# CE5045
# Embedded System Design

## FreeRTOS and Embedded Platform

https://github.com/tychen-NCU/EMBS-NCU

Instructor: Dr. Chen, Tseng-Yi

Computer Science & Information Engineering

# Outline

- FreeRTOS
  - ✓ What is FreeRTOS
  - ✓ Kernel Overview
  - ✓ Tasks versus Co-Routines
  - ✓ Task Details
  - ✓ IPC and Synchronization in FreeRTOS
- Embedded Platforms
  - ✓ Arduino UNO R3
  - ✓ Circuit.io On-line Emulator

# Outline

- FreeRTOS
  - ✓ What is FreeRTOS
  - ✓ Kernel Overview
  - ✓ Tasks versus Co-Routines
  - ✓ Task Details
  - ✓ IPC and Synchronization in FreeRTOS
- Embedded Platforms
  - ✓ Arduino UNO R3
  - ✓ Circuit.io On-line Emulator

# Background Information

- The FreeRTOS Project supports **25 official architecture ports**, with many more community developed ports
- The FreeRTOS RT kernel is **portable**, **open source**, **royalty free**, and **very small**.
- OpenRTOS is a commercialized version by the sister company High Integrity Systems
- FreeRTOS has been used in some rockets and other aircraft, but nothing too commercial

# FreeRTOS Configuration

- The operation of FreeRTOS is governed by `FreeRTOS.h`, with application specific configuration appearing in `FreeRTOSConfg.h`.

- Obviously, these are **static configuration options**.

- Some examples:
  - ✓ `configUSE_PREEMPTION`
  - ✓ `configCPU_CLOCK_HZ` – CPU frequency.
  - ✓ `configTICK_RATE_HZ` – RTOS
  - ✓ `configMAX_PRIORITIES` – To new list, so memory sensitive mach

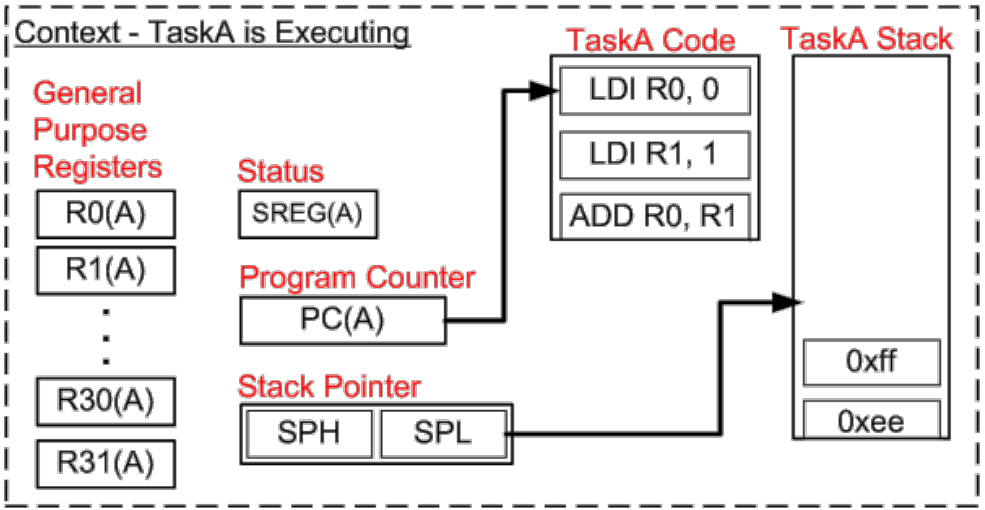
```
87  #ifndef configUSE_PREEMPTION
88          #error Missing definition:  configUSE_PREEMPTION should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configura
89  #endif
90
91  #ifndef configUSE_IDLE_HOOK
92          #error Missing definition:  configUSE_IDLE_HOOK should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configurat
93  #endif
94
95  #ifndef configUSE_TICK_HOOK
96          #error Missing definition:  configUSE_TICK_HOOK should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configurat
97  #endif
98
99  #ifndef configUSE_CO_ROUTINES
100         #error  Missing definition:  configUSE_CO_ROUTINES should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configu
101 #endif
102
103 #ifndef INCLUDE_vTaskPrioritySet
104         #error Missing definition:  INCLUDE_vTaskPrioritySet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Confi
105 #endif
106
107 #ifndef INCLUDE_uxTaskPriorityGet
108         #error Missing definition:  INCLUDE_uxTaskPriorityGet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Conf
109 #endif
110
111 #ifndef INCLUDE_vTaskDelete
112         #error Missing definition:  INCLUDE_vTaskDelete         should be defined in FreeRTOSConfig.h as either 1 or 0.  See the C
113 #endif
```

# RTOS Fundamentals

- Introduction to real time and multitasking concepts
  - ✓ Multitasking
  - ✓ Scheduling
  - ✓ Context Switching
  - ✓ Real Time Applications
  - ✓ Real Time Scheduling

