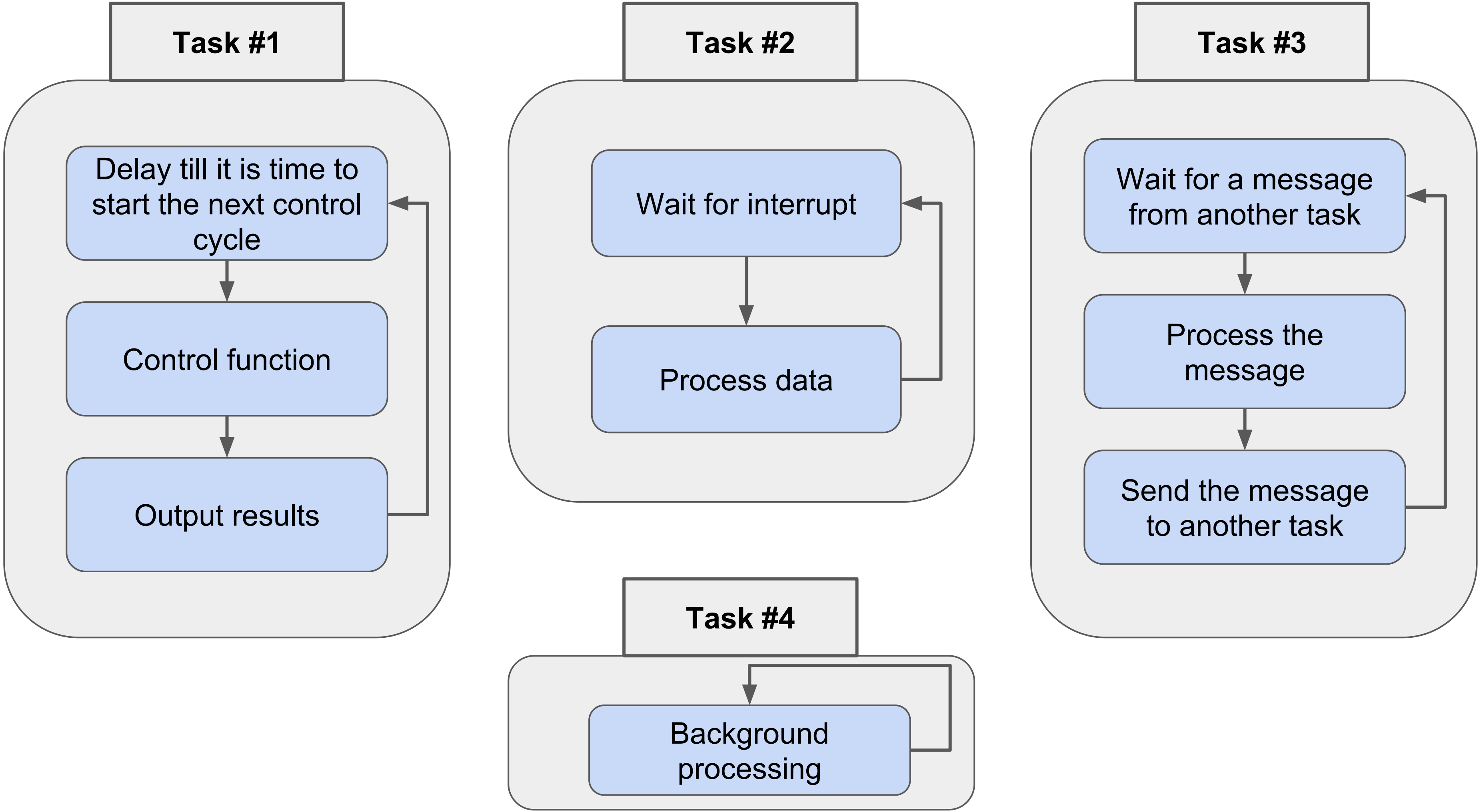
Introduction to embedded programming on STM32

RTOS. Interprocess communication (IPC)

Skoltech, 2019


Task #1
Delay till it is time to start the next control cycle
Control function
Output results
Task #2
Wait for interrupt
Process data
Task #3
Wait for a message from another task
Process the message
Send the message to another task
Task #4
Background processing

