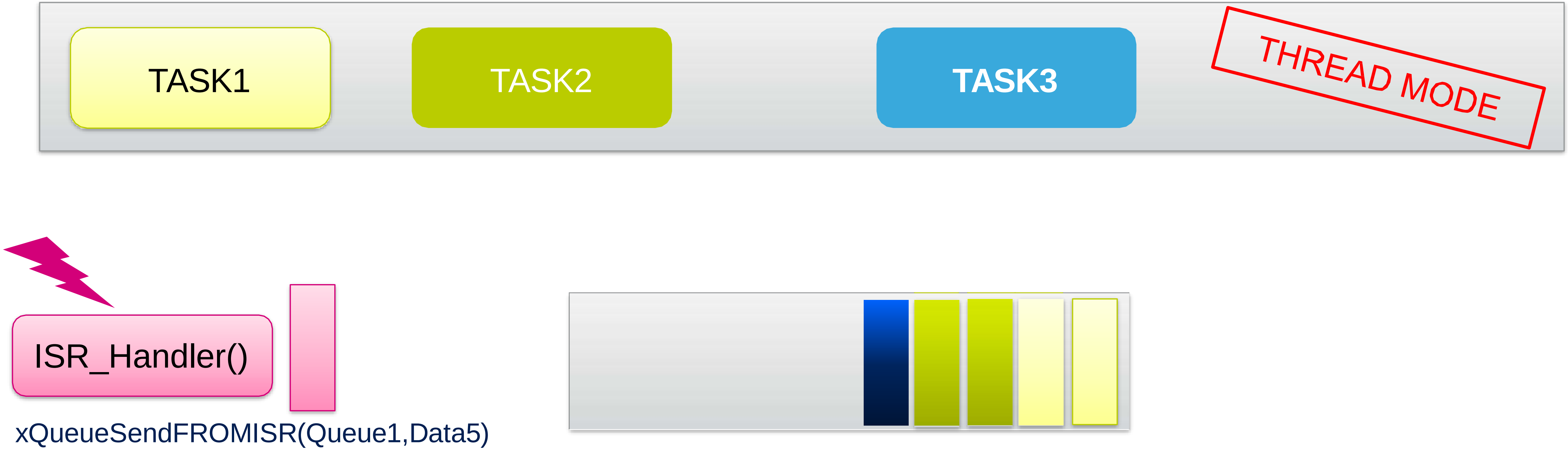
Queue Simulation



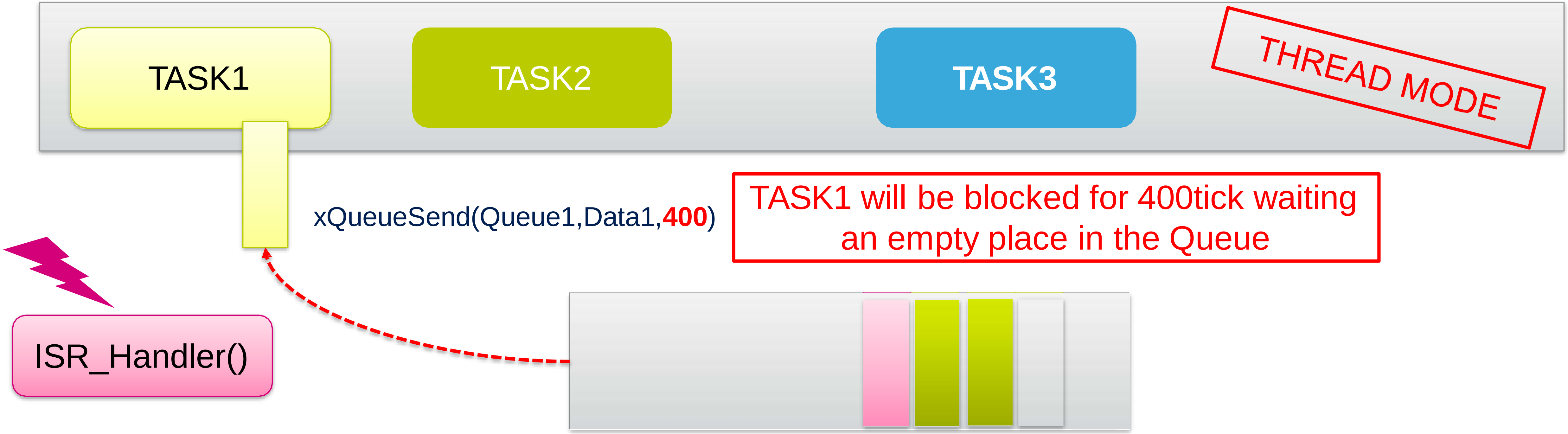
Queue Simulation



Queue Simulation

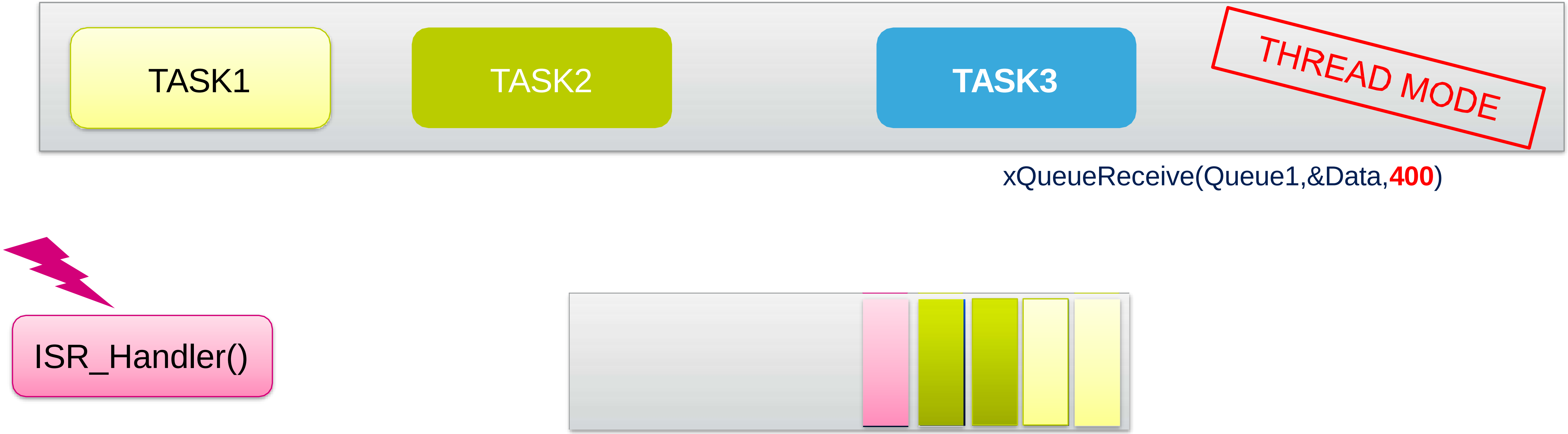


Queue Simulation

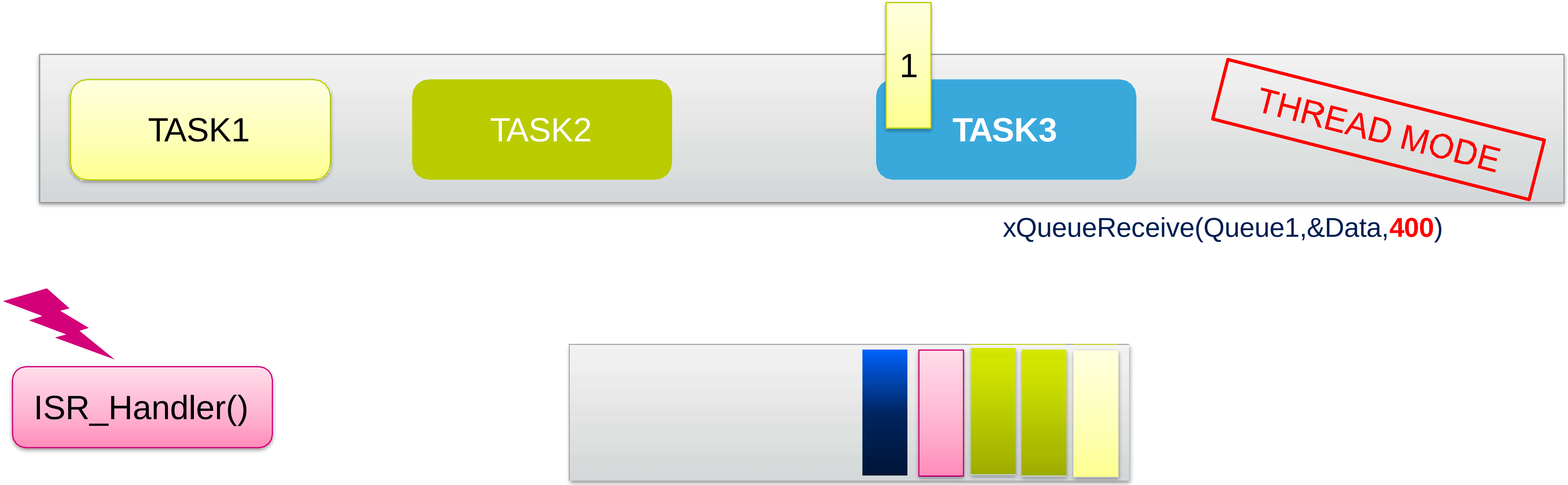


After the 400 Tick, xQueueSend API will return a pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL

Queue Simulation



Queue Simulation



Queue API

Creation

xQueueCreate
vQueueDelete

Fully Featured

xQueueSend
xQueueSendToBack
xQueueSendToFront
xQueueReceive
xQueuePeek
uxQueueMessagesWaiting

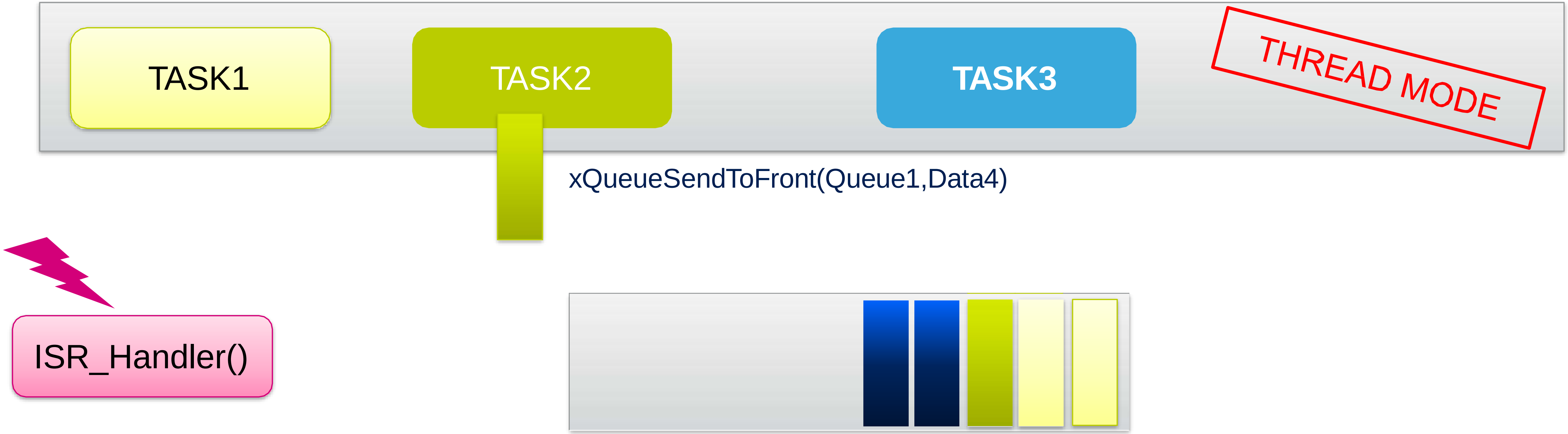
Light Weight

xQueueSendFromISR
xQueueSendToBackFromISR
xQueueSendToFrontFromISR
xQueueReceiveFromISR
uxQueueMessagesWaitingFromISR
xQueueIsQueueEmptyFromISR
xQueueIsQueueFullFromISR

Alternative

xQueueAltSendToBack
xQueueAltSendToFront
xQueueAltReceive
xQueueAltPeek

Queue Simulation