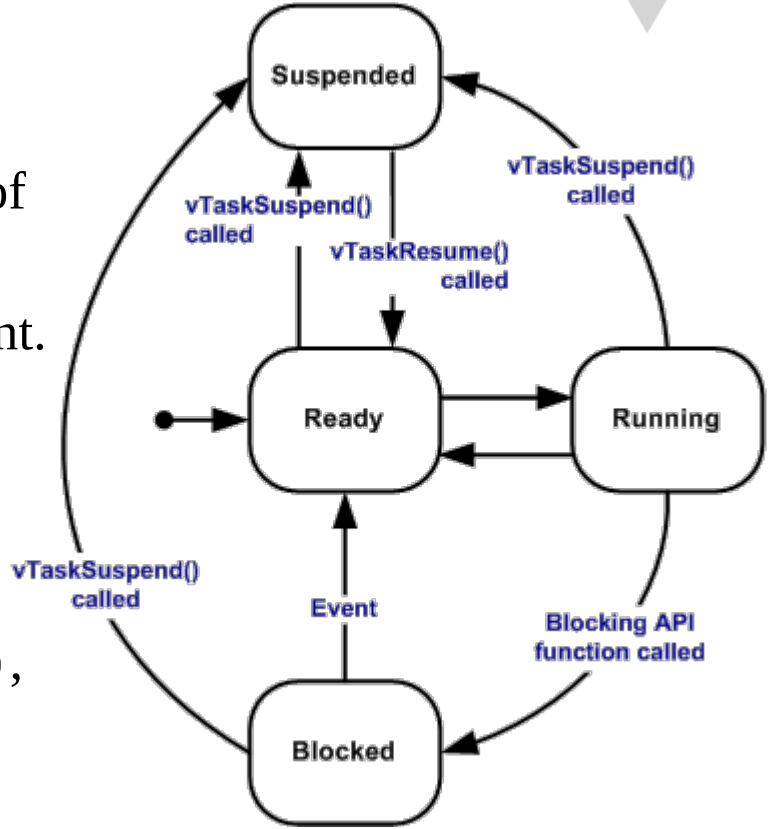
# Tasks in FreeRTOS

- Tasks have their own context. No dependency on other tasks unless defined.
- One task executes at a time.
- Tasks have no knowledge of scheduler activity. The scheduler handles context switching.
- Thus, tasks each have their own stack upon which execution context can be saved.
- Prioritized and preemptable



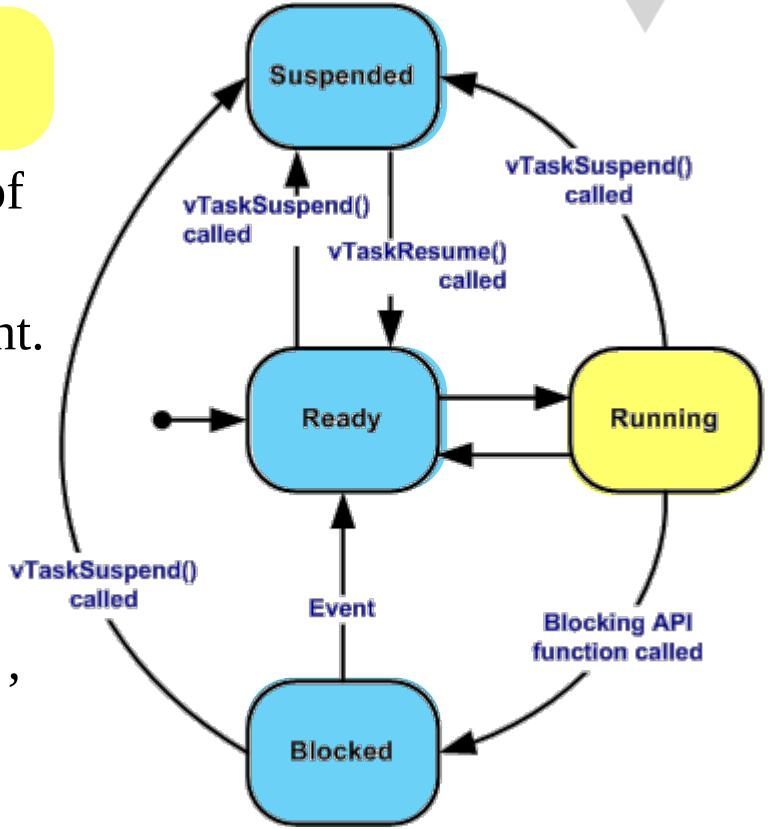
# Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the **Running** state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



