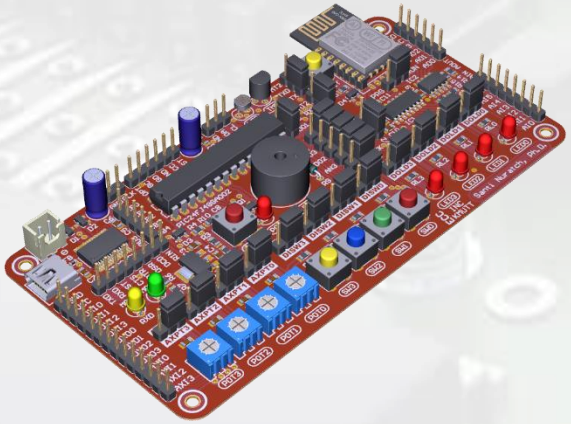
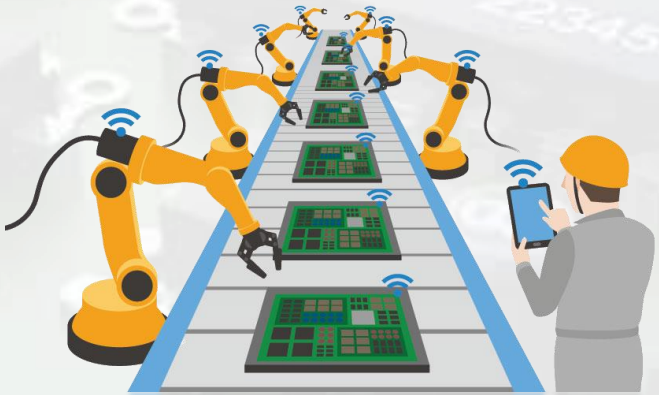
# Embedded Real-Time Operating Systems