xQueueSendToFront



Queue Simulation

xQueuePeek

TASK1

TASK2

TASK3

THREAD MODE

xQueuePeek(Queue1,&Data,**400**)

ISR_Handler()

Queue and Blocking

- Acces to queue is either blocking or non-blocking
- The scheduler blocks tasks when they attempt to read from or write to a queue that is either empty or full, respectively
- It the xTickToWait varible is zero and the queue is empty(full), the task does not block. Otherwise the task will block for xTickToWait scheduler tick or until an event on the queue free up the ressource

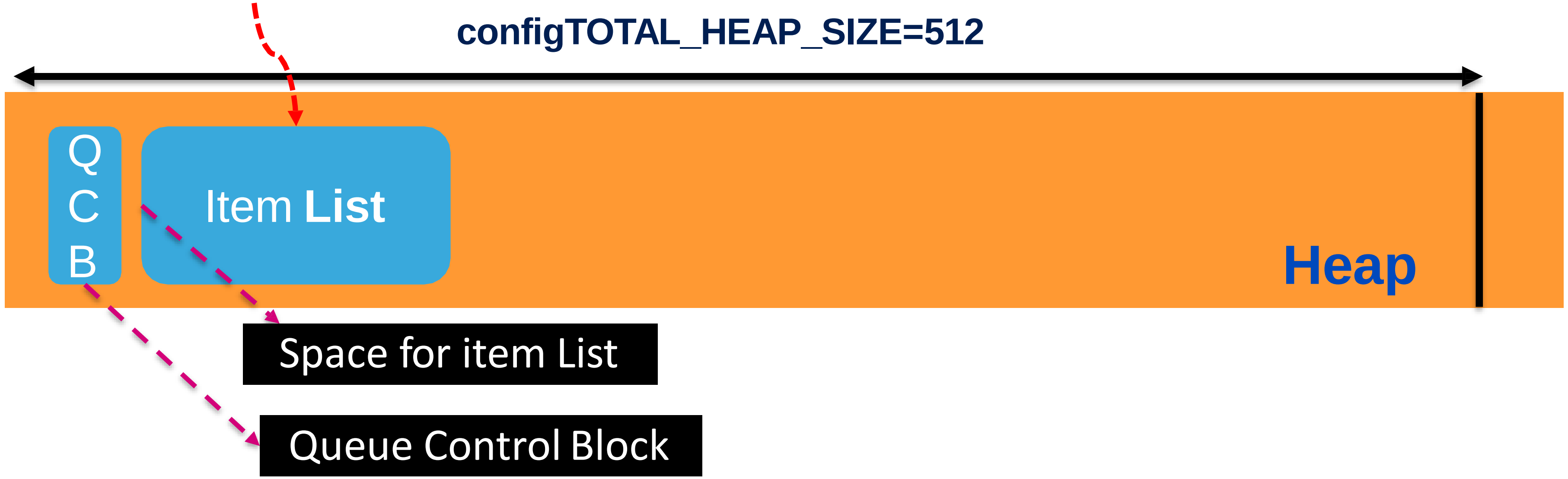
Creating a Queue


```
QueueHandle_t xQueue1;

/**
  * @brief  The application entry point.
  *
  * @retval None
  */
int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
  HAL_Init();


  /* Configure the system clock */
  SystemClock_Config();


  xQueue1 = xQueueCreate( 10, sizeof(uint32_t ) );
```