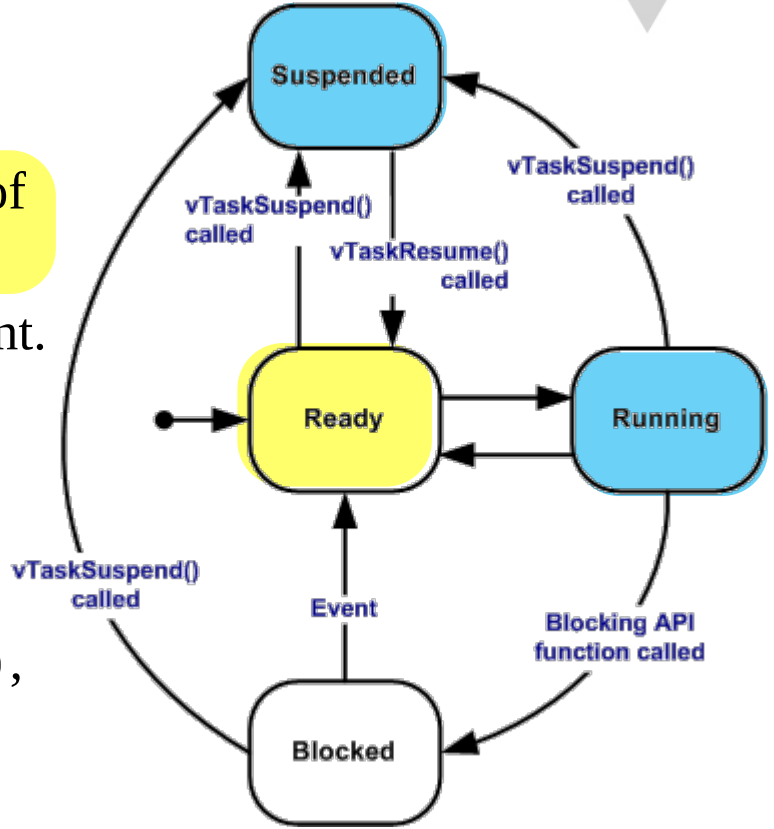
# Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the **Running** state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



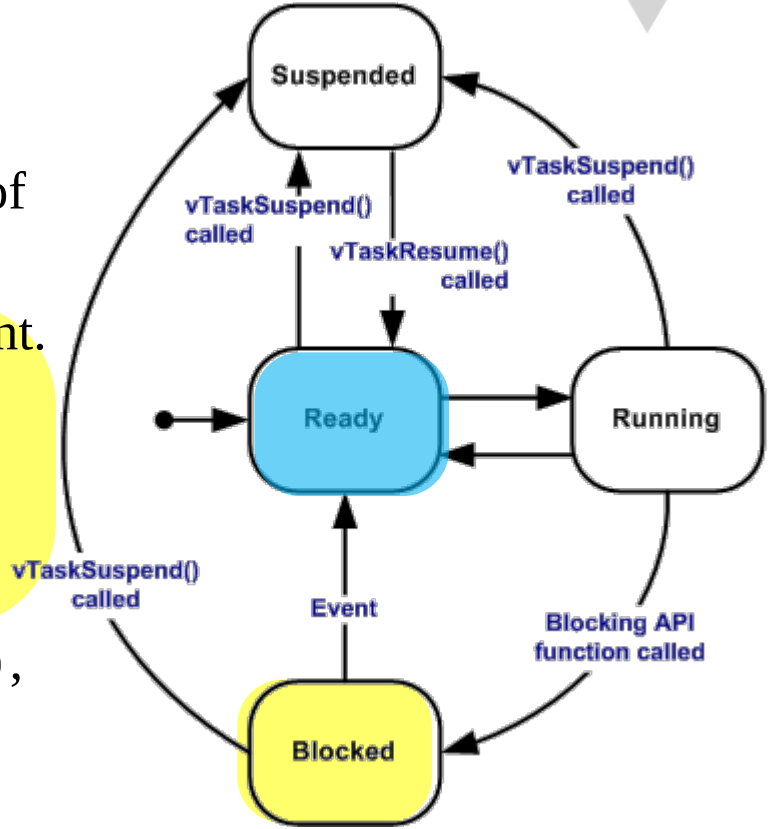
# Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the **Running** state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



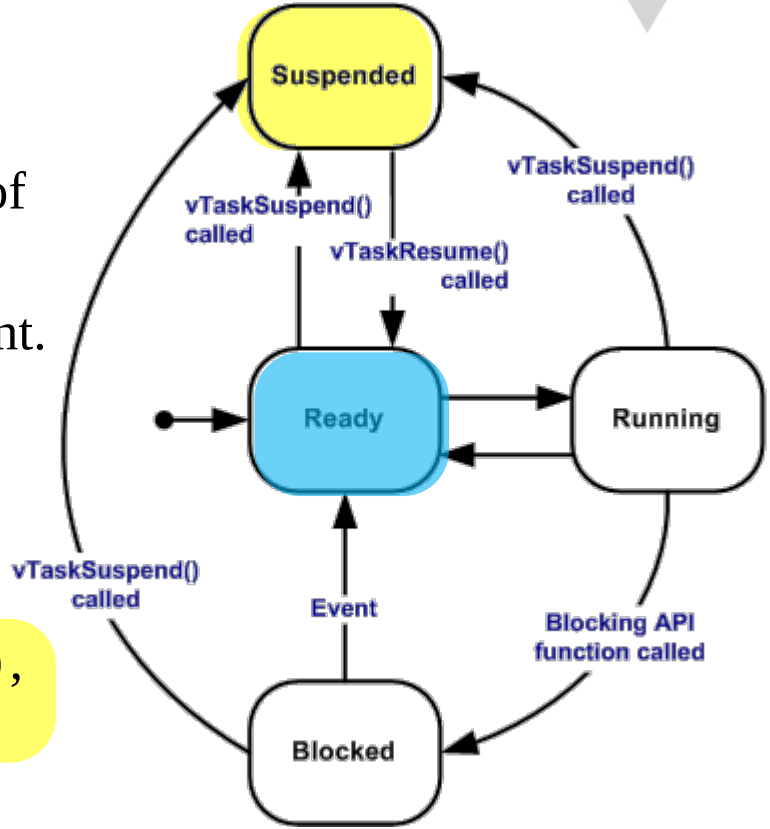
# Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the **Running** state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



# Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the **Running** state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



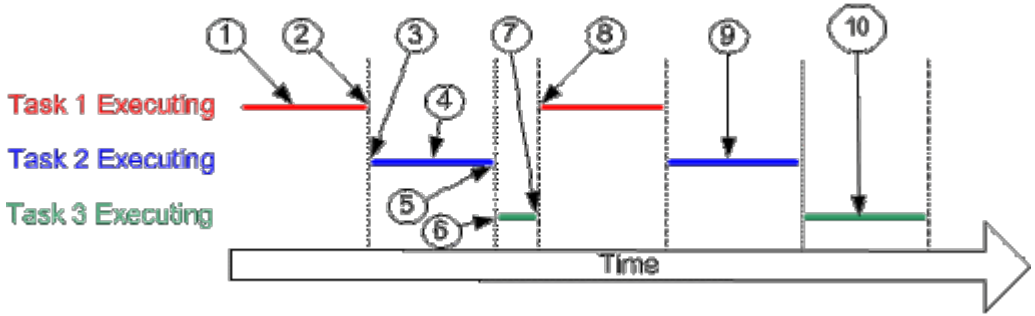
# RTOS - Multitasking

- Each program is **a task** under control of the OS
  - ✓ Allow users access to resources **seemingly simultaneously** multiple tasks can execute apparently concurrently
- Advantages
  - ✓ Multitasking and **inter-task communications** features
  - ✓ Easier and more efficient development
  - ✓ “Disregard” complex timing and sequencing details

# RTOS - Scheduling

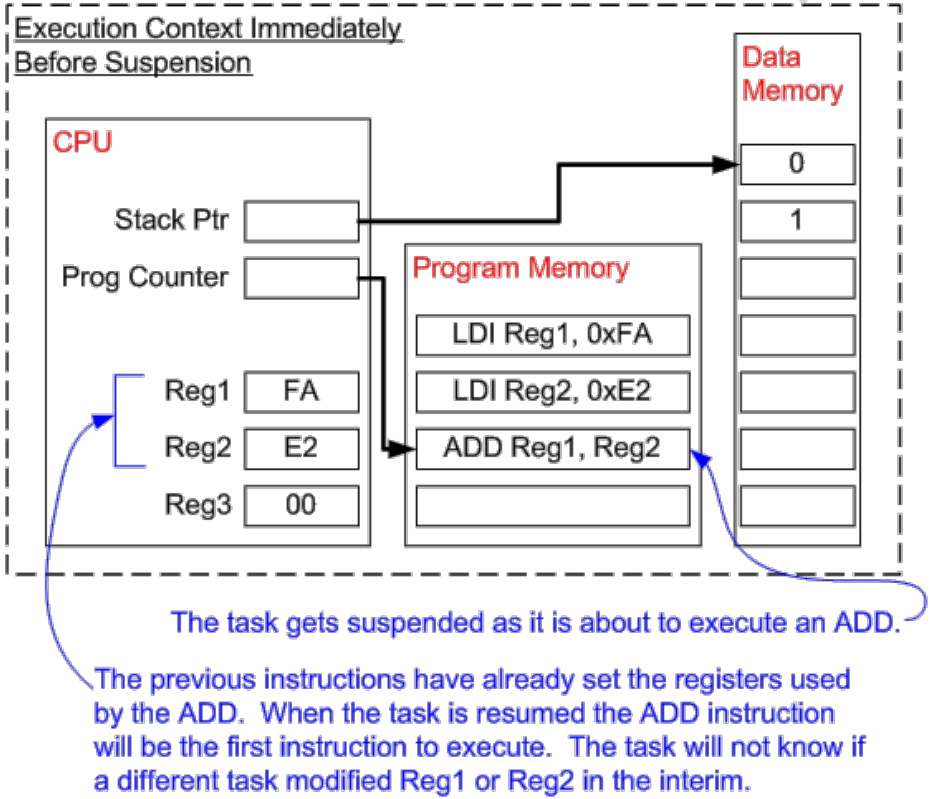
- Algorithm used to **decide the task to execute**
  - ✓ Tasks can be **suspend/resume** many times during its lifetime
  - ✓ "**Fair**" proportions of CPU time are allocated to tasks
  - ✓ A task can also suspend itself (e.g., **delay** (**sleep**) for a fixed period, or **wait** (**block**) for a resource to become available)