FreeRTOS. Interprocess communication

- Task Notifier
- Mutexes
- Queues

FreeRTOS. Mutexes

- Multitasking systems are errors prone
- All running tasks might be preempted at anytime
- Resources might be left in inconsistent states

Example #1

1. Task 1 is executed which in turn starts to write string “Hello world” to the LCD
2. Task 1 is preempted by Task 2 after outputting just the initial part of the string - “Hell”
3. Task2 writes “Skoltech” to the display and enters blocked state
4. Task1 continues from the point it was preemted and outputs the remaining part of the phrase - “o world”
5. Now, LCD shows “HellSkoltecho world”

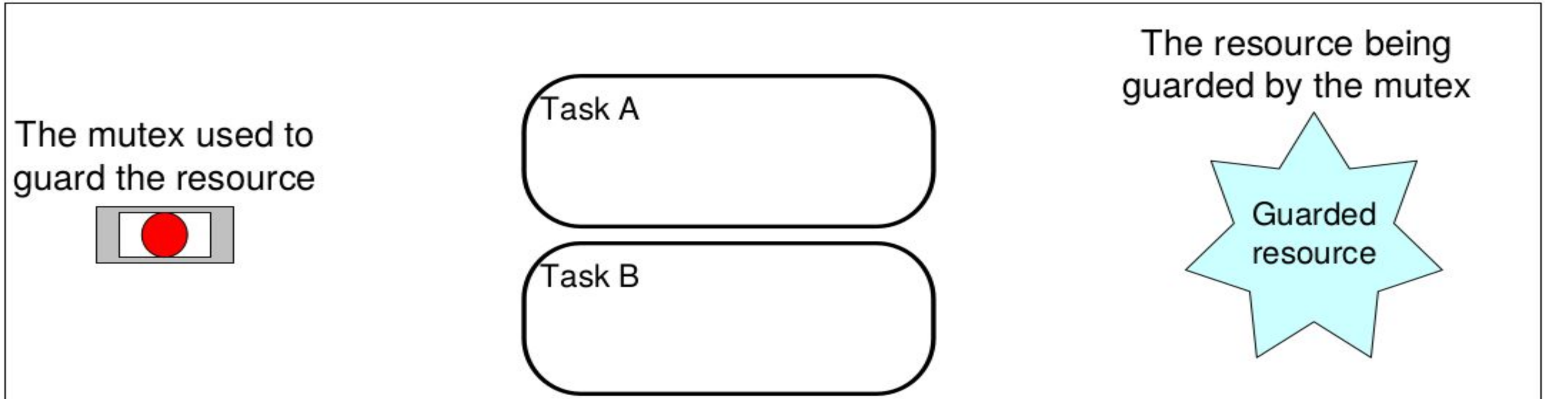
FreeRTOS. Mutexes

Example #2 (RMW operations) - to modify output state of the whole port, we read current state (R), modify some of bits (M) and store result (W).

1. Task #1 loads the value of GPIOA->ODR into a reg
2. Task #1 is preempted by Task #2 before it completes the modify - write portions
3. Task #2 changes the value of GPIOA->ODR and goes to Blocked state
4. Task #1 modify the value it already holds in a reg and writes an outdated value to GPIOA->ODR

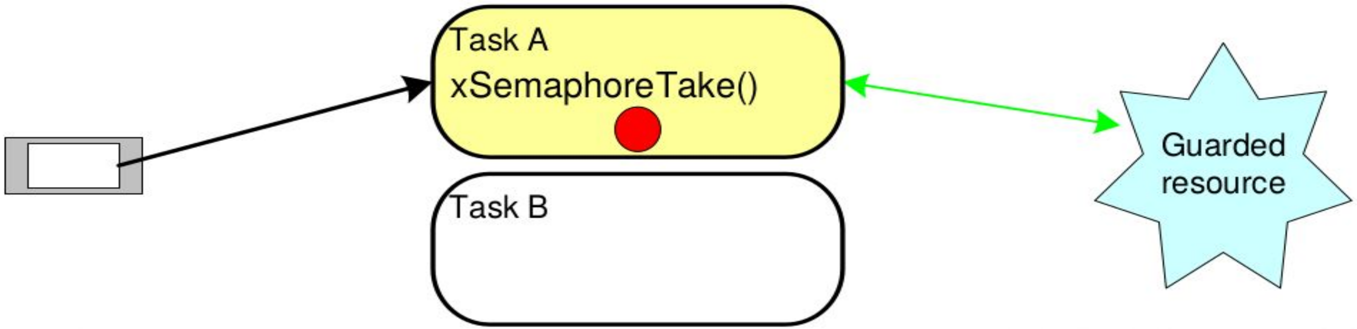
Access is not atomic!