Lab Queue

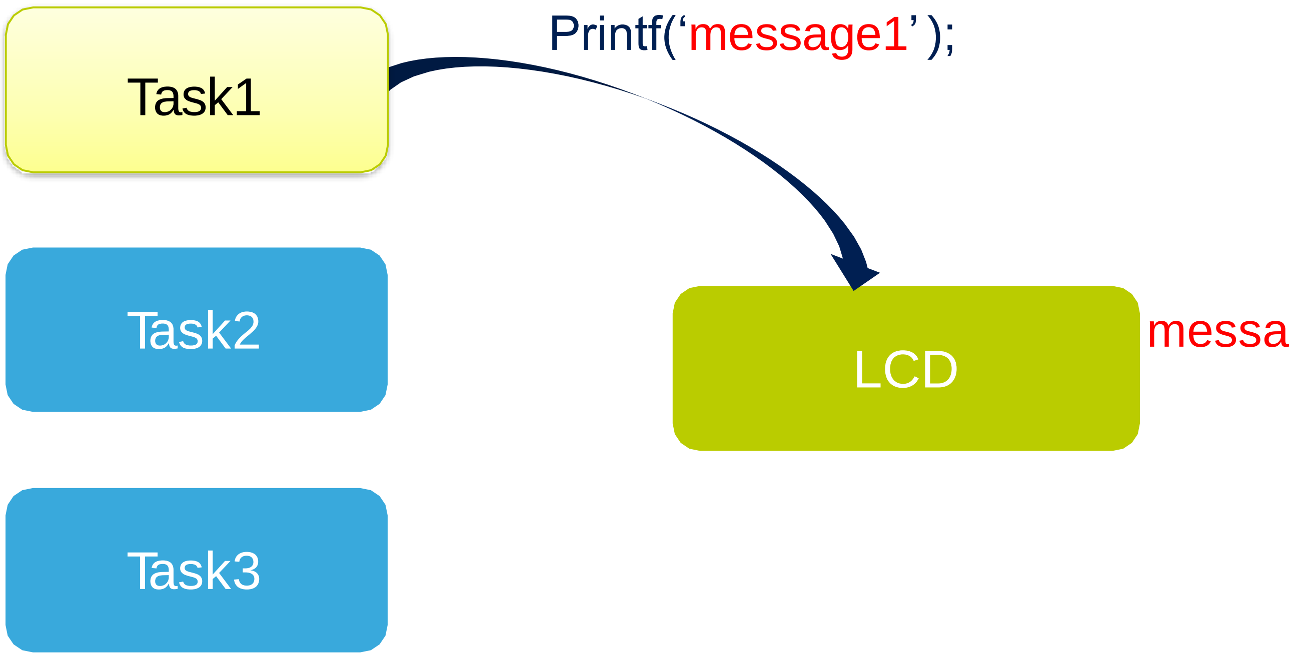
Lab Queue

Ressource mangament

Semaphore/Mutex/Critical Section

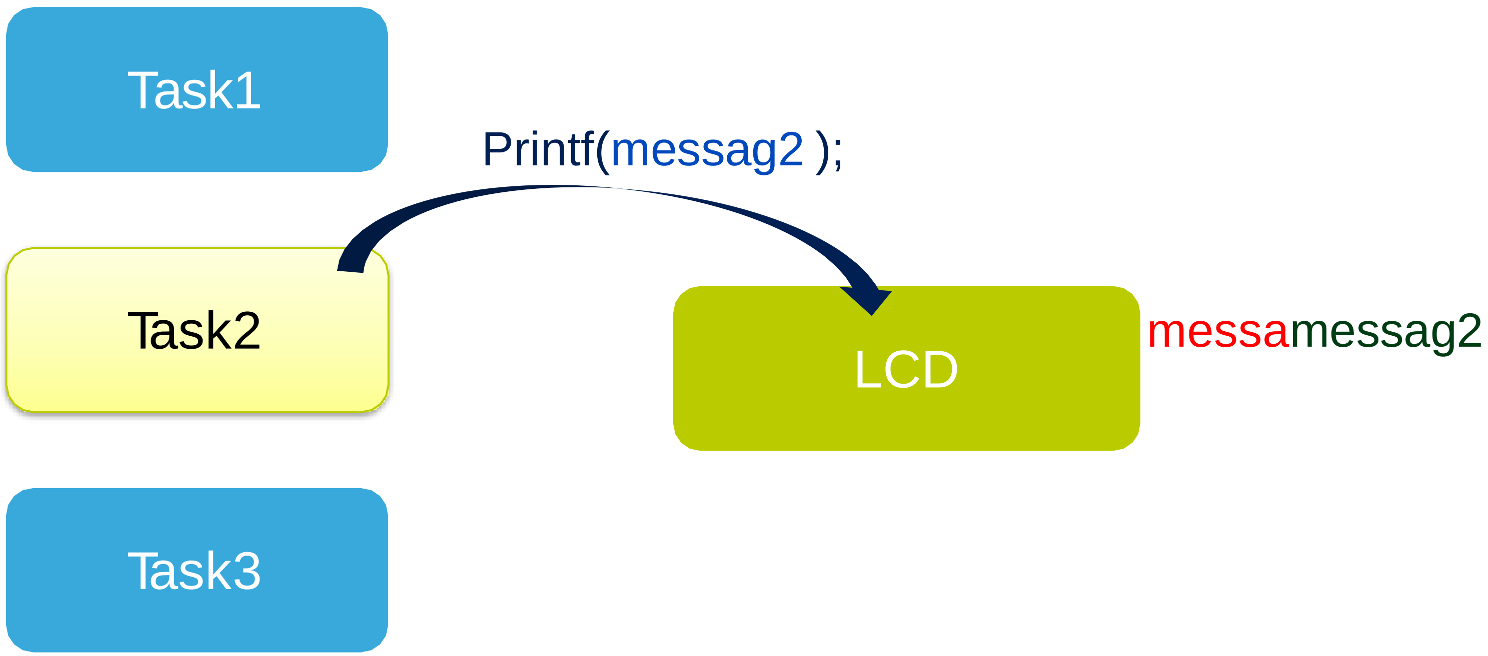
Ressource managment

- In a multitasking system, there is potential conflict if one task starts to access a resource, but does not complete its access before being transited out of the running state.