Example:
2: T1 preemption
4: T2 lock a resource R
7: try access resource R
9: release lock on R
10: lock resource R



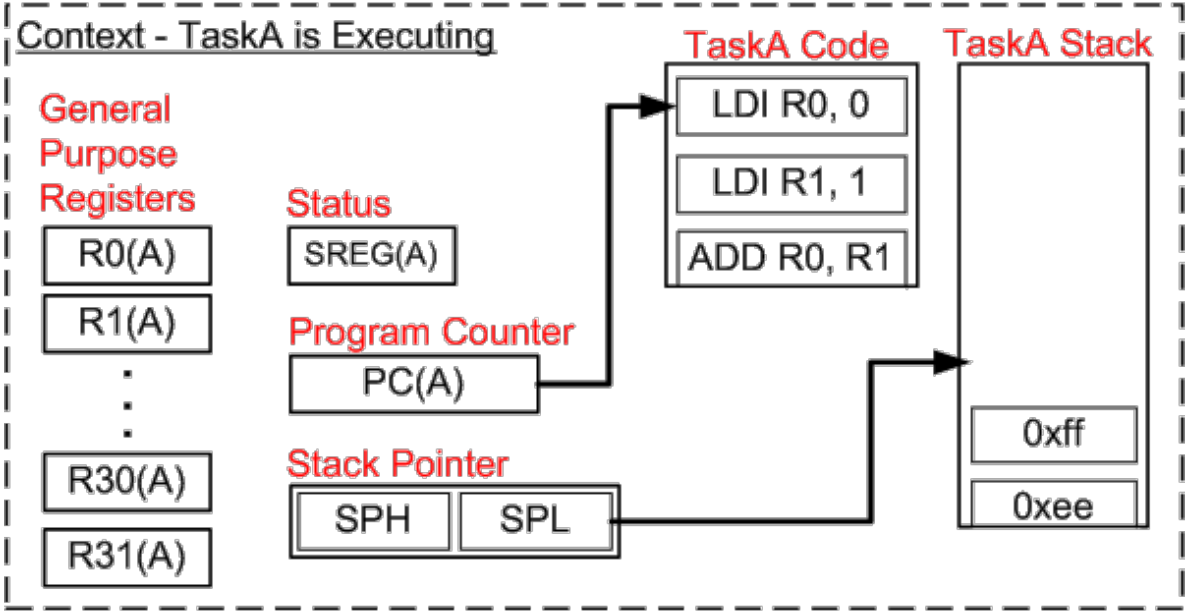
# RTOS – Context Switch

- Task execution requires **exclusive usage** of some computation resources
  - ✓ e.g. CPU registers and some RAM memory
- A task **does not know** when it is going to get suspended or resumed by the kernel
  - ✓ Does not even know when this has happened



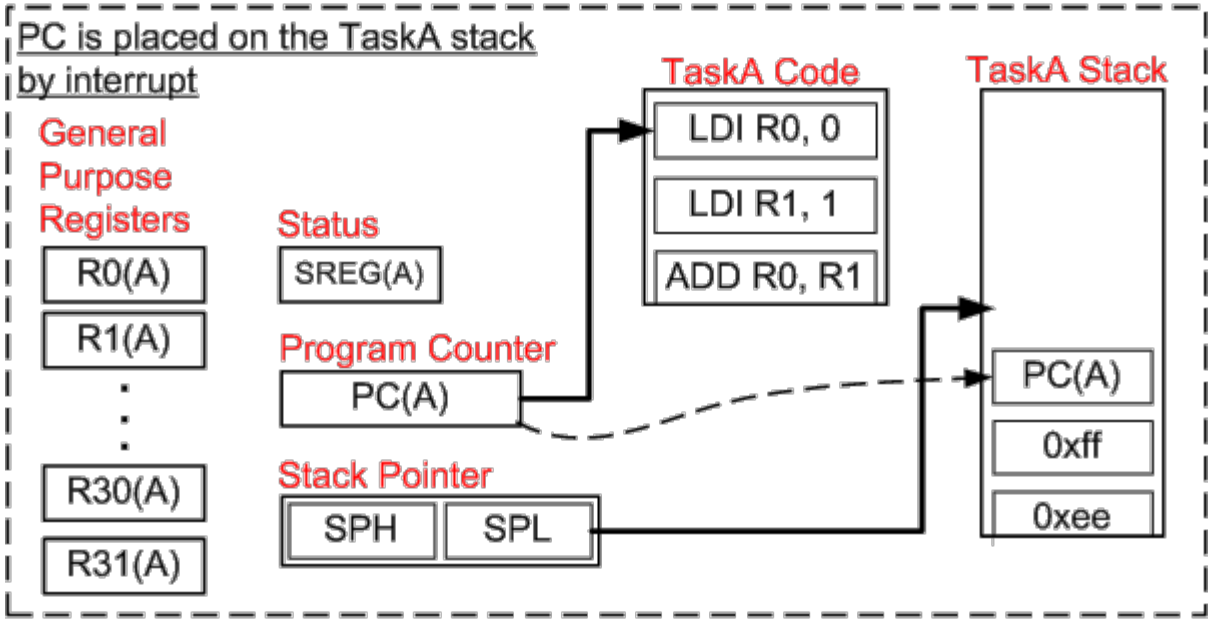
# Context Switch in FreeRTOS (Step 1)