FreeRTOS. Mutexes



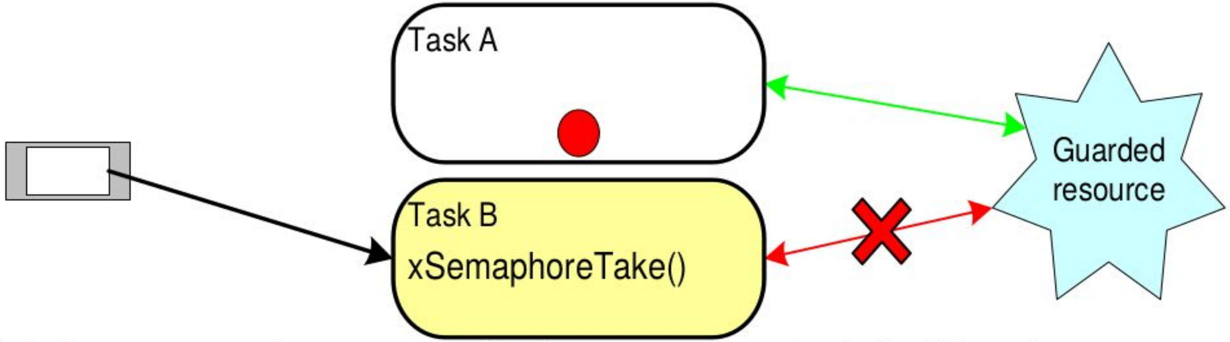
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.

FreeRTOS. Mutexes



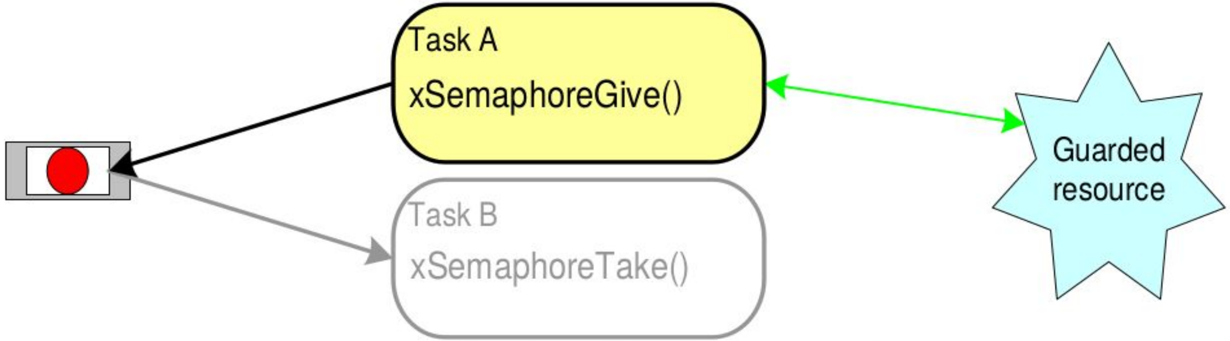
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.

FreeRTOS. Mutexes



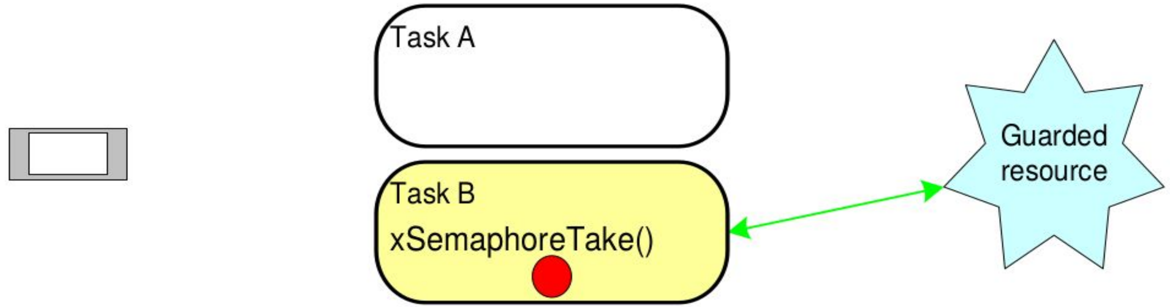
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.