Ressource managment

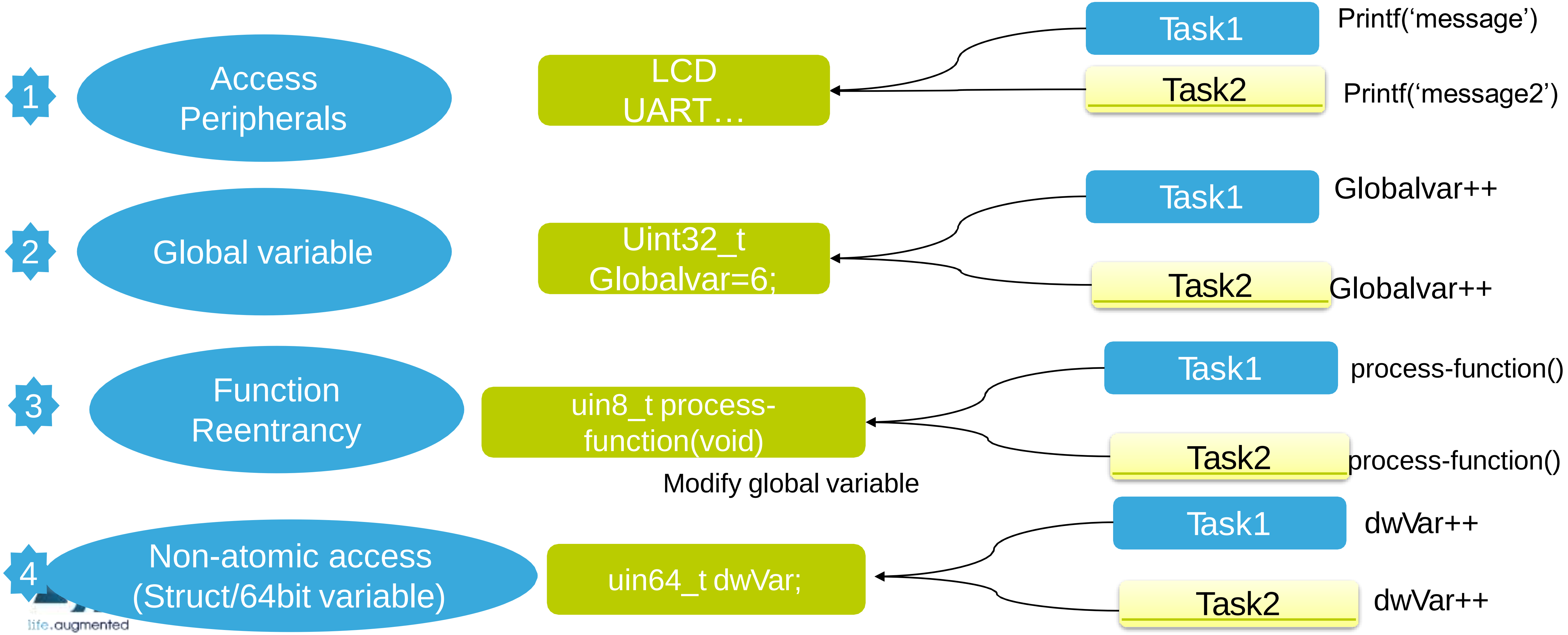
- In a multitasking system, there is potential conflict if one task starts to access a resource, but does not complete its access before being transited out of the running state.



Ressource management:

Shared ressource

- **What is a shared ressource ?**



life.augmented

Ressource managent
Shared ressource

Access peripherals:

- Task A write a string Hello word to LCD
- TaskA is pre-empted by taskB after outputting « Hello w »
- TaskB write « Abort, Retry, Fail » to LCD before entring the Blocked state
- TaskA continues from the point at which it was pre-empted « ord »

=== > the LCD display « Hello wAbort,Rerty, Faillord »

Read, Modify, Write operation:

- A global variable is read from memory into a register, modifed within the register and then written back to memory, **this a non-atomic operation** because it take more than one instruction attemps to update a variable GlobalVar.
- TaskA load GloablVar into a register
- TaskA is pre-empted by TaskB before the modif&write portion of sampe operation
- TasK B update the value of GloablVar then enter in Blocked state
- TaskA contiue writeBack an out-of-date value for GlobalVar
- === > corrupting variable

Ressource managent

Shared ressource

Non-atomic Access to variable

- Updating multipile member of a structure, or updating a variable that is larger than the natural word (64bit)

Function Reentrancy

A function is reentrant if it is safe to cal the func from more than one task or from interrupt