- **Status:** TaskB has previously been suspended so its context has already been stored on the TaskB stack.
- **Now**: TaskA has the context demonstrated by the diagram below.



# Context Switch in FreeRTOS (Step 2)

- **Status:** The RTOS tick occurs just as TaskA is about to execute an LDI instruction.
- **Now**: When the interrupt occurs, the AVR microcontroller automatically places the current program counter (PC) onto the stack.



# Context Switch in FreeRTOS (Step 3)

- **Status:** Jump to Interrupt service routine (ISR).
- **Now**: `portSAVE_CONTEXT()` pushes the entire AVR execution context onto the stack of TaskA.

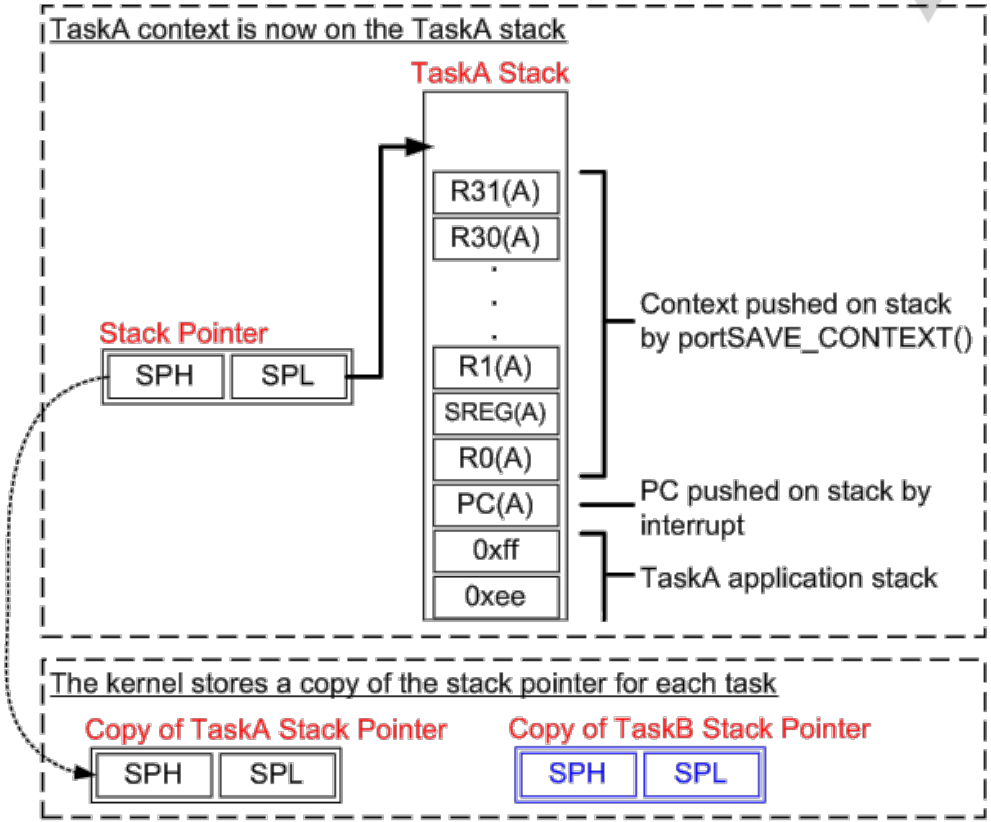
```
/* Interrupt service routine for
the RTOS tick. */
void SIG_OUTPUT_COMPARE1A( void )
{
    vPortYieldFromTick();
    asm volatile ( "reti" );
}
```

```
void vPortYieldFromTick( void )
{
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ( "ret" );
}
```

# Context Switch in FreeRTOS (Step 3)

- **Status:** Jump to Interrupt service routine (ISR).

- **Now**: `portSAVE_CONTEXT()` pushes the entire AVR execution context onto the stack of TaskA.



# Context Switch in FreeRTOS (Step 4)

- The RTOS function `vTaskIncrementTick()` executes after the **TaskA** context has been saved.