FreeRTOS. Mutexes



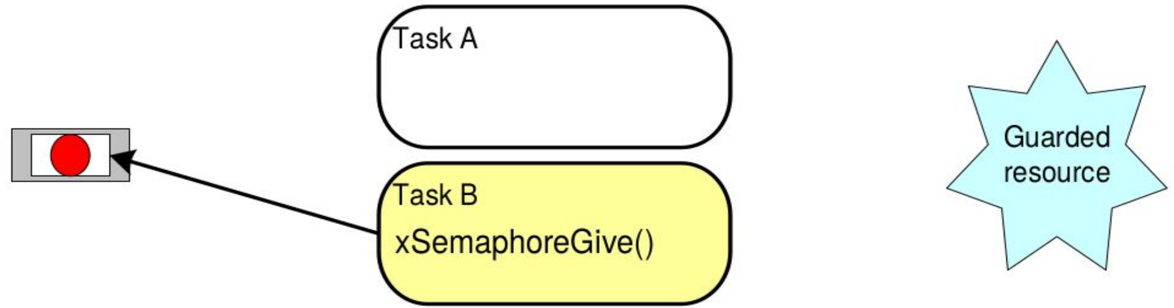
Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again.
Task A finishes with the resource so ‘gives’ the mutex back.

FreeRTOS. Mutexes



Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.