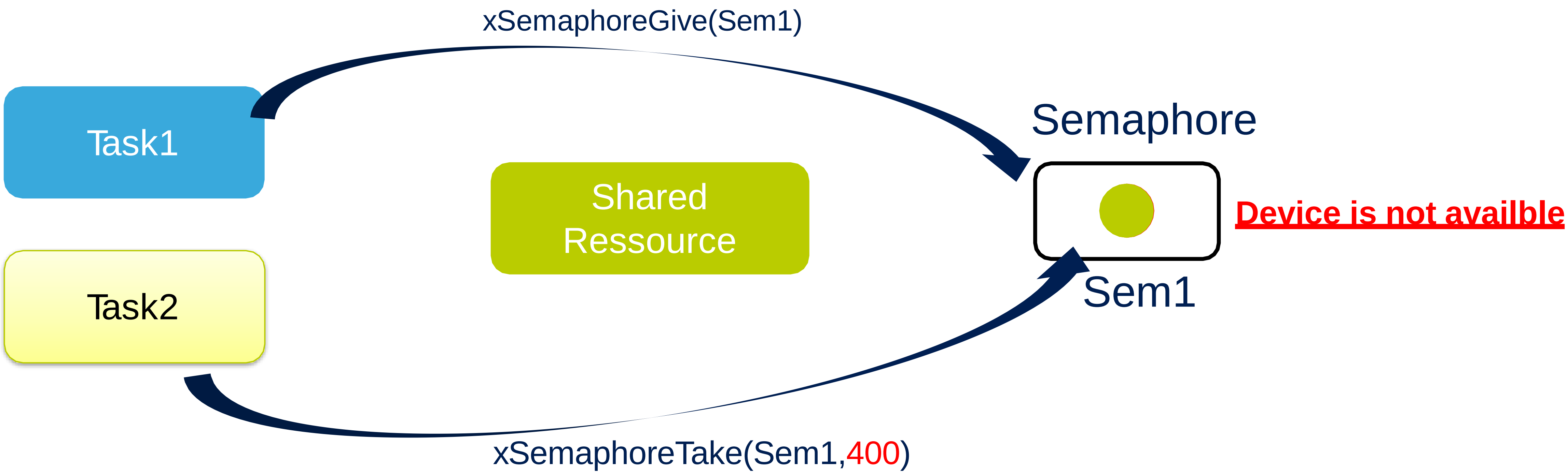
Ressource management:Mutal Exclusion

Ressource management:Mutal Exclusion

Ressource management:Mutal Exclusion

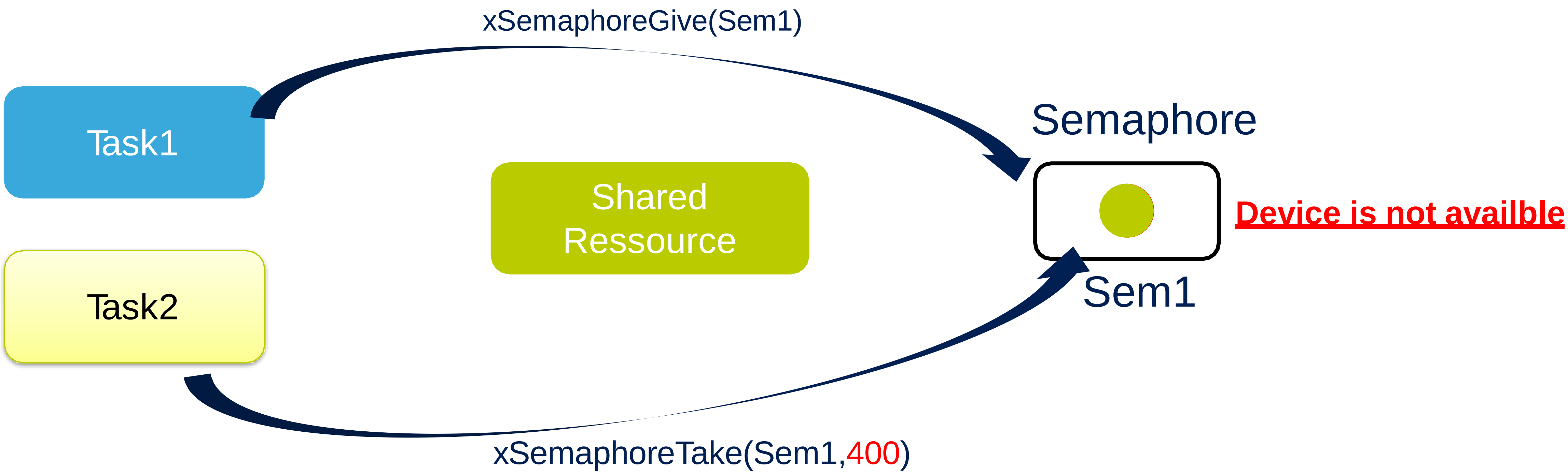
Task2 want to access to the device, it will be blocked until the **semaphore is availble or for 400 Tick.**

Ressource management:Mutal Exclusion



Task2 want to access to the device, it will be blocked until the **semaphore is availble or for 400 Tick.**

Ressource management:Mutal Exclusion



Task2 want to access to the device, it will be blocked until the **semaphore is availble or for 400 Tick.**

Counting Semaphores

Depending on the ressource, how many tasks it can be allowed to use it on the same time. Is there a place for many task ? Or just one task (Mutual exclusion) ?

Counting Semaphores

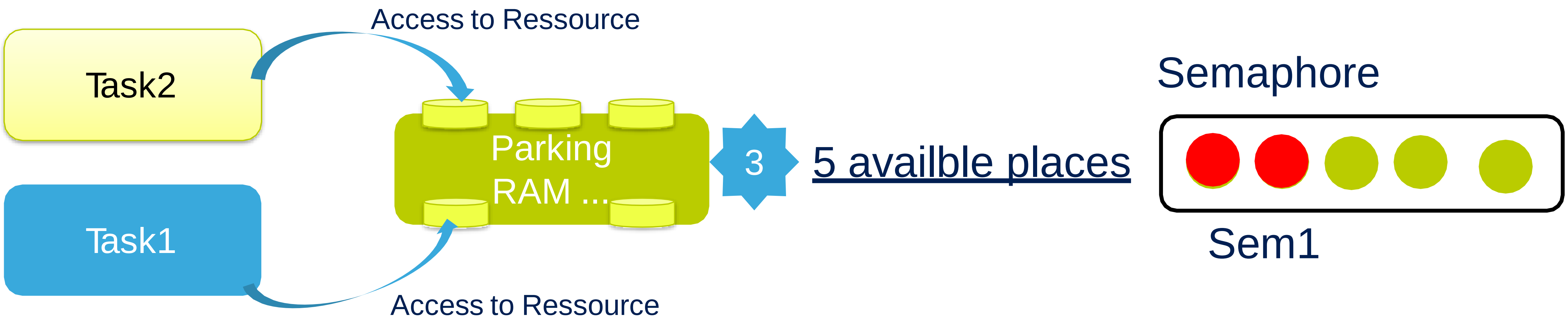
Depending on the ressource, how many tasks it can be allowed to use it on the same time. Is there a place for many task ? Or just one task (Mutual exclusion) ?

Counting Semaphores

Depending on the ressource, how many tasks it can be allowed to use it on the same time. Is there a place for many task ? Or just one task (Mutual exclusion) ?

Counting Semaphores

Depending on the ressource, how many tasks it can be allowed to use it on the same time. Is there a place for many task ? Or just one task (Mutual exclusion) ?



Counting Semaphores

Typically used for two things:

- Counting events:
  - An *event handler* will 'give' a semaphore each time an event occurs, and a *handler task* will 'take' a semaphore each time it processes an event
- Resource management:
  - The count value indicates number of available resources
  - To get a resource, a task must obtain (take) a semaphore
  - When a task finishes with the resource, it 'gives' the semaphore back
-
```
SemaphoreHandle_t xSemaphoreCreateCounting(
        UBaseType_t uxMaxCount,
        UBaseType_t uxInitialCount)
```

Critical section

- Basic Critical Section

taskENTER_CRITICAL() and **taskEXIT_CRITICAL()/**

A switch to another task cannot occur, interrupt ma still execute, but only inerrupts whose priority is above the value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY const

```
void vPrintString( const char *pcString )
{
static char cBuffer[ ioMAX_MSG_LEN ];

    /* Write the string to stdout, using a critical section as a crude method
    of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    taskEXIT_CRITICAL();
}
```

life.augmented

Critical section

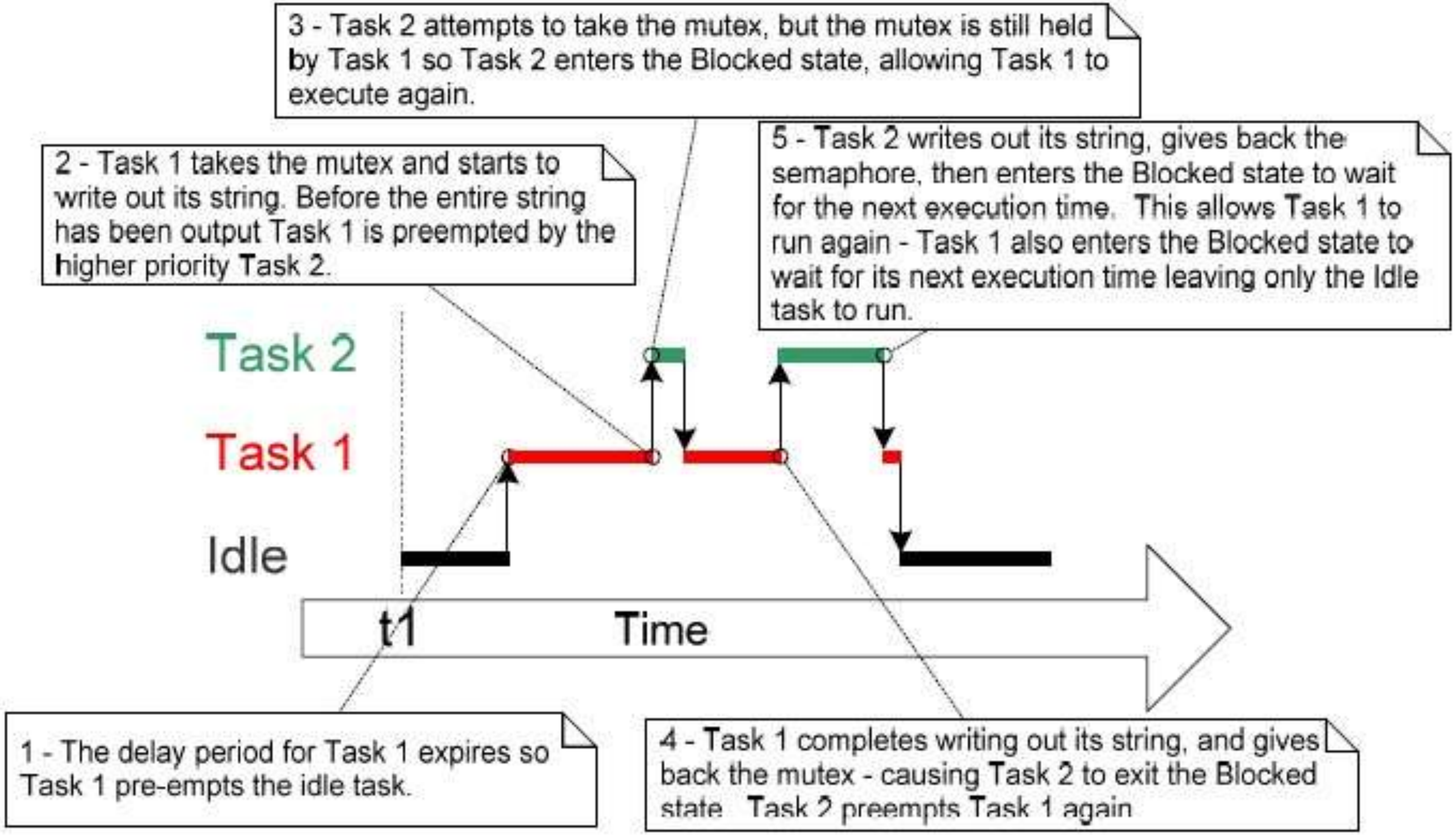
The task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exist