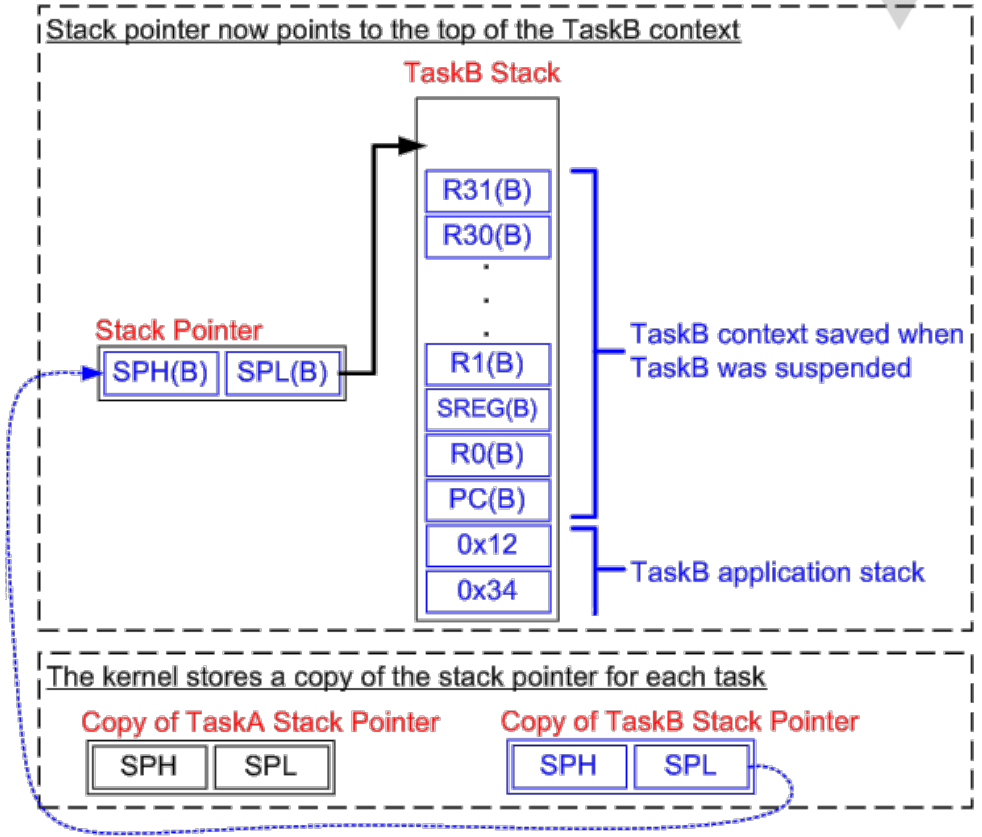
- Some assumptions:
  - ✓ Incrementing the tick count has caused TaskB to become **ready to run**
  - ✓ **TaskB** has a higher priority than **TaskA**

- `vTaskSwitchContext()` selects TaskB as the task to be given processing time when the ISR completes.

# Context Switch in FreeRTOS (Step 5)

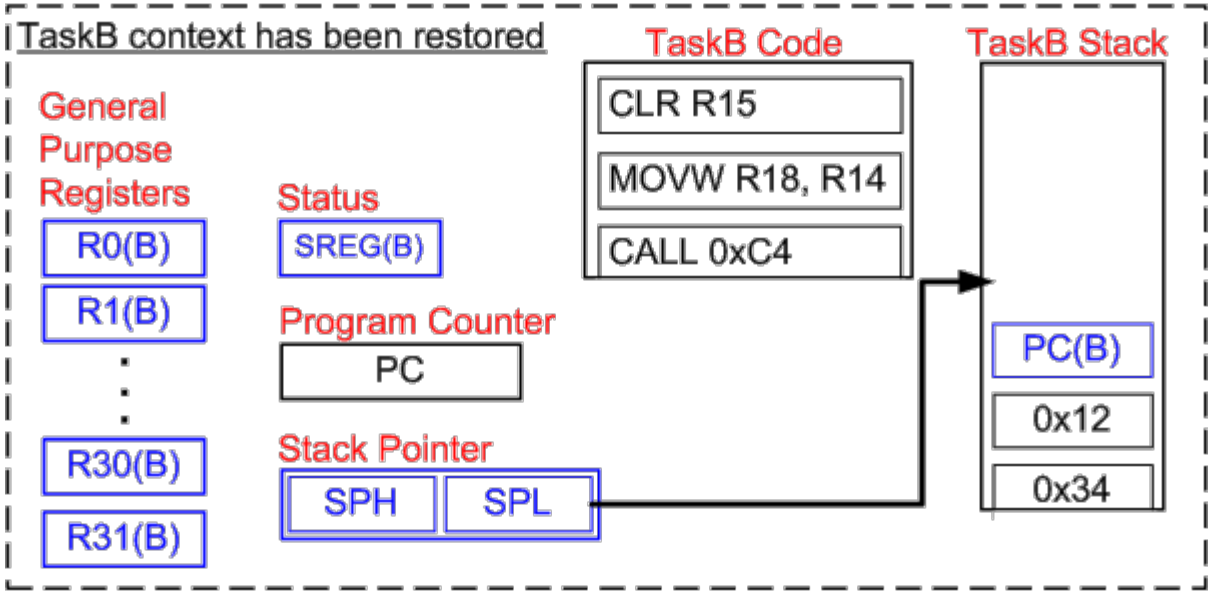
- **Status:** The TaskB context must be restored..