FreeRTOS. Mutexes

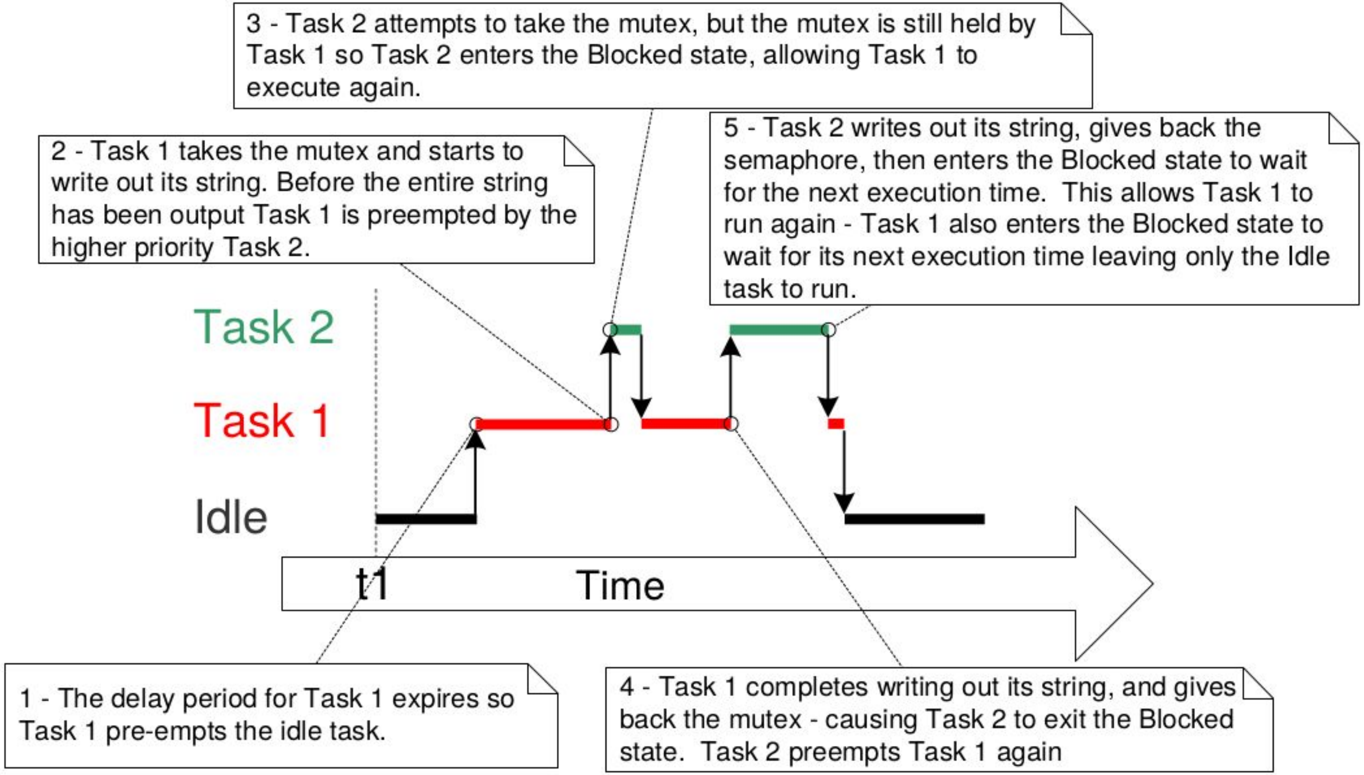


When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

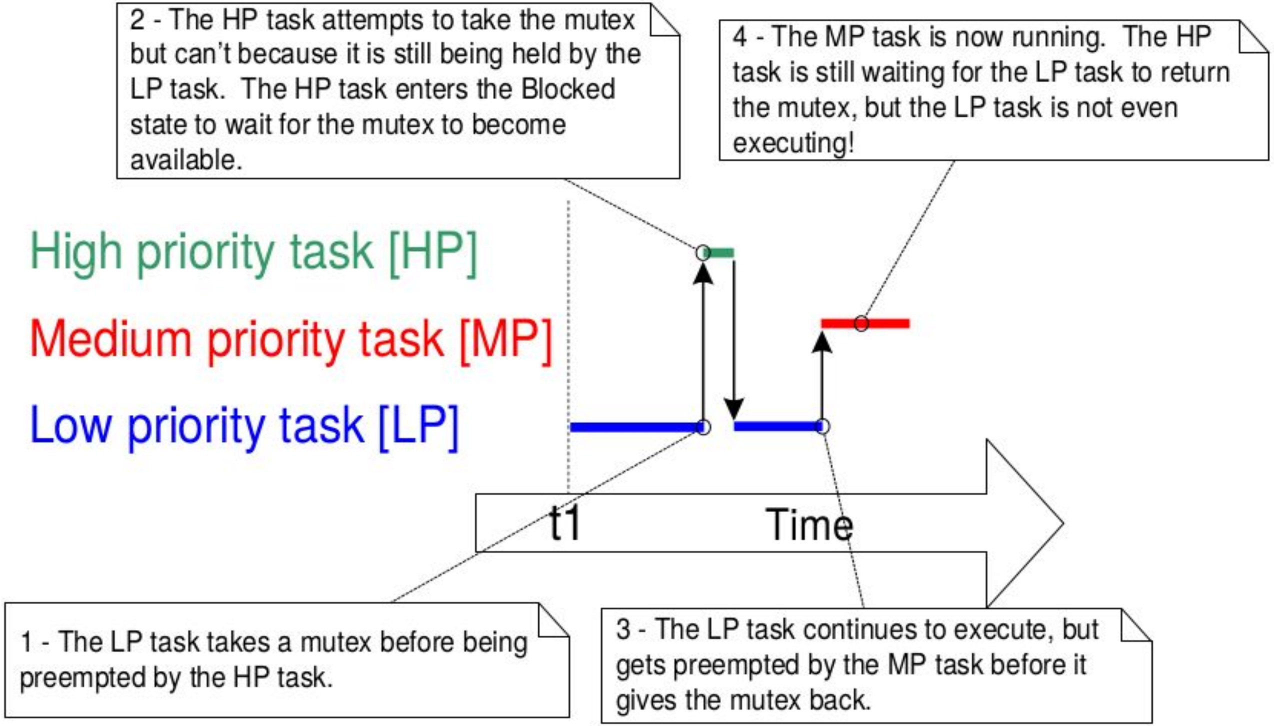
FreeRTOS. Mutexes. API

- *SemaphoreHandle_t* xSemaphoreCreateMutex(void)
- xSemaphoreGive(*SemaphoreHandle_t* xMutex)
- xSemaphoreTake(*SemaphoreHandle_t* xMutex, *uint32_t* delay)