- *Critial sectino must be kept very short, standard out(stdout, or stream..) shoul not be protected using a critical section because operation are relatively long*

Mutex(binary semaphore)

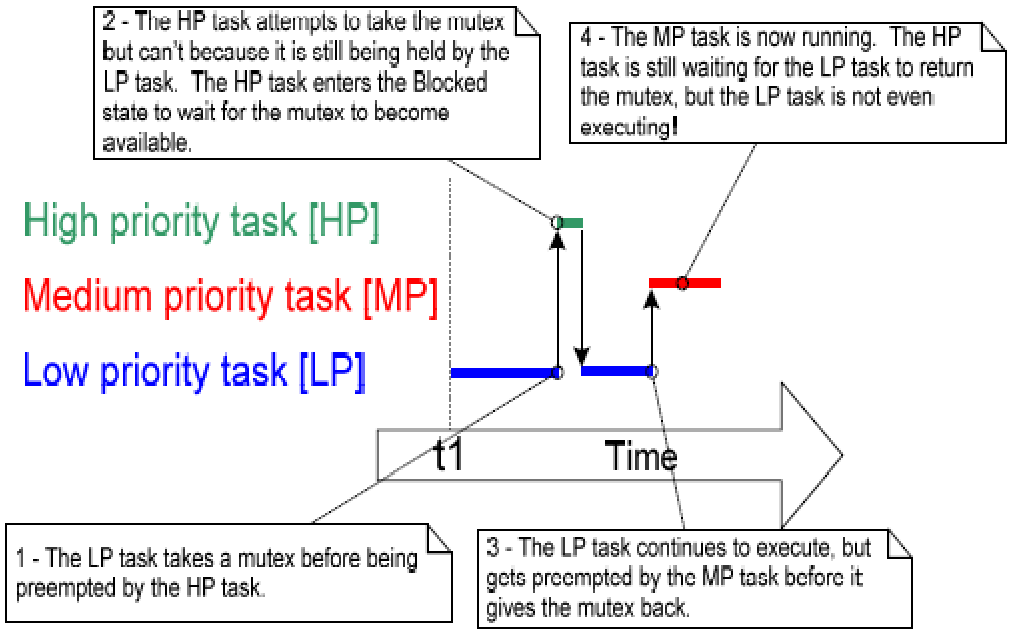
- Mutex is special type of binary semaphore that is used to control access to ressource that is shared between two or more task.
- Mutex (Mutual Exclusion), when used in mutual exclusion scenario, the mutex can be though as a token that is assiciated with the ressource being shared
- A semaphore that is used for mutual execlusion must always be returned
- A semphore that is used for sychronization is normally discarded and not returned.

Mutual exclusion



life.augmented

Priority inversion (Mutual exclusion)



The result would be a high priority task waiting for a low priority task without the low prioriy task even being able to execute.

Priority inversion can be a significant problem, but in small embedded system it can often be avoided at system design time, by considering how ressource are accessed/

Priority Inheritance

- The difference between binary semaphore and mutex:
- **Mutex include a basic priority inheritance mechanism**

Priority inheritance: is a scheme that mnimizes the negative effects of priority inversion