## RTOS Programming for Embedded Developers






ดร.สันติ นุราช

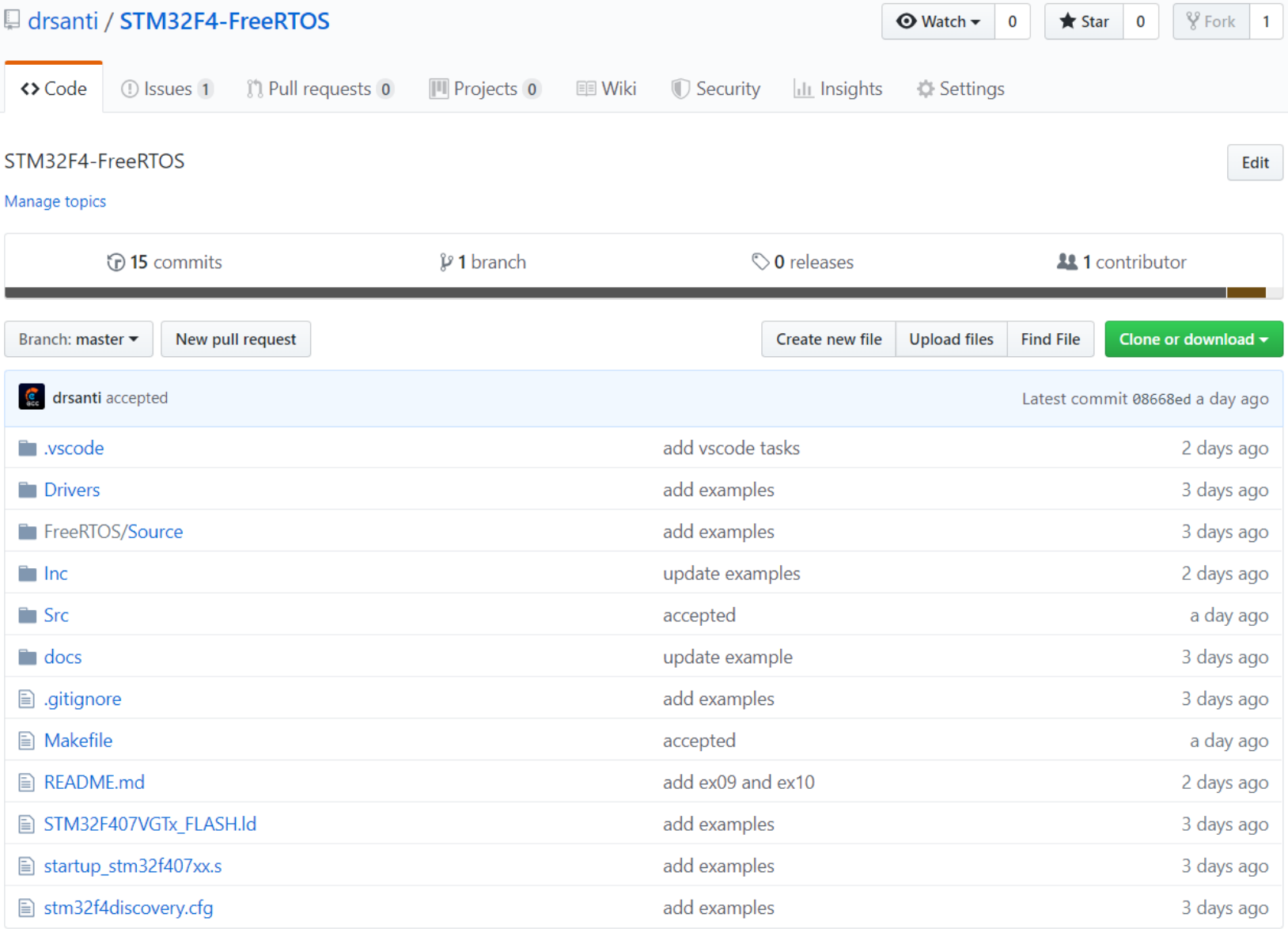
**Santi Nuratch., Ph.D.**

**Embedded Computing and Control Lab. @ INC-KMUTT**

santi.inc.kmutt@gmail.com, santi.nur@kmutt.ac.th

Department of Control System and Instrumentation Engineering,
King Mongkut's University of Technology Thonburi, **KMUTT**

# github.com/drsanti (all are there)



drsanti / STM32F4-FreeRTOS

Watch 0 | Star 0 | Fork 1

Code | Issues 1 | Pull requests 0 | Projects 0 | Wiki | Security | Insights | Settings

STM32F4-FreeRTOS

Edit

Manage topics

15 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

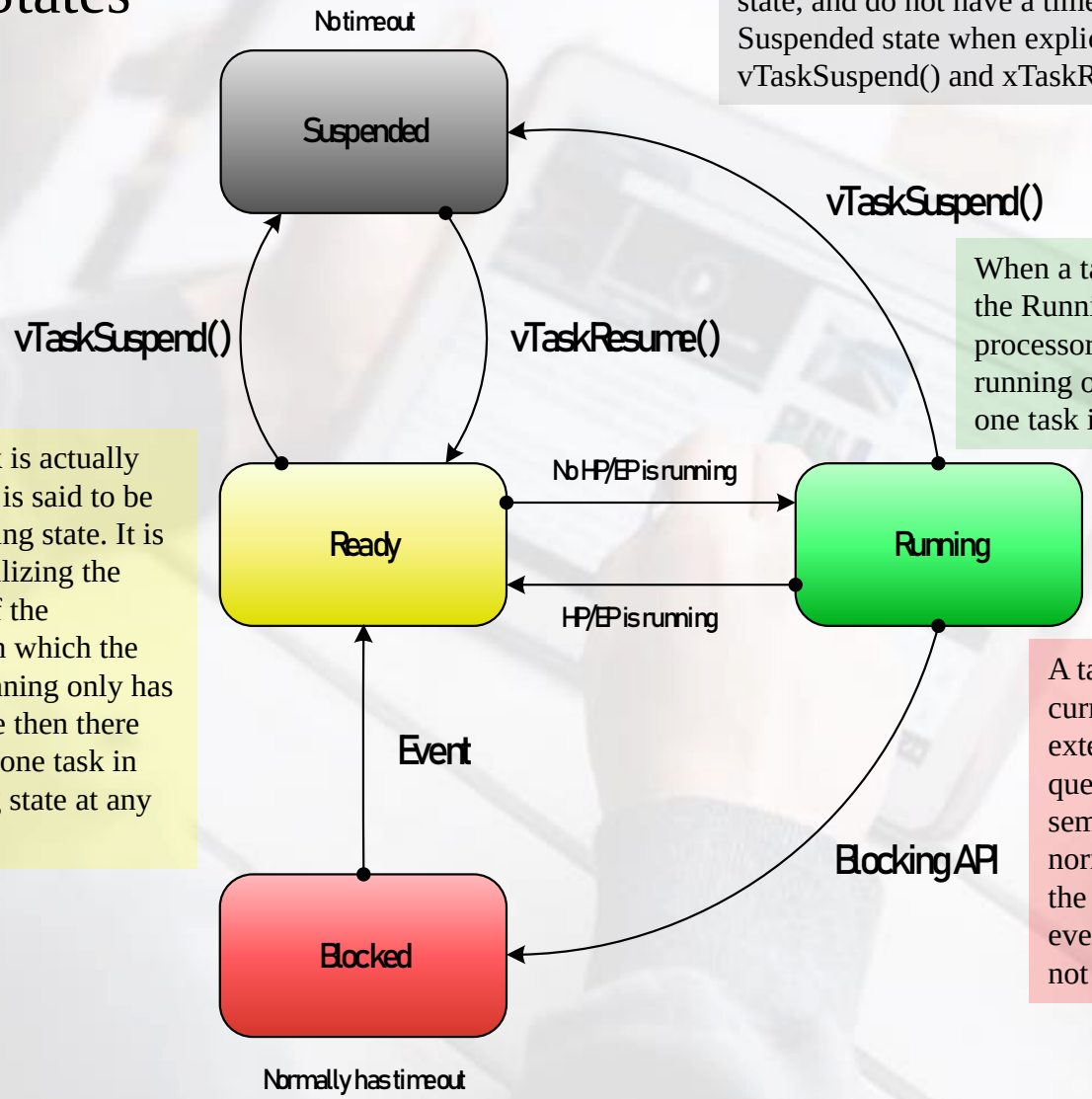
drsanti accepted — Latest commit 08668ed a day ago