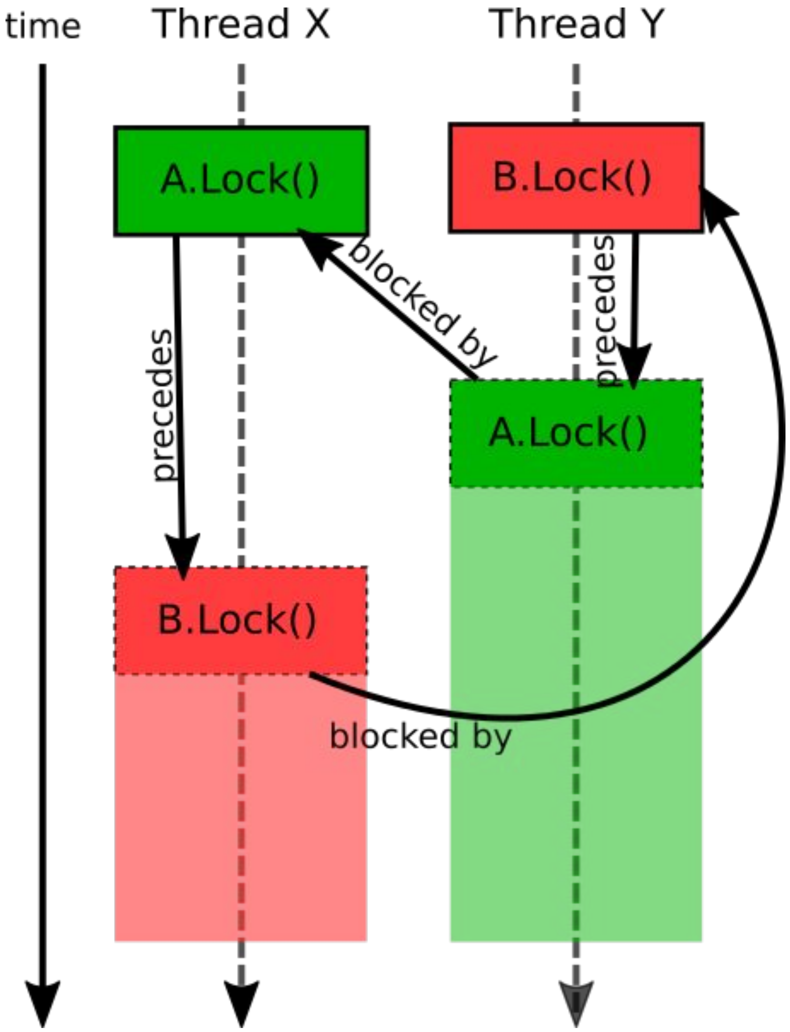
FreeRTOS. Mutexes. Fixed example with LCD