Raise temporay the task LP prioriy to the HP prioriy to finsh it job

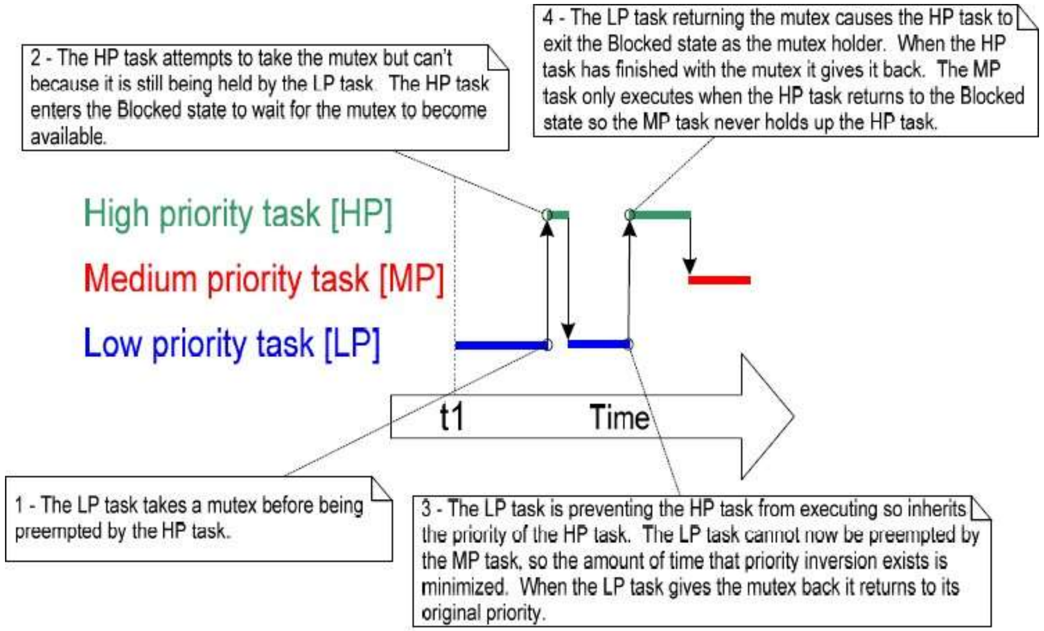


Figure 39. Priority inheritance minimizing the effect of priority inversion

Prioriy Inheritance

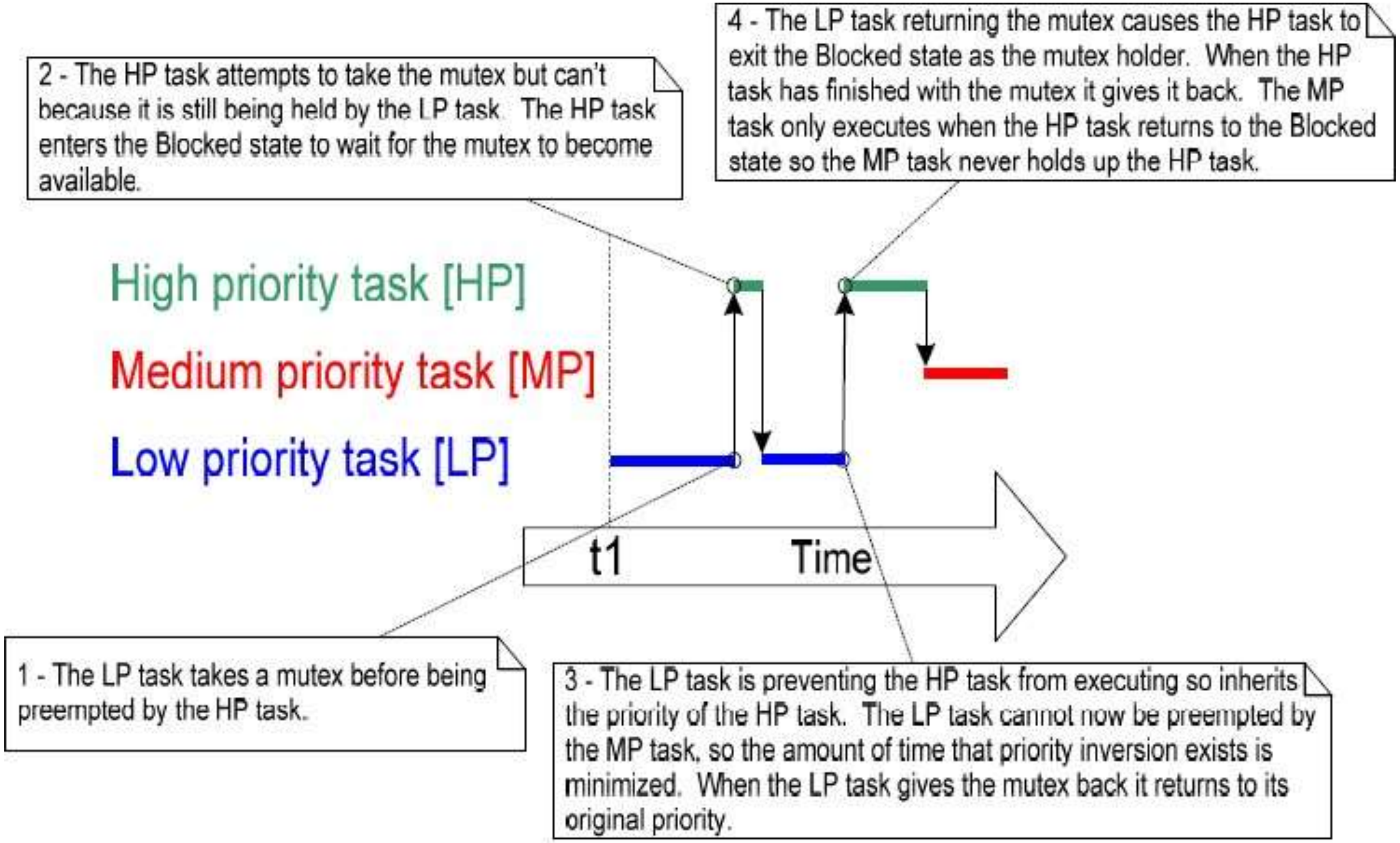


Figure 39. Priority inheritance minimizing the effect of priority inversion

Deadlock (Deadly Embrace)

- Potential putfall (piege) that can occur when using mutexes
- Deadlock occurs when two tasks cannot proceed because they are both waiting for a resoucre that is held by the other.
- Task A execute and take mutex X
- Task A preempted by Task B which take a mutex Y
- TasA try to take semaphore Y → go Blocking
- TaskB try to ake mutex X -- > go Blocking
- **== > the system designer should avoid this case**

Deadlock (Deadly Embrace)

- Task A execute and take mutex X
- Task A preempted by Task B which take a mutex Y
- TasA try to take semaphore Y → go Blocking
- TaskB try to ake mutex X -- > go Blocking

Binary semaphore vs Mutex

- Mutexes and binary semaphores are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

- The priority of a task that ‘takes’ a mutex will be temporarily raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex ‘inherits’ the priority of the task attempting to ‘take’ the same mutex. This means the mutex must always be ‘given’ back – otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never ‘disinherit’ the priority.

Lab Semaphore & Critical Section

life.augmented

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore. */
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )
    {
        /* There was insufficient FreeRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the
        xSemahore variable.  Calling xSemaphoreTake() on the semaphore here
        will fail until the semaphore has first been given. */
    }
}
```

```
/* See if we can obtain the semaphore.  If the semaphore is not
available wait 10 ticks to see if it becomes free. */
if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
{
    /* We were able to obtain the semaphore and can now access the
    shared resource. */