- **Now**: The first thing RTOS macro `portRESTORE_CONTEXT` does is to retrieve the TaskB stack pointer from the copy taken when TaskB was suspended.



# Context Switch in FreeRTOS (Step 6)

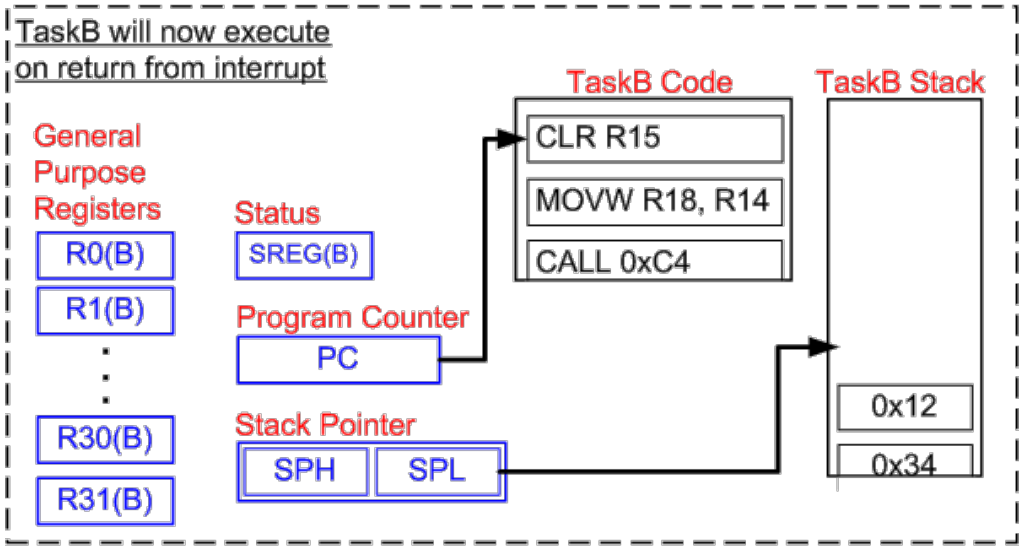
- **Status:** `portRESTORE_CONTEXT()` completes by restoring the TaskB context from its stack into the appropriate processor registers..
- **Now**: Only the program counter remains on the stack.



# Context Switch in FreeRTOS (Step 7)

- `vPortYieldFromTick()` returns to `SIG_OUTPUT_COMPARE1A()` where the final instruction is a return from interrupt (`RETI`).

- The RTOS tick interrupt interrupted **TaskA**, but is returning to **TaskB** – the context switch is complete!



# Task Priorities

- Each task gets a priority from 0 to `configMAX_PRIORITIES -1`
- Priority can be set on **per application** basis
- Tasks can change **their own priority**, as well as the priority of other tasks
- `tskIDLE_PRIORITY` = 0

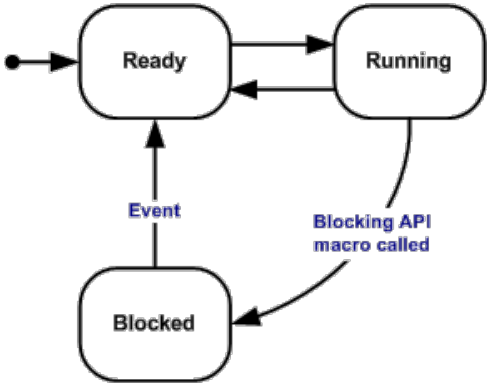
# The Idle Task

- The idle task is **created automatically** when the scheduler is started.
- It **frees** memory allocated by the RTOS to tasks that have since been **deleted**
- Thus, applications that use `vTaskDelete()` to remove tasks should ensure the idle task **is not starved**
- The idle task has **no other active functions** so can legitimately be starved of microcontroller time under all other conditions
- There is an **idle task hook**, which can do some work **at each idle interval** without the RAM usage overhead associated with running a task at the idle priority.

# Co-Routines

- Co-routines are only intended for use on very **small processors** that have **severe RAM constraints**, and would **not normally** be used on 32-bit microcontrollers

- A co-routine can exist in one of the following states:
  - ✓ **Running**: When a co-routine is actually executing it is said to be in the Running state. It is currently utilising the processor
  - ✓ **Ready**: Another co-routine of equal or higher priority is already in the Running state, or a task is in the Running state
  - ✓ **Blocked**: A temporal or external event (e.g., `crDELAY()`)



# The Limitations of Co-Routines

- Co-Routines share **a single stack**
- Prioritized relative to **other co-routines**, but **preempted by tasks**
- The structure of co-routines is rigid due to the unconventional implementation.
  - ✓ Lack of stack requires special consideration
  - ✓ Restrictions on where API calls can be made
  - ✓ Cannot communicate with tasks