FreeRTOS. Mutexes. Priority Inversion



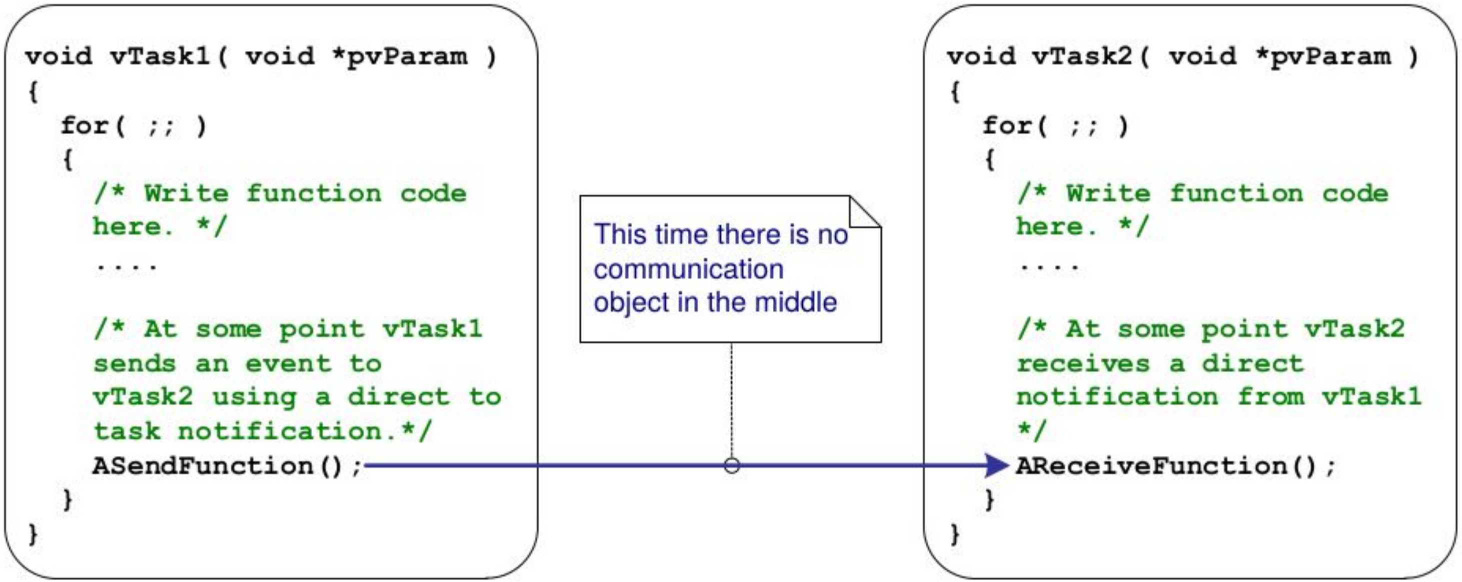
FreeRTOS. Mutexes. Deadlock



FreeRTOS. Task Notifier

- Allows to communicate with other tasks, or synchronize with ISRs, w/o the need for a separate communication object
- Lightweight compared to any other IPC
- Less use of RAM
- But it cannot be used in the following scenario:
  - Sending event to an ISR
  - More than one receiving task
  - Buffering several data items

FreeRTOS. Task Notifier



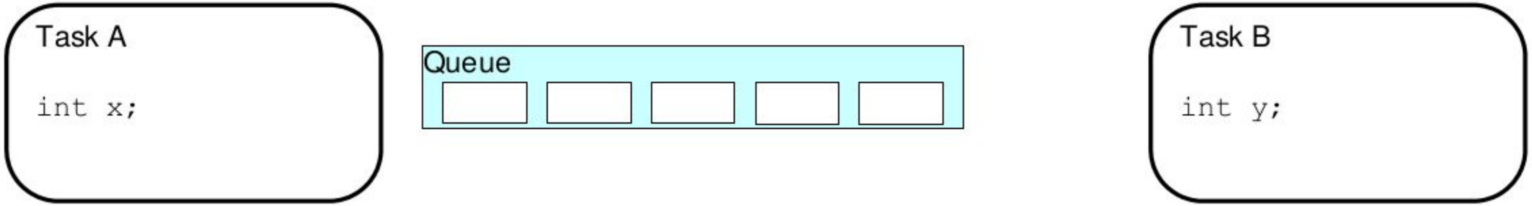
FreeRTOS. Task Notifier. API

- `TaskHandle_t xTaskGetCurrentTaskHandle()`
- `xTaskNotifyGive(TaskHandle_t task_handler)`
- `ulTaskNotifyTake(BaseType_t xClearCountOnExit, TickType_t xTicksToWait)`
- `vTaskNotifyGiveFromISR(TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken)`

FreeRTOS. Queue

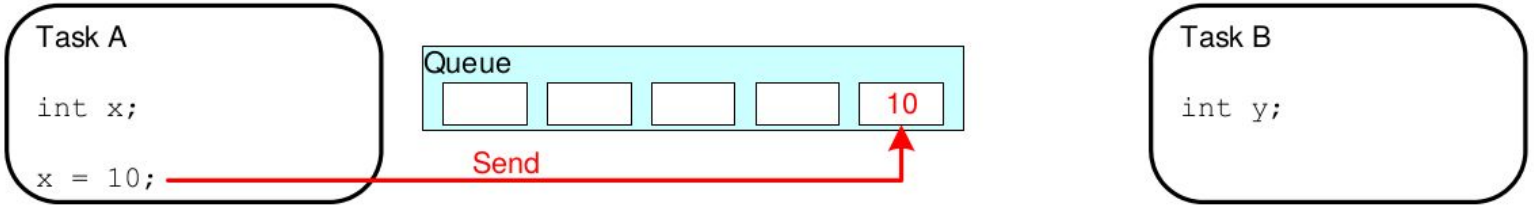
- Hold a finite number of fixed size data items (max number is called length of queue)
- Queues are used as FIFO
- Can be task-to-task, task-to-interrupt, interrupt-to-task
- Can block the current task