| | | |
|---|---|---|
| .vscode | add vscode tasks | 2 days ago |
| Drivers | add examples | 3 days ago |
| FreeRTOS/Source | add examples | 3 days ago |
| Inc | update examples | 2 days ago |
| Src | accepted | a day ago |
| docs | update example | 3 days ago |
| .gitignore | add examples | 3 days ago |
| Makefile | accepted | a day ago |
| README.md | add ex09 and ex10 | 2 days ago |
| STM32F407VGTx_FLASH.ld | add examples | 3 days ago |
| startup_stm32f407xx.s | add examples | 3 days ago |
| stm32f4discovery.cfg | add examples | 3 days ago |

# Tasks

## Task States

Tasks in the Suspended state cannot be selected to enter the Running state, and do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.



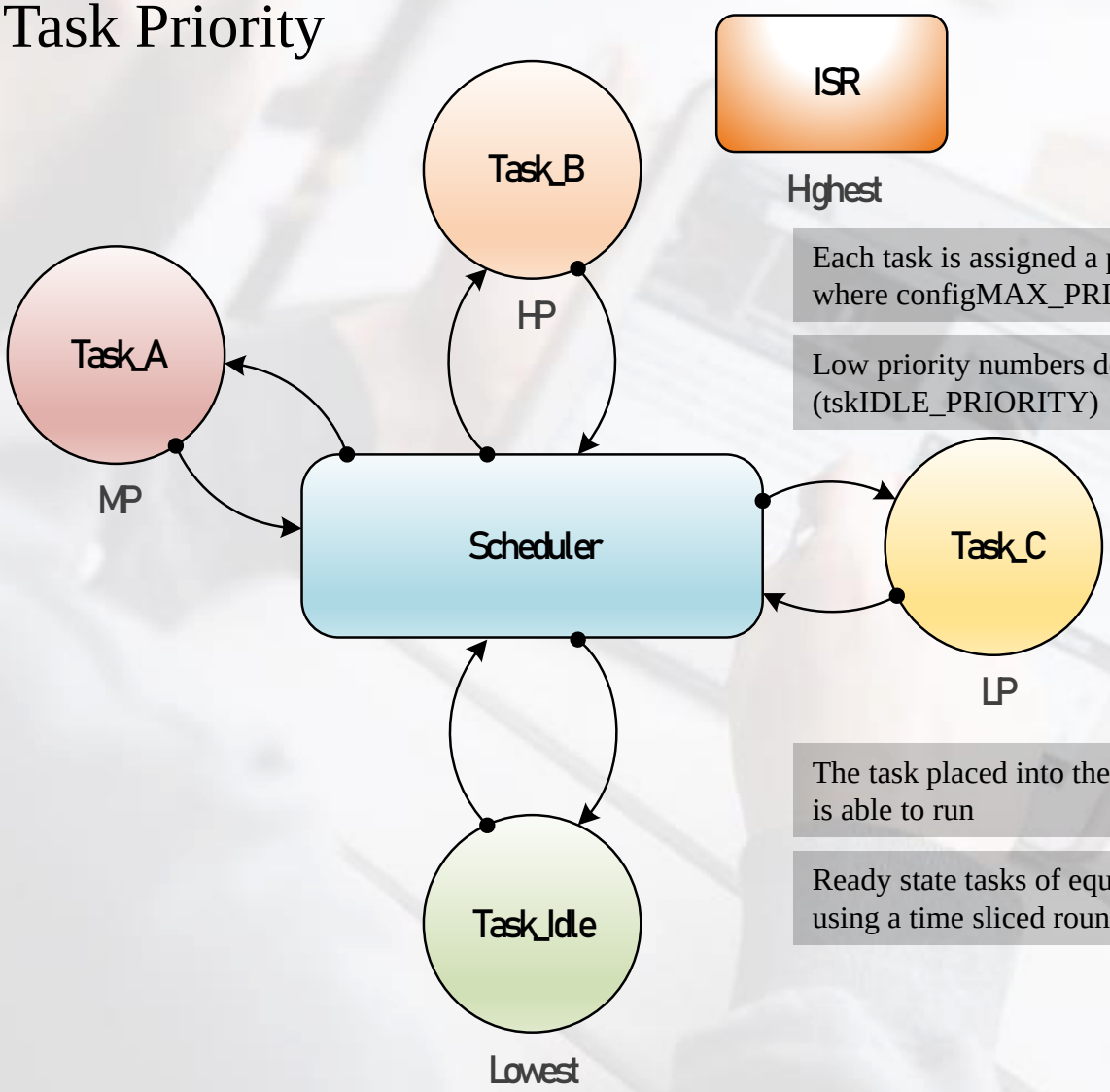

When a task is actually executing it is said to be in the Running state. It currently utilizing the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.

# Tasks

## Task Priority




Each task is assigned a priority from **0** to **configMAX_PRIORITIES-1** where configMAX_PRIORITIES is defined within FreeRTOSConfig.h

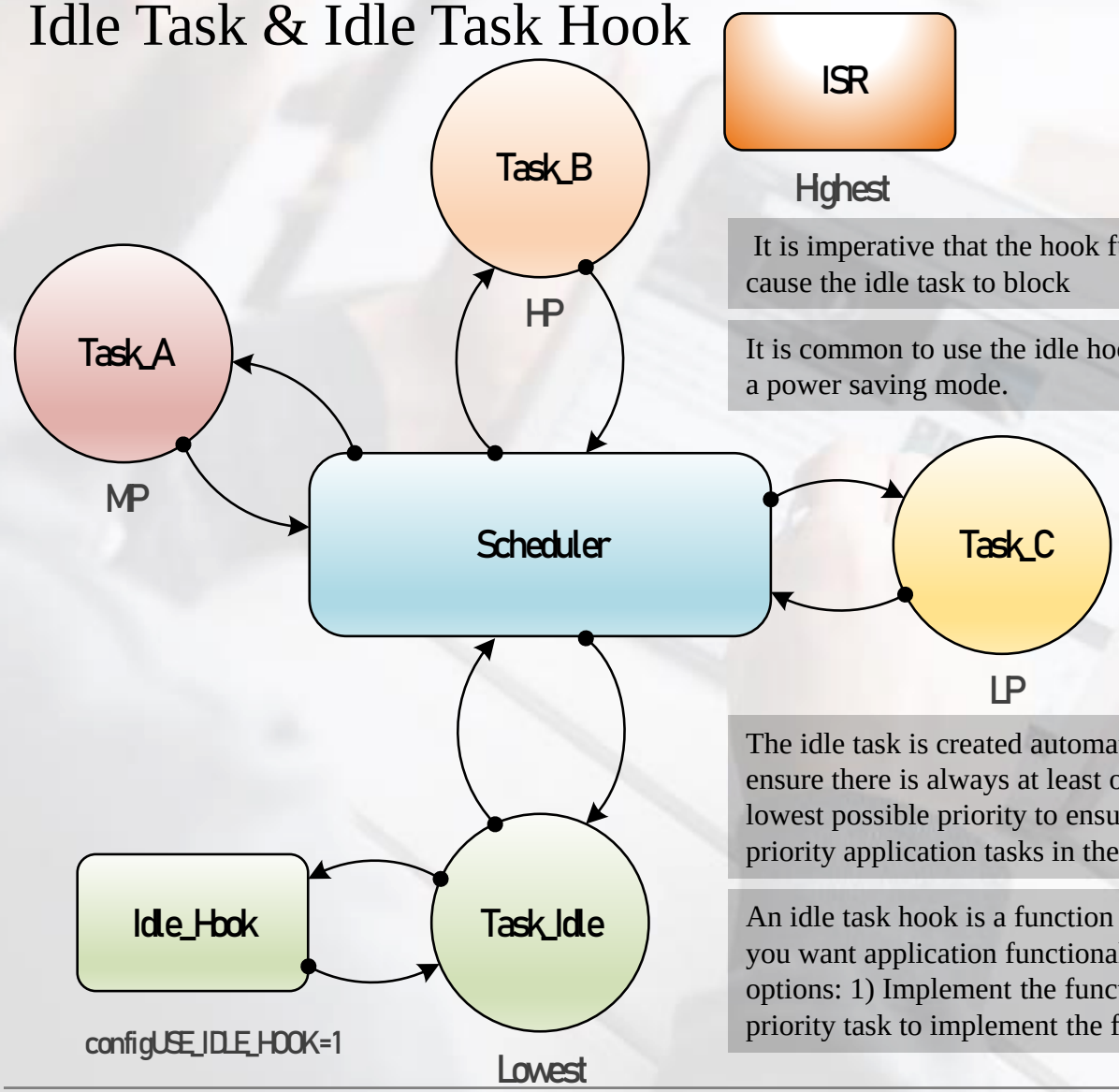
Low priority numbers denote low priority tasks. The idle task has priority zero (tskIDLE_PRIORITY)

The task placed into the Running state is always the highest priority task that is able to run

Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme.

# Tasks

## Idle Task & Idle Task Hook




It is imperative that the hook function does not call any API functions that might cause the idle task to block

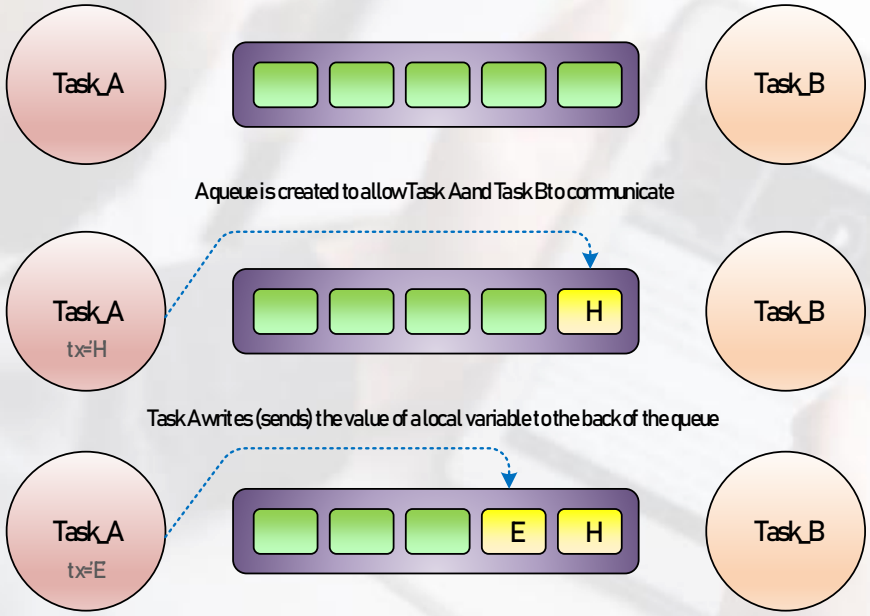
It is common to use the idle hook function to place the microcontroller CPU into a power saving mode.

The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run. It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

An idle task hook is a function that is called during each cycle of the idle task. If you want application functionality to run at the idle priority then there are two options: 1) Implement the functionality in an idle task hook. 2) Create an idle priority task to implement the functionality

# Queues, Mutexes, Semaphores

## Queue (First In First Out, FIFO)



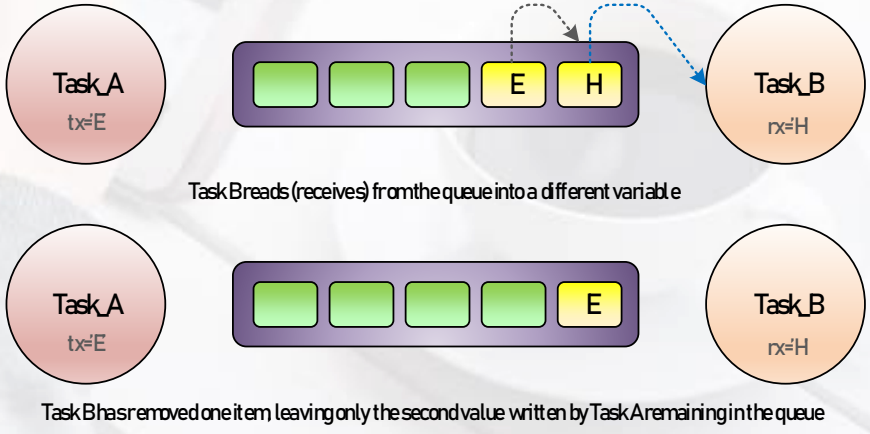

A queue can hold a finite number of fixed size data items. The maximum number of items a
queue can hold is called its 'length'. Both the length and the size of each data item are set
when the queue is created.

There are two ways in which queue behavior could have been implemented:

1) Queue by copy: Queuing by copy means the data sent to the queue is copied byte for byte into the queue.
2) Queue by reference: Queuing by reference means the queue only holds pointers to the data sent to the queue, not the data itself

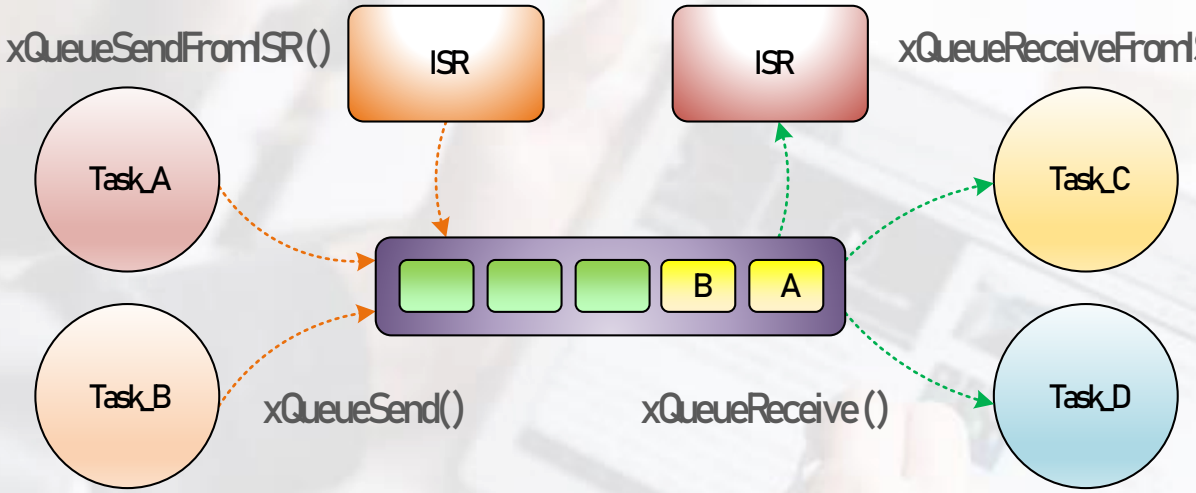
FreeRTOS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queueing by reference

Queues are objects in their own right that can be accessed by any task or ISR that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. **In practice** it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers



# Queues, Mutexes, Semaphores

## Queue (Accessing and Blocking)




Queue can be accessed by multiple tasks including ISRs

Queues can be grouped into sets, allowing a task to enter the Blocked state to wait for data to become available on any of the queues in the set

When a task attempts to read from a queue, it can optionally specify a 'block' time. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue

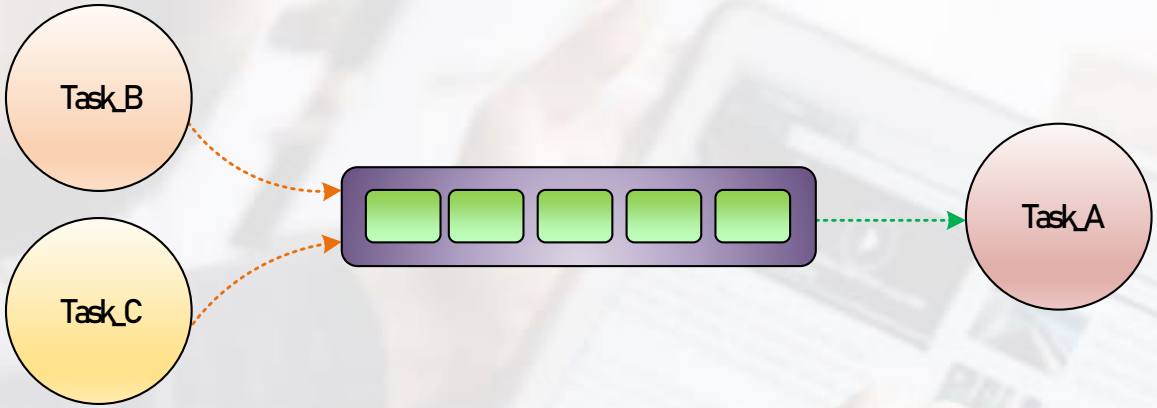
Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data. If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked

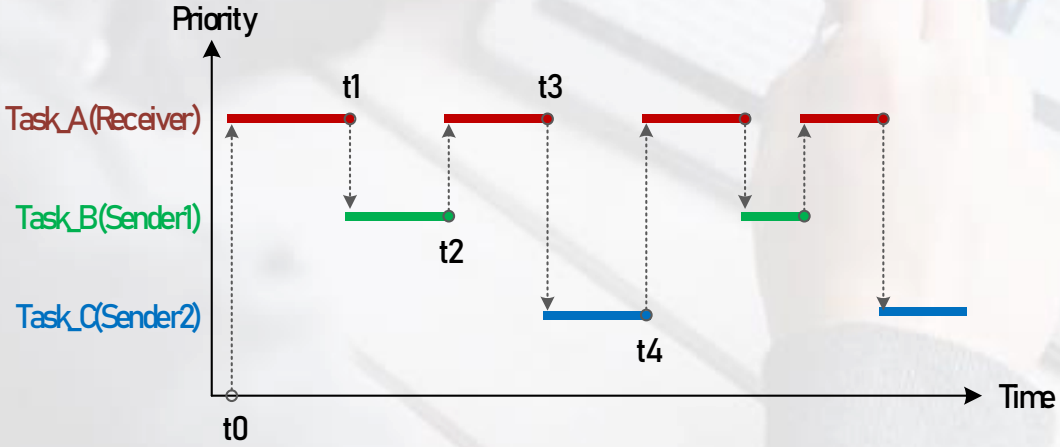
Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue

Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available.

# Queues, Mutexes, Semaphores

## Queue (Sequence of Execution)




**t0:** **Task_A** runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the **Task_A** enters the Blocked state to wait for data to become available

**t1:** **Task_B** runs after the **Task_A** has blocked

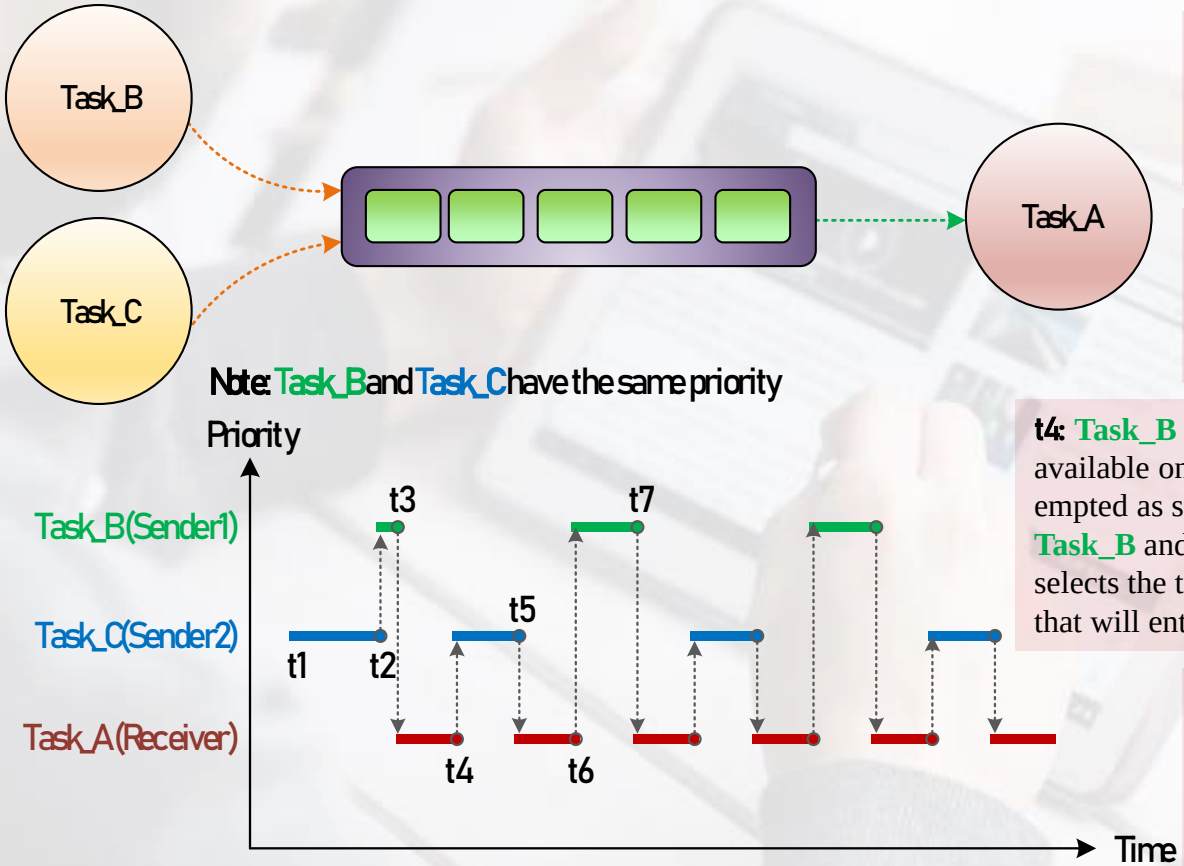
**t2:** **Task_B** writes to the queue, causing the **Task_A** to exit the Blocked state. The **Task_A** has the highest priority so pre-empts **Task_B**

**t3:** **Task_A** empties the queue then enters the Blocked state again. This time **Task_C** runs after the **Task_A** has blocked

**t4:** **Task_C** writes to the queue, causing the **Task_A** to exit the Blocked state and pre-empt **Task_C** - and so it goes on

# Queues, Mutexes, Semaphores

## Queue (Sequence of Execution)




**t1:** **Task_C** executes and sends 5 data items to the queue

**t2:** The queue is full so **Task_C** enters the Blocked state to wait for its next send to complete. **Task_B** is able to run, so enters the Running state

**t3:** **Task_B** finds the queue is already full, so enters the Blocked state to wait for its first send to complete. **Task_A** is now able to run, so enters the Running state

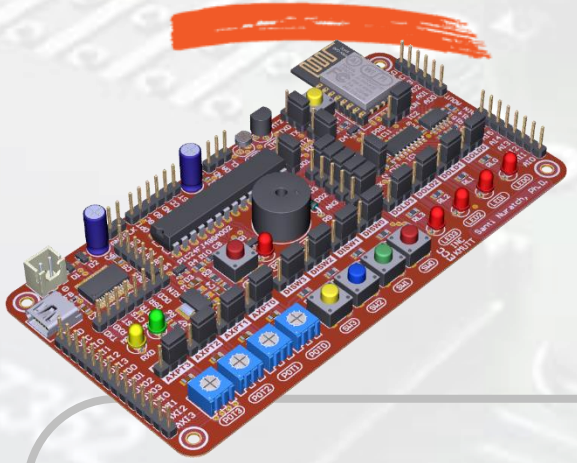
**t4:** **Task_B** and **Task_C** are waiting for space to become available on the queue, resulting in **Task_A** being pre-empted as soon as it has removed one item from the queue. **Task_B** and **Task_C** have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state

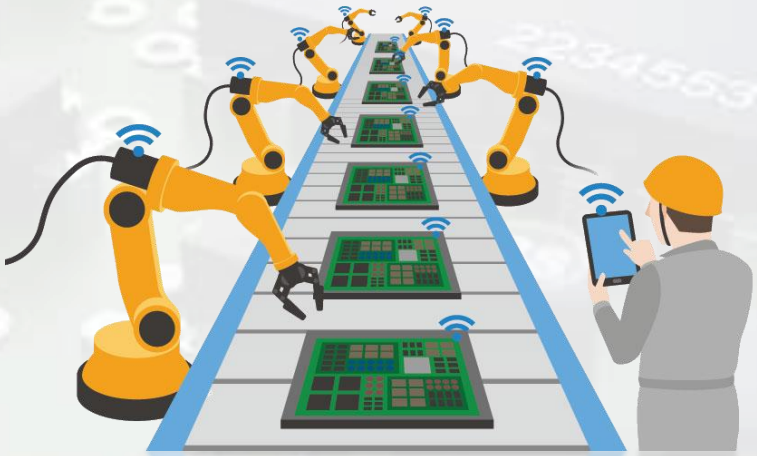
**t5:** **Task_C** sends another data item to the queue. There was only one space in the queue, so task **Task_C** enters the Blocked state to wait for its next send to complete. Task **Task_A** is again able to run so enters the Running state

**t6:** Both **Task_B** and **Task_C** have other items to be sent to the queue. This time **Task_B** has been waiting longer than **Task_C**, so **Task_B** enters the Running state (**Task_A** is pre-empted)

**t7:** **Task_B** sends data item to the queue. The, there was no space in the queue so **Task_B** enters the Blocked state to wait for its next send to complete. Both tasks **Task_C** and **Task_B** are waiting for space to become available on the queue, so task **Task_A** is the only task that can enter the Running state

# THANK YOU!






## ดร.สันติ นุราช

**Santi Nuratch., Ph.D.**

**Embedded Computing and Control Lab. @ INC-KMUTT**

santi.inc.kmutt@gmail.com, santi.nur@kmutt.ac.th

Department of Control System and Instrumentation Engineering,
King Mongkut's University of Technology Thonburi, **KMUTT**