    /* ... */

    /* We have finished accessing the shared resource.  Release the
    semaphore. */
    xSemaphoreGive( xSemaphore );
}
```

Task synchronization

Semaphore & Queue

Task sychronization

Meeting @ 3pm

Sychronization established

Meeting @ 3pm

Wasting your time arriving sooner

Meeting @ 3pm

You are not yet arrived-wasting time

Sychronization with Semaphore

Sychronization with Queue

Interrupt managment &Deferred Interrupt

Deferred Interrupt Handling

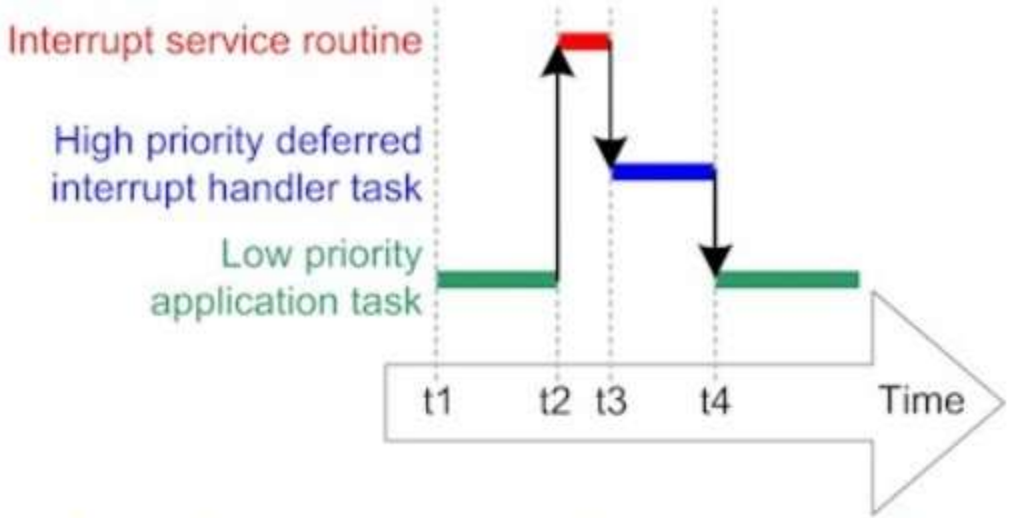
What is deferred interrupt handling?

- In FreeRTOS, a deferred interrupt handler refers to an RTOS task that is unblocked (triggered) by an interrupt service routine (ISR) so the processing necessitated by the interrupt can be performed in the unblocked task, rather than directly in the ISR. The mechanism differs from standard interrupt processing, in which all the processing is performed within the ISR, because the majority of the processing is deferred until after the ISR has exited:

- **Standard ISR Processing**Standard ISR processing will typically involve recording the reason for the interrupt, clearing the interrupt, then performing any processing necessitated by the interrupt, all within the ISR itself.

- **Deferred Interrupt Processing** Deferred interrupt processing will typically involve recording the reason for the interrupt and clearing the interrupt within the ISR, but then unblocking an RTOS task so the processing necessitated by the interrupt can be performed by the unblocked task, rather than within the ISR.If the task to which interrupt processing is deferred is assigned a high enough priority then the ISR will return directly to the unblocked task (the interrupt will interrupt one task, but then return to a different task), resulting in all the processing necessitated by the interrupt being performed contiguously in time (without a gap), just as if all the processing had been performed in the ISR itself. This can be see in the image below, where all the interrupt processing occurs between times t2 and t4, even though part of the processing is performed by a task.

Deferred interrupt



Deferred interrupt processing execution sequence when the deferred handling task has a high priority

With reference to the image above:

1. At time t2: A low priority task is pre-empted by an interrupt.
2. At time t3: The ISR returns directly to a task that was unblocked from within the ISR. The majority of interrupt processing is performed within the unblocked task.
3. At time t4: The task that was unblocked by the ISR returns to the Blocked state to wait for the next interrupt, allowing the lower priority application task to continue its execution.

Deferred Interrupt Handling

When to use deferred interrupt handling

- Most embedded engineers will strive to minimise the amount of time spent inside an ISR (to minimise jitter in the system, enable other interrupts of the same or lower priority to execute, maximise interrupt responsiveness, etc.), and the technique of deferring interrupt processing to a task provides a convenient method of achieving this. However, the mechanics of first unblocking, and then switching to, an RTOS task itself takes a finite amount of time, so typically an application will only benefit from deferring interrupt processing if the processing:

- Needs to perform lengthy operations, or

- Would benefit from using the full RTOS API, rather than just the ISR safe API, or

- Needs to perform an action that is not deterministic, within reasonable bounds.

The macro portEND_SWITCHING_ISR()

- The macro is part of the FreeRTOS cortexM port, and is the ISR safe equivalent of taskYIELD(). It will force a contex switch only if its parameter is not zero(pdFALSE).
- Note how the xHigherPrioritTaskWoken is used, it is intialized to pdFALSE before being passed by reference to xSemaphoreGiveFROMISR(), where it will get set to pdTRUE only if xSemaphoreGiveFromISR() causes a task of >= prioriy than the currently executing task to leave the blocked state.
- portEND_SWITCHING_ISR(à then performs a context switch only if the xHigherPrioritTaskWoken == pdTRUE, in all other case a contex switch is not necassary, because the task that was executing before the interrupt occurs will still be the highest priority task that is able to run.

Queue From ISR

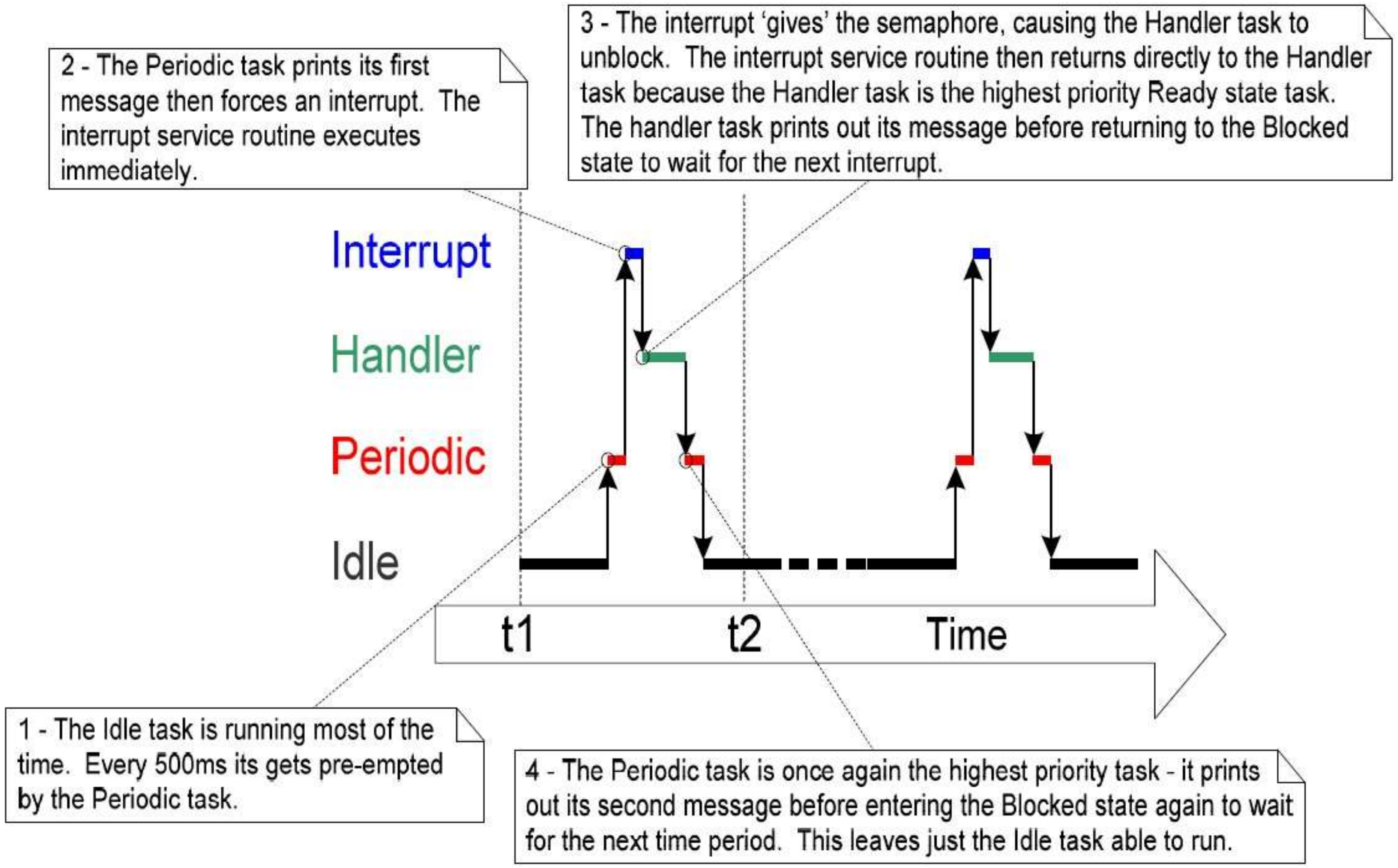


Figure 28. The sequence of execution when Example 12 is executed

Interrut config FREERTOS

Freertosconfig.h

```
/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY   15

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5

/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY                 ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

Interrupt Nesting

Constant	Description
configKERNEL_INTERRUPT_PRIORITY	Sets the priority of interrupts used by the kernel itself—namely the timer interrupt used to generate the tick and the PendSV (Pend Service Call) interrupt used within the API. configKERNEL_INTERRUPT_PRIORITY will almost always be set to the lowest possible interrupt priority.
configMAX_SYSCALL_INTERRUPT_PRIORITY	Defines the highest interrupt priority from which FreeRTOS API functions can be called. Only API functions that end in 'FromISR' can be called from within an interrupt.

Interrupt nesting