FreeRTOS. Queues



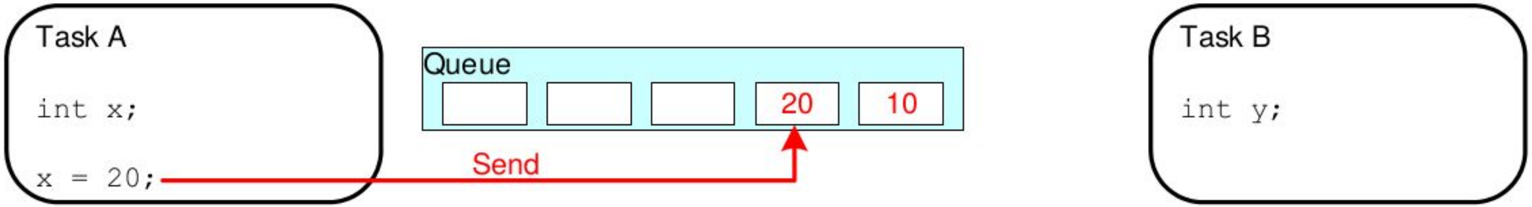
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

FreeRTOS. Queues



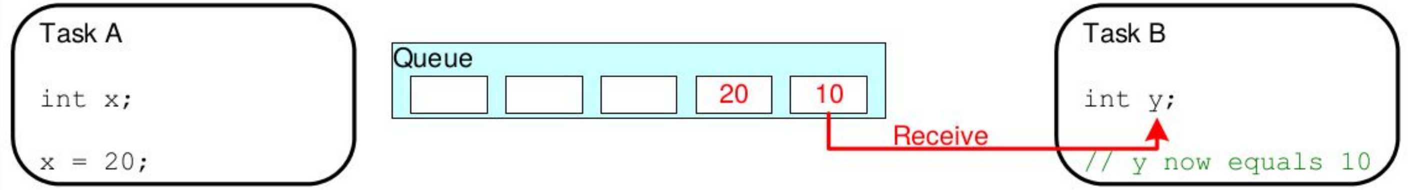
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

FreeRTOS. Queues



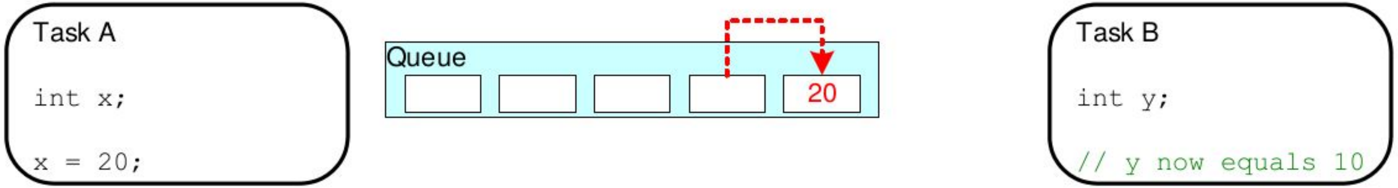
Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

FreeRTOS. Queues



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

FreeRTOS. Queues



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

FreeRTOS. Queues. Copying policy

- **Queue by copy (FreeRTOS)**
- Queue by reference

FreeRTOS. Queues. Features

- Can be accessed by multiple tasks
- Blocks on queue reads
- Might be blocked on write

FreeRTOS. Queues. API

- xQueueCreate()
- xQueueSendToBack()
- xQueueSendToBackFromISR()
- xQueueSendToFront()
- xQueueSendToFrontFromISR()
- xQueueReceive() (received object is removed from queue)
- xQueueReceiveFromISR()

More detailed doc

https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf