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

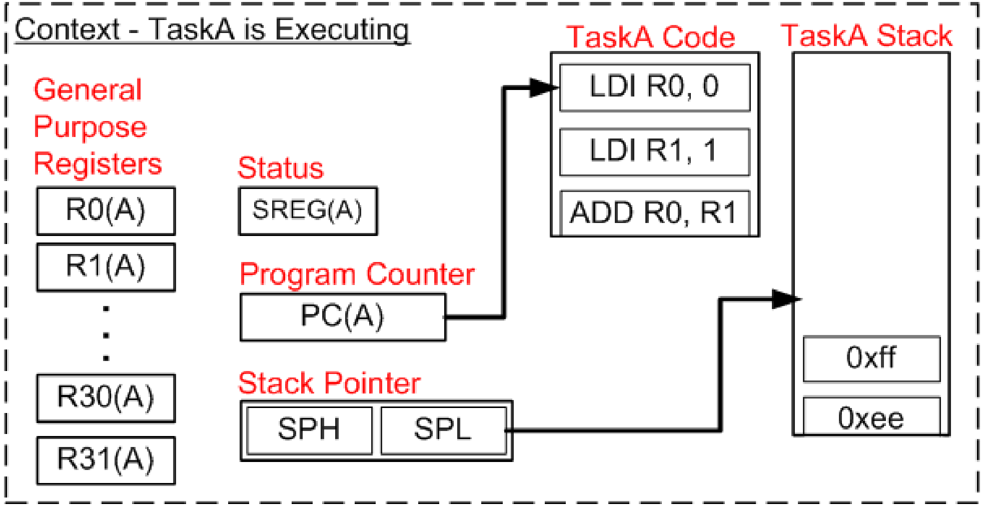
```
#ifndef configUSE_PREEMPTION
	#error Missing definition:  configUSE_PREEMPTION should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configura
#endif

#ifndef configUSE_IDLE_HOOK
	#error Missing definition:  configUSE_IDLE_HOOK should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configurat
#endif

#ifndef configUSE_TICK_HOOK
	#error Missing definition:  configUSE_TICK_HOOK should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configurat
#endif

#ifndef configUSE_CO_ROUTINES
	#error  Missing definition:  configUSE_CO_ROUTINES should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configu
#endif

#ifndef INCLUDE_vTaskPrioritySet
	#error Missing definition:  INCLUDE_vTaskPrioritySet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Confi
#endif

#ifndef INCLUDE_uxTaskPriorityGet
	#error Missing definition:  INCLUDE_uxTaskPriorityGet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Conf
#endif

#ifndef INCLUDE_vTaskDelete
	#error Missing definition:  INCLUDE_vTaskDelete          should be defined in FreeRTOSConfig.h as either 1 or 0.  See the C
#endif
```

RTOS Fundamentals

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

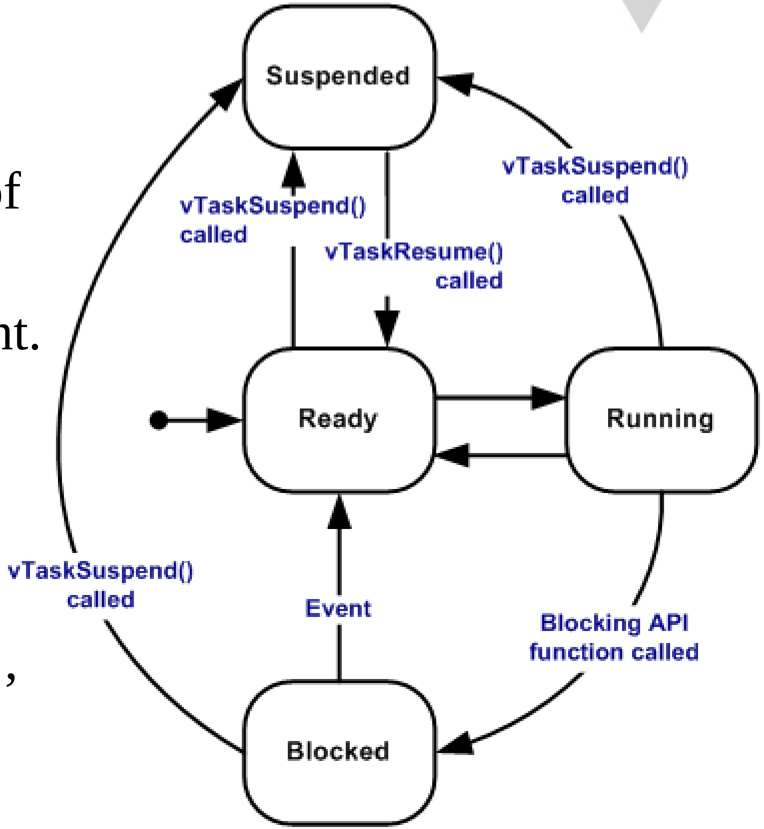
Tasks in FreeRTOS

- 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



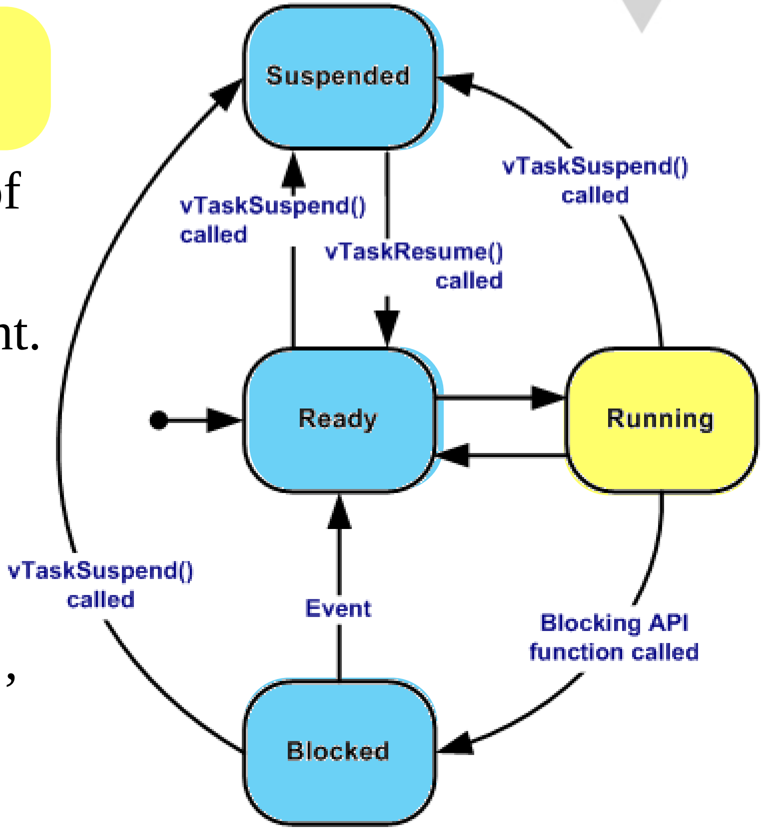
Task States

- **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.



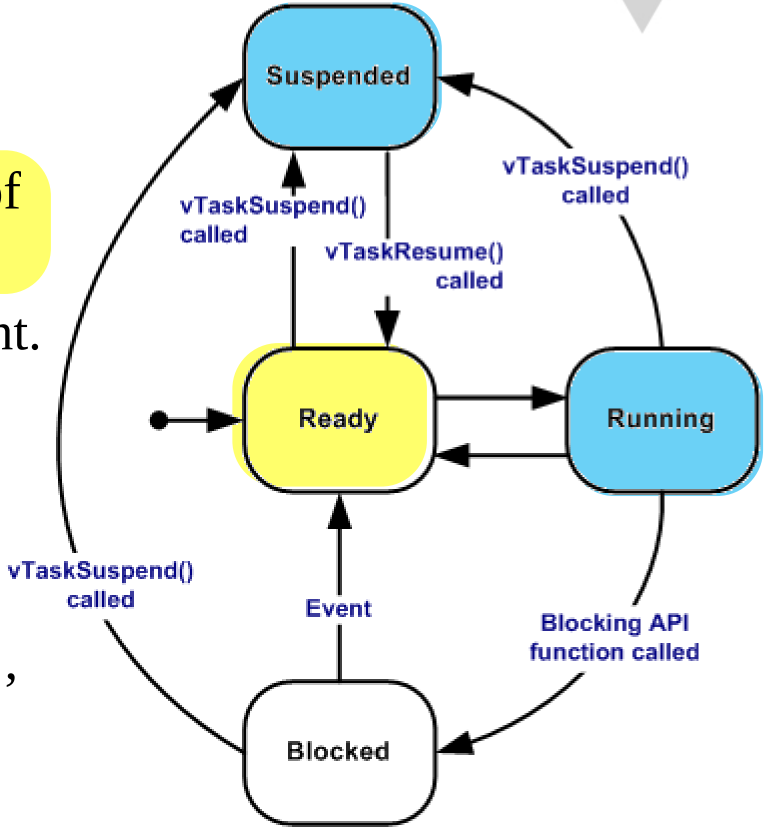
Task States

- **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.



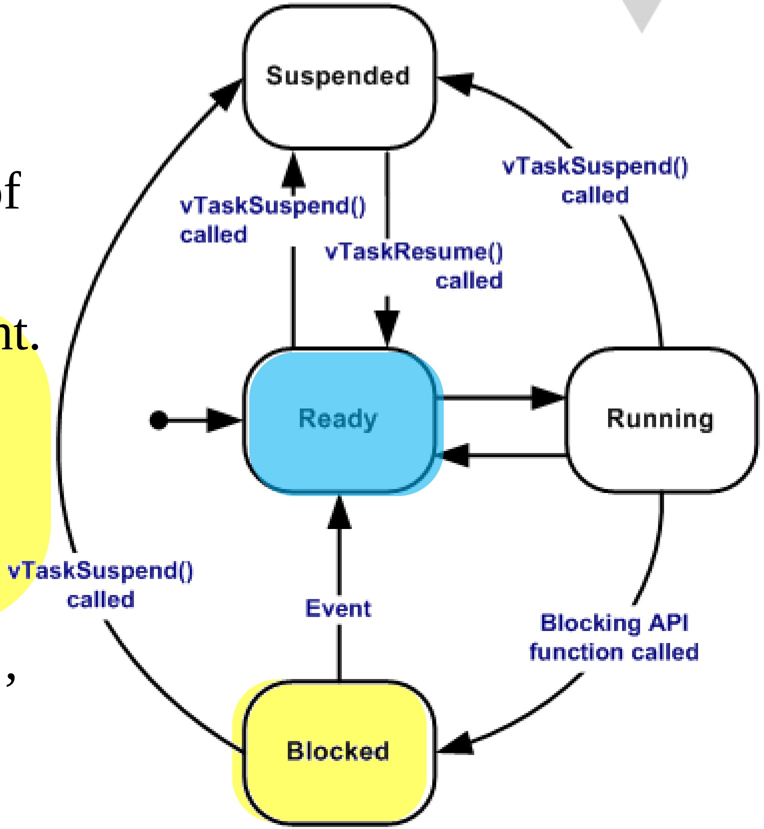
Task States

- **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.



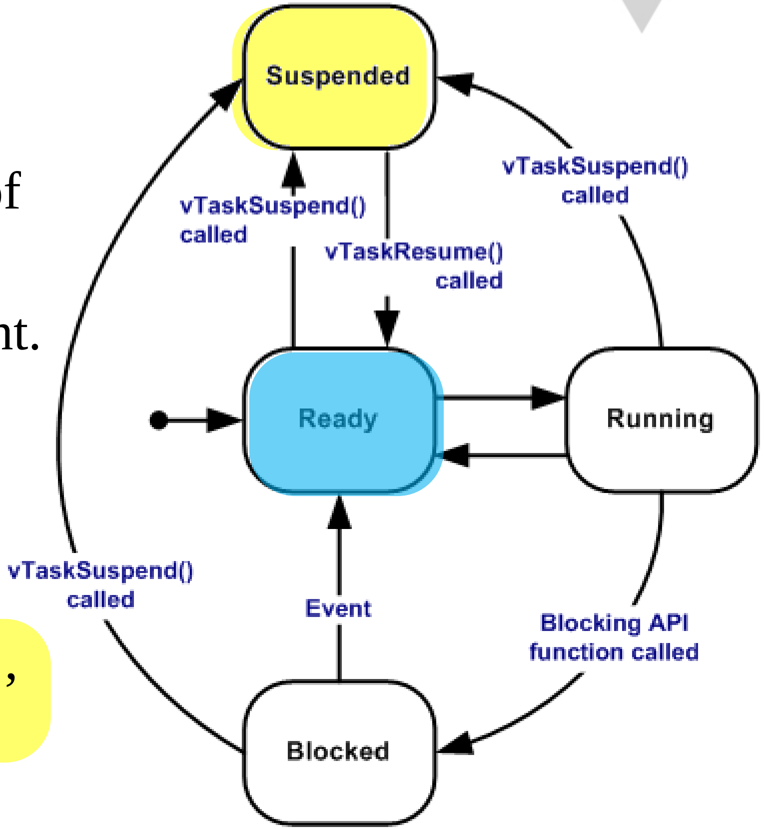
Task States

- **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.



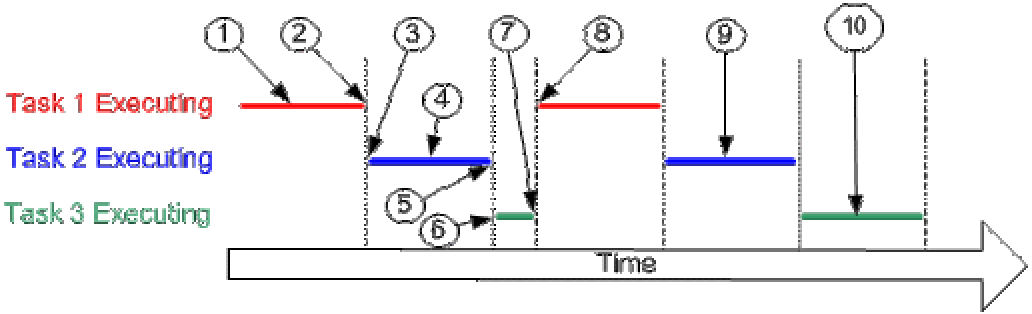
RTOS - Multitasking

- 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)

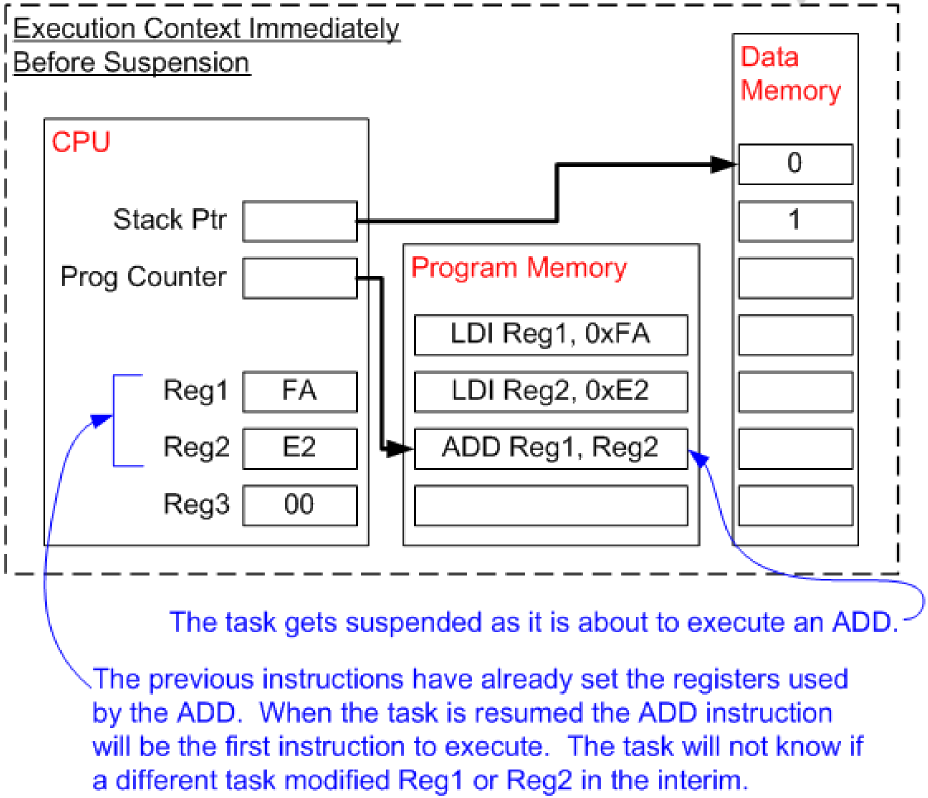
Example:
2: T1 preemption
4: T2 lock a resource R
7: try access resource R
9: release lock on R
10: lock resource R



RTOS – Context Switch

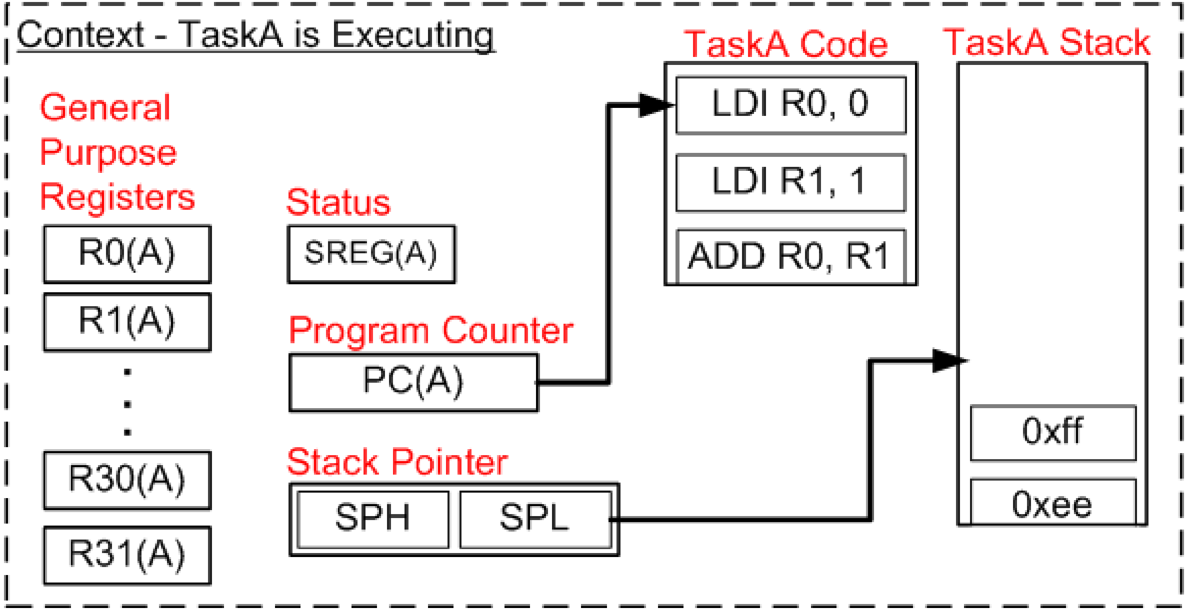
- Task execution requires **exclusive usage** of some computation resources
  - ✓ e.g. CPU registers and some RAM memory

- 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



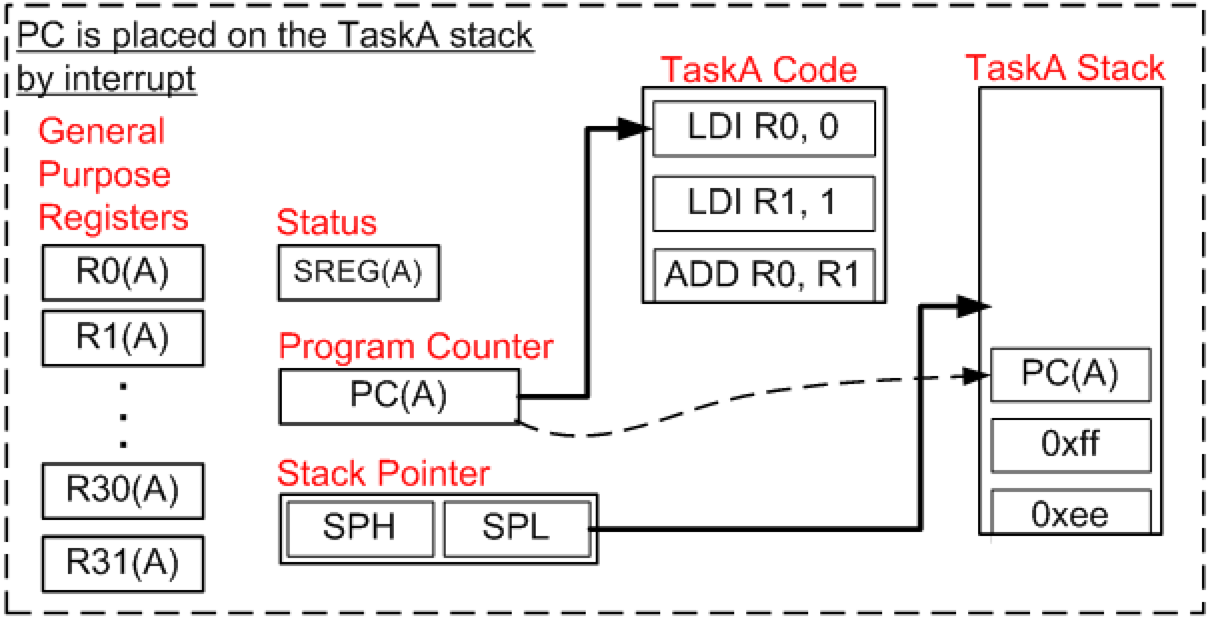
Context Switch in FreeRTOS (Step 1)

- **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 <u>TaskA</u> 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)

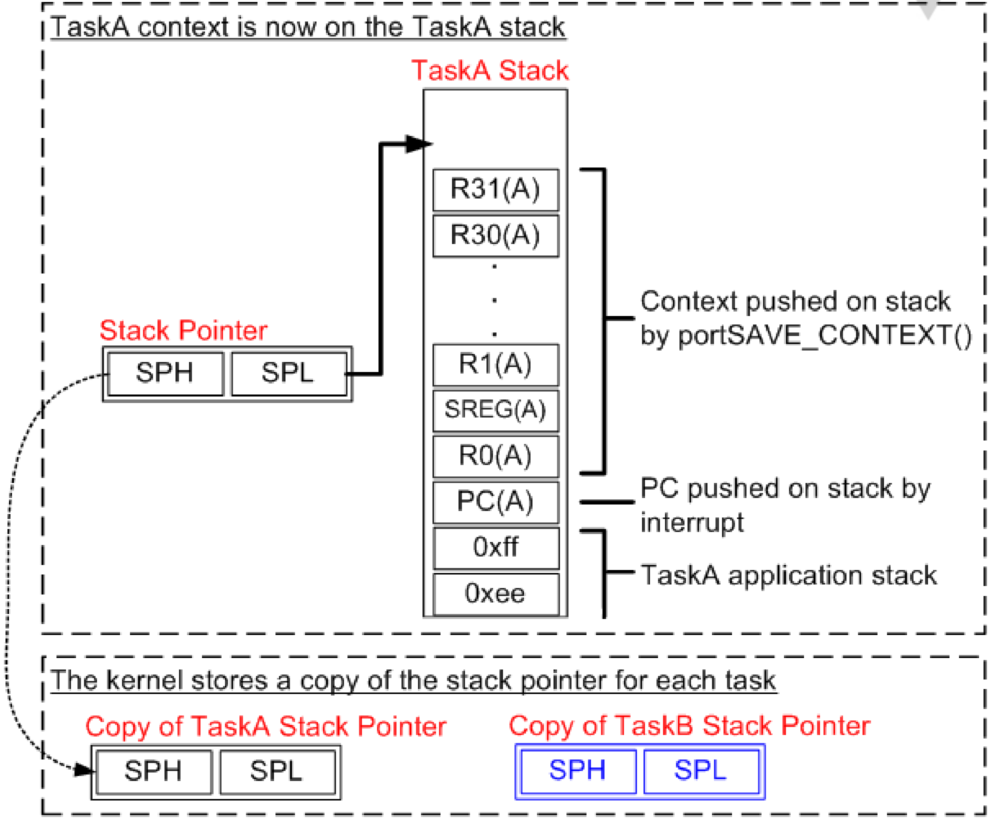
- **Status:** Jump to Interrupt service routine (ISR).
- **Now**: `portSAVE_CONTEXT()` pushes the entire AVR execution context onto the stack of TaskA.

```
/* 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.

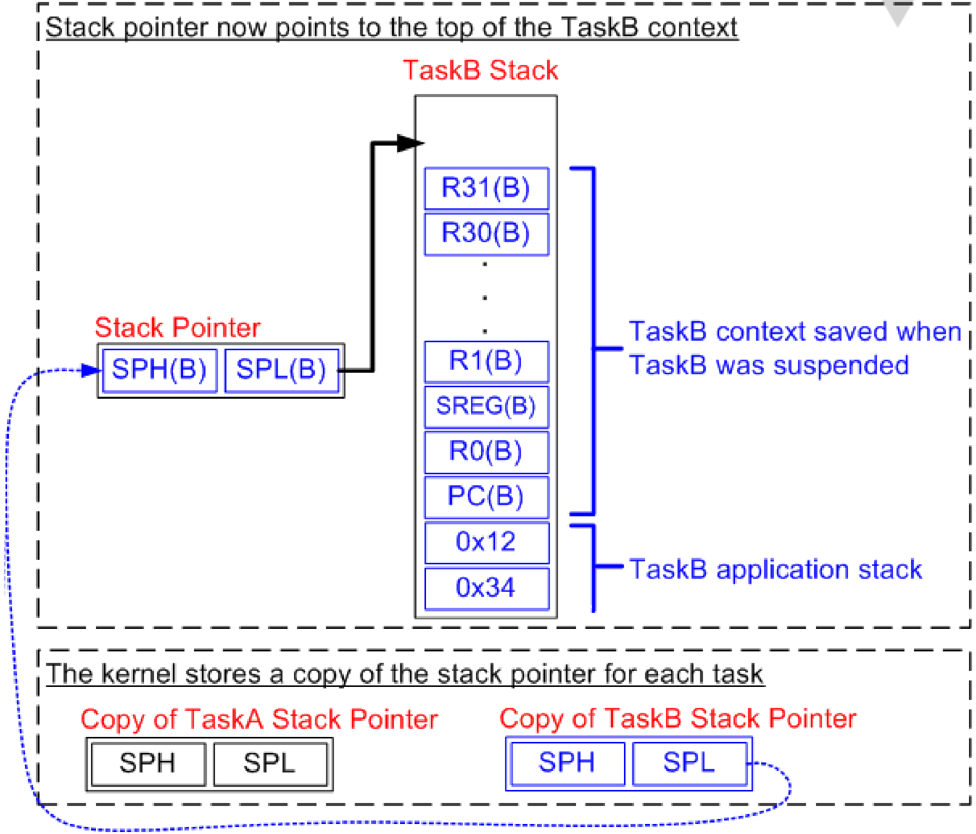
- Some assumptions:
  - ✓ Incrementing the tick count has caused TaskB to become **ready to run**
  - ✓ **TaskB** has a higher priority than **TaskA**

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

Context Switch in FreeRTOS (Step 5)

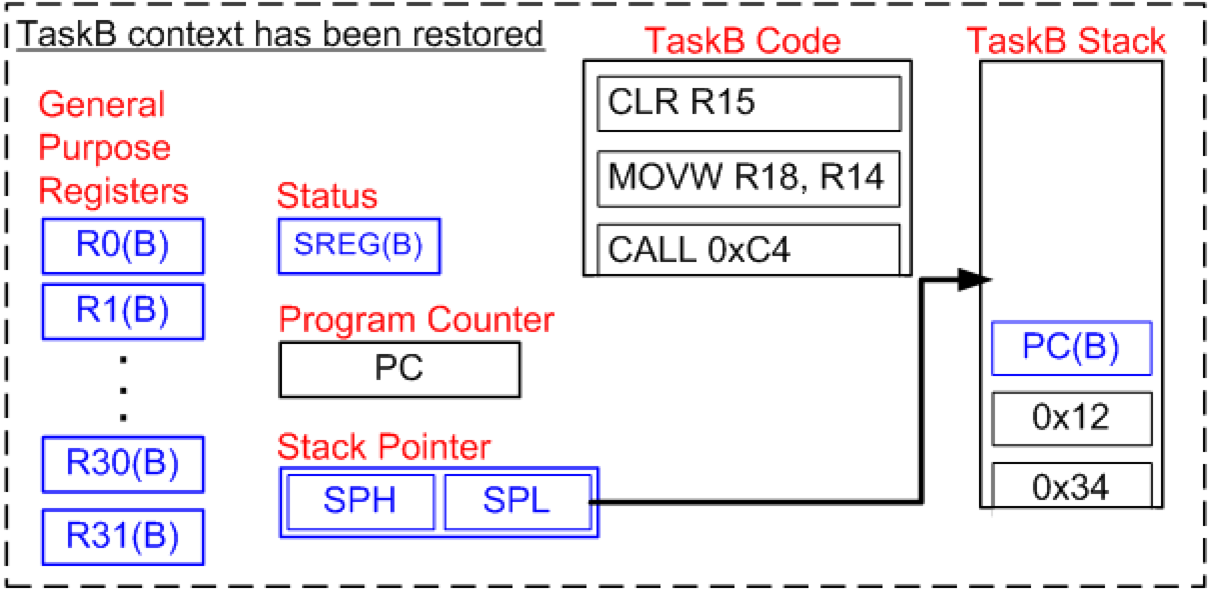
- **Status:** The TaskB context must be restored..

- **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)

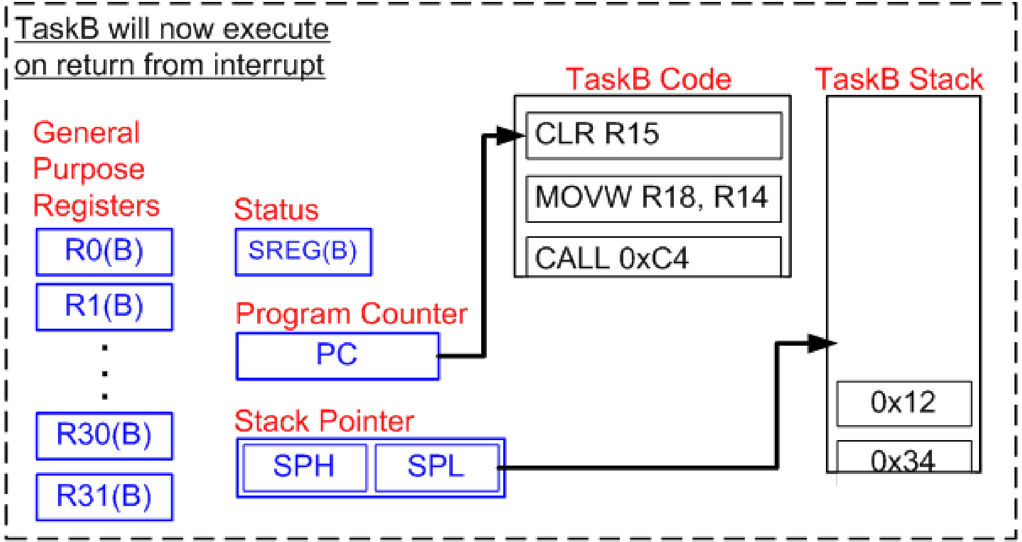
- **Status:** `portRESTORE_CONTEXT()` completes by restoring the TaskB context from its stack into the appropriate processor registers..
- **Now**: Only the program counter remains on the stack.



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

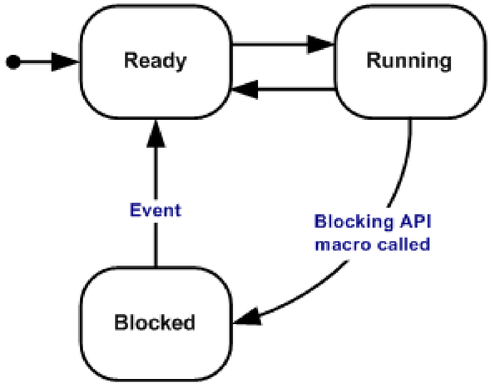
The Idle Task

- 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

RealTime Applications (1/2)

- RTOS have **specific objectives** w.r.t. other OS
  - ✓ These differences are reflected in the scheduling policy
- RTOS are designed to provide **timely responses**
  - ✓ To real world events
- Events occurring in the real world can have **deadlines**
  - ✓ Before which the real time embedded system **must respond**
  - ✓ the RTOS scheduling policy must <u>grant</u> these deadlines
- How to achieve such a behavior?
  - ✓ Software engineer must first <u>assign a priority to each task</u>
  - ✓ The RTOS scheduler must ensure that the **highest priority task**, which is ready to execute, is **the selected one**

RealTime Applications (2/2)

- Example: RT control system with keypad and LCD
  - R1: users expect **visual feedback** within a reasonable period
  - R2: a control function relies on a digitally filtered input
- Interface and Control Functions
  - Two user defined tasks

```
void vKeyHandlerTask( void *pvParameters )
{
    // Key handling is a continuous process and as such the task
    // is implemented using an infinite loop (as most real time
    // tasks are).
    for( ;; )
    {
        [Suspend waiting for a key press]

        [Process the key press]
    }
}
```

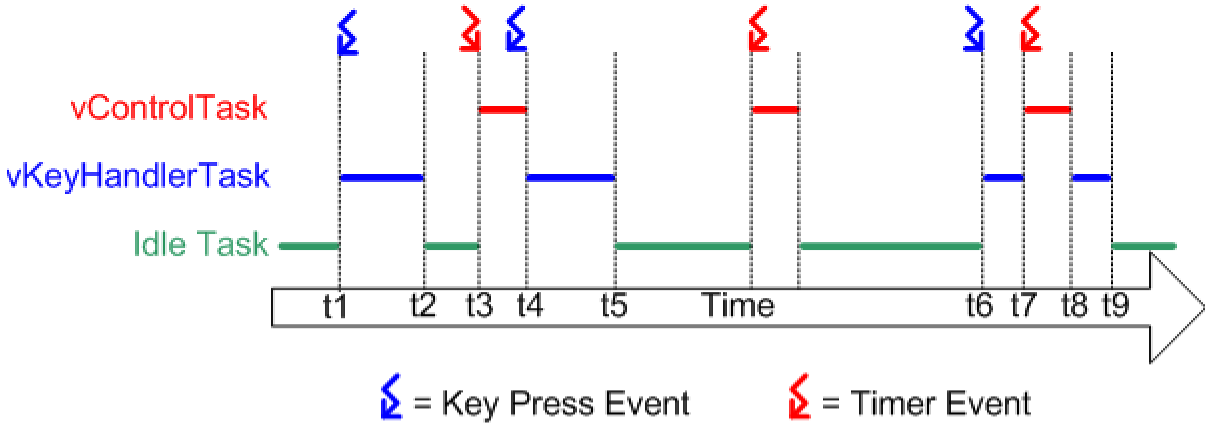
Deadline missing has less severe consequances

```
void vControlTask( void *pvParameters )
{
    for( ;; )
    {
        [Suspend waiting for 2ms since the start of the previous
        cycle]

        [Sample the input]
        [Filter the sampled input]
        [Perform control algorithm]
        [Output result]
    }
}
```

Stricter (critical) deadline

RealTime Scheduling

- An additional **idle task** is created by the RTOS
  - It is always **in a state** where it is able to execute
  - It will be executed only when there are **no other tasks** able to do so
- Task scheduling is **event-based**, considering priorities
  - <u>The higher priority task</u> which is ready to run gets the CPU



Building Blocks

- Overall view of mechanisms for task scheduling
  - ✓ Development Tools
  - ✓ The RTOS Tick
  - ✓ WinAVR Signal Attribute
  - ✓ WinAVR Naked Attribute
  - ✓ Example: The AVR Context
  - ✓ Saving the Context
  - ✓ Restoring the Context

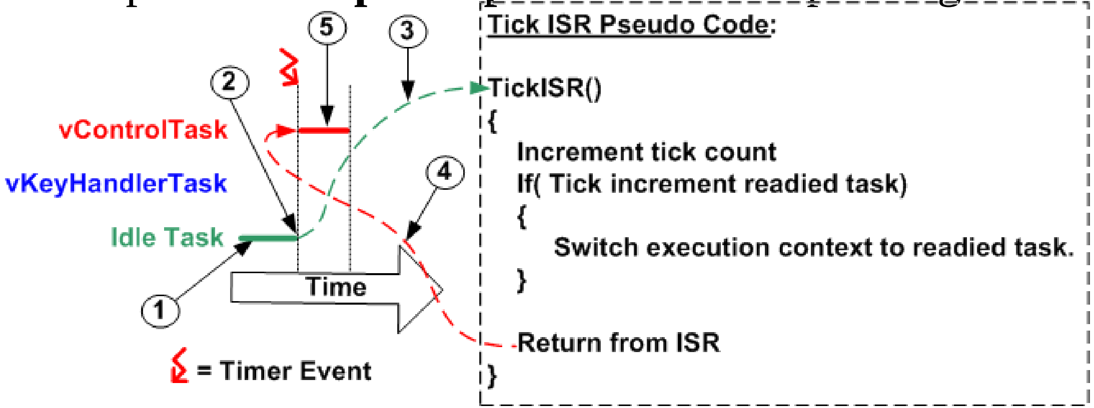
The Tick Timer (1/2)

- Provide support for **real time kernel** time measures
  - It is just a “counting variable”
  - A timer interrupt **increments** the <u>tick count</u>
  - Allowing the **real time kernel** to measure time
    - With a **resolution** of the chosen <u>*timer interrupt frequency*</u>

- Task could be suspended for **two main reasons**
  - Sleeping => specify time after which it requires 'waking‘
    - e.g. to introduce a certain delay between two actions
  - Blocking => specify a maximum time it wishes to wait
    - e.g. to wait for an event or a resource being available

The Tick Timer (2/2)

- When the tick **interrupt is triggered**, the kernel:
  - ✓ Increment the tick count
  - ✓ Check to see if it is now time to **unblock** or **wake** a task
  - ✓ Return to the newly **woken/unblocked** task
    - *Iff newly ready task has higher priority than the interrupted one*
- Preemptive context switch
  - ✓ The interrupted task is **preempted** without suspending itself voluntarily



GCC Signal Attribute

- GCC allows interrupts to be written in C
  - ✓ e.g. a compare match event on the AVR Timer 1 peripheral can be written using the following syntax

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
void SIG_OUTPUT_COMPARE1A( void )
{
    /* ISR C code for RTOS tick. */
    vPortYieldFromTick();
}
```

- Exploiting the '__attribute__ ((signal))' directive
  - ✓ Informs the compiler that the function <u>is an ISR</u>
  - ✓ Results in two important changes in the compiler output
    1. Ensures that every processor register that **gets modified** during the ISR is restored to its original value when the ISR exits
    2. Forces a '**return from interrupt**' instruction (**RETI**) to be used

GCC Naked Attribute

- Prevents GCC to generate **function entry** or **exit code**
  - ✓ Context switch requires the entire context to be saved
  - ✓ Switching code explicitly save **all CPU registers** at ISR entry
    - *This would result in some CPU registers **being saved twice***
- Two FreeRTSO macros save and restore the entire execution context

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );

void SIG_OUTPUT_COMPARE1A( void )
{
    /* Macro that explicitly saves the execution context. */
    portSAVE_CONTEXT();
    /* ISR C code for RTOS tick. */
    vPortYieldFromTick();
    /* Macro that explicitly restores the execution context. */
    portRESTORE_CONTEXT();
    /* The return from interrupt call must also be explicitly added. */
    asm volatile ( "reti" );
}
```

GCC Naked Attribute - Comparison

- With naked attribute

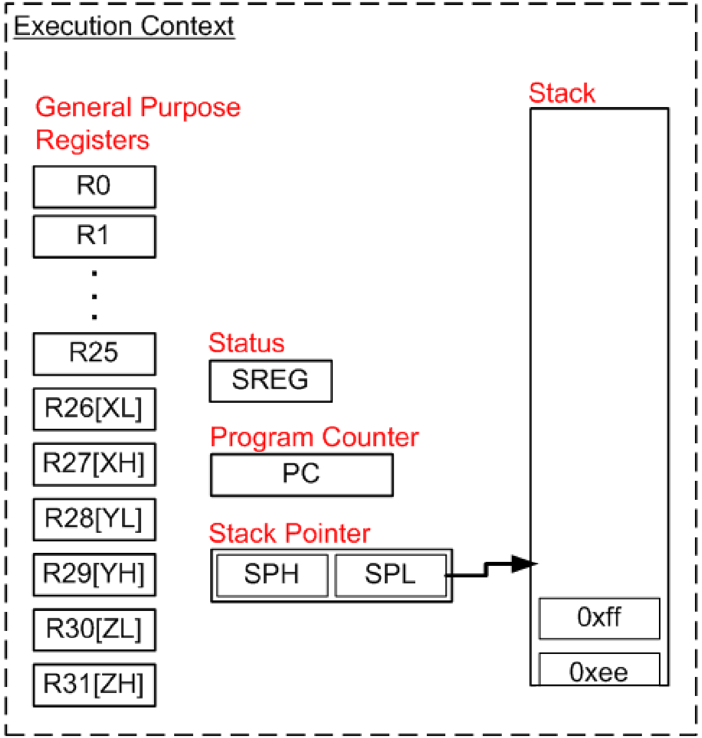
```
;void SIG_OUTPUT_COMPARE1A( void )
;{
    ; ---------------------------------------
    ; NO COMPILER GENERATED CODE HERE TO SAVE
    ; THE REGISTERS THAT GET ALTERED BY THE
    ; ISR.
    ; ---------------------------------------

    ; CODE GENERATED BY THE COMPILER FROM THE
    ; APPLICATION C CODE.

    ;vTaskIncrementTick();
    CALL    0x0000029B      ;Call subroutine

    ; ---------------------------------------

    ; NO COMPILER GENERATED CODE HERE TO RESTORE
    ; THE REGISTERS OR RETURN FROM THE ISR.
    ; ---------------------------------------
;}
```

- Without naked attribute

```
;void SIG_OUTPUT_COMPARE1A( void )
;{
    ; ---------------------------------------
    ; CODE GENERATED BY THE COMPILER TO SAVE
    ; THE REGISTERS THAT GET ALTERED BY THE
    ; APPLICATION CODE DURING THE ISR.

    PUSH    R1
    PUSH    R0
    IN      R0,0x3F
    PUSH    R0
    CLR     R1
    PUSH    R18
    PUSH    R19
    PUSH    R20
    PUSH    R21
    PUSH    R22
    PUSH    R23
    PUSH    R24
    PUSH    R25
    PUSH    R26
    PUSH    R27
    PUSH    R30
    PUSH    R31
    ; ---------------------------------------

    ; CODE GENERATED BY THE COMPILER FROM THE
    ; APPLICATION C CODE.

    ;vPortYieldFromTick();
    CALL    0x0000029B      ;Call subroutine
;}
    ; ---------------------------------------

    ; CODE GENERATED BY THE COMPILER TO
    ; RESTORE THE REGISTERS PREVIOUSLY
    ; SAVED.

    POP     R31
    POP     R30
```

The AVR Context (Example)

- On the AVR microcontroller the context consists of
  - ✓ 32 general purpose processor registers
    - The gcc development tools assume register R1 is set to zero
  - ✓ Status register, which value affects instruction execution
    - Thus it must be **preserved** across context switches
  - ✓ Program counter
    - Upon resumption, a task **must continue execution** from the instruction that was about to be executed immediately prior to its suspension
  - ✓ The two stack pointer registers



Saving the Context

- Each real time task has its **own stack memory area**
  - ✓ The context can be saved by **pushing CPU registers there**

- Saving the context is one place where **assembly code** is usually unavoidable
  - ✓ the portSAVE_CONTEXT() is implemented as a macro

```
#define portSAVE_CONTEXT()
asm volatile (
  "push   r0                     nt"   (1)
  "in     r0, __SREG__           nt"   (2)
  "cli                           nt"   (3)
  "push   r0                     nt"   (4)
  "push   r1                     nt"   (5)
  "clr    r1                     nt"   (6)
  "push   r2                     nt"   (7)
  "push   r3                     nt"
  "push   r4                     nt"
  "push   r5                     nt"

      :
      :
      :

  "push   r30                    nt"
  "push   r31                    nt"
  "lds    r26, pxCurrentTCB      nt"   (8)
  "lds    r27, pxCurrentTCB + 1  nt"   (9)
  "in     r0, __SP_L__           nt"   (10)
  "st     x+, r0                 nt"   (11)
  "in     r0, __SP_H__           nt"   (12)
  "st     x+, r0                 nt"   (13)
);
```

Restoring the Context

- The context is restored by “**reversing**” the saving code
  - ✓ The `portRESTORE_CONTEXT()` macro is the reverse of `portSAVE_CONTEXT()`
- The kernel
  - ✓ Retrieves the stack pointer for the task
  - ✓ POP's the context back into the correct CPU registers

```
#define portRESTORE_CONTEXT()
asm volatile (
  "lds   r26, pxCurrentTCB          nt"   (1)
  "lds   r27, pxCurrentTCB + 1      nt"   (2)
  "ld    r28, x+                    nt"
  "out   __SP_L__, r28              nt"   (3)
  "ld    r29, x+                    nt"
  "out   __SP_H__, r29              nt"   (4)
  "pop   r31                        nt"
  "pop   r30                        nt"

    :
    :
    :

  "pop   r1                         nt"
  "pop   r0                         nt"   (5)
  "out   __SREG__, r0               nt"   (6)
  "pop   r0                         nt"   (7)
);
```

Implementing a Task (1/2)

- A task should
  - ✓ Have a function prototype of type `pdTASK_CODE`
  - ✓ never return
    - Typically implemented as a continuous loop

```
void vATaskFunction( void *pvParameters ) {
        For( ;; ) {
            // Task application code here
        }
    }
```

- Tasks are created by calling `xTaskCreate()` and deleted by calling `vTaskDelete()`

Implementing a Task (2/2)

- Task functions can be defined **using macros**
  - ✓ Allow compiler **specific syntax** to be added
    - prototype definition: `portTASK_FUNCTION_PROTO`
    - function definition: `portTASK_FUNCTION`

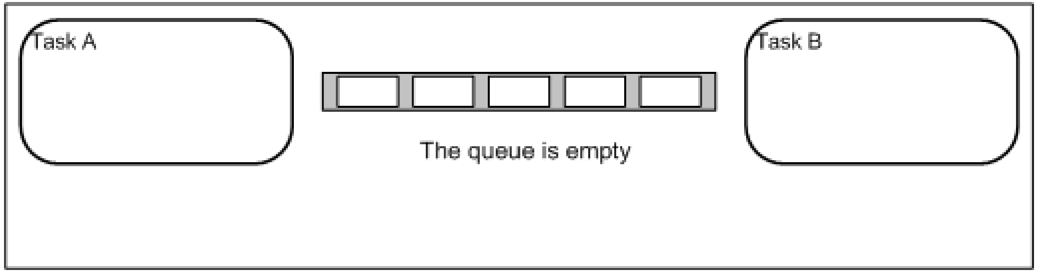
```
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
portTASK_FUNCTION( vATaskFunction, pvParameters ) {
        For( ;; ) {
            // Task application code here
        }
    }
```

Queues (1/2)

- **Primary form** of inter-task communication
  - ✓ Send messages **between tasks**, and **between ISR and tasks**
  - ✓ Takes care of **all mutual exclusion** issues for you
- Mainly used as thread **safe FIFO buffers**
  - ✓ Data being sent to the back of the queue
  - ✓ Data can also be sent to the front
- Can contain 'items' of **fixes size**
  - ✓ Item size and max number of items defined at creation time
- Items are placed into a queue **by copy**
  - ✓ Simplifies your application design
  - ✓ Keep item size to a minimum

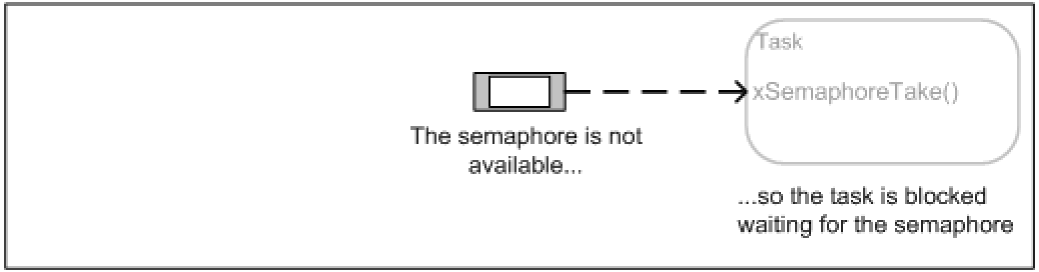
Queues (2/2)

- Allows a blocking time to be specified
  - Maximum number of '**ticks**' a task should be in **Blocked state**
    - Reading => wait for data to become available on a queue
    - Writing => wait for space to become available on a queue
  - Task with the highest priority will be unblocked first



Binary Semaphores

- Better choice for implementing synchronization
  - ✓ **Does not** support priority inheritance
  - ✓ **Looks like** a queue that can only hold a single item
  - ✓ **Support** for blocking timeout
- Example: use to synchronize a task with an interrupt
  - ✓ The interrupt only ever '**gives**' the semaphore
  - ✓ While the task only ever '**takes**' the semaphore

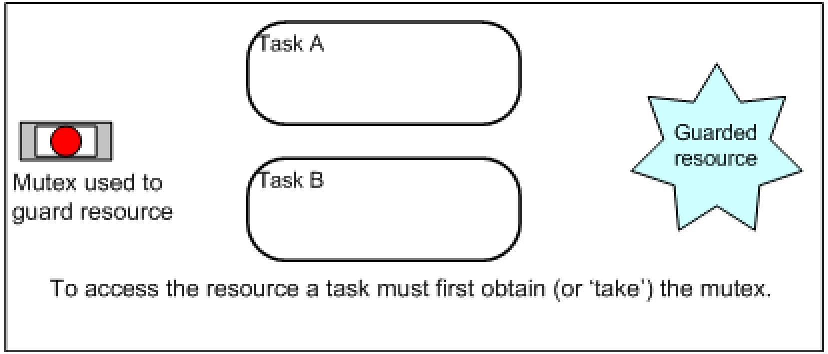


Counting Semaphores

- Looks like a queues of length greater than one
  - Users not interested in the data that is stored in the queue
  - Just whether or not the queue is **empty** or **not**
- Typically used for two things
  - <u>Counting events</u> difference between occurred and processed events
    - **Event handler** will 'give' a semaphore each time an event occurs
    - **Handler task** will 'take' a semaphore each time it processes an event
  - <u>Resource management</u> count of available resources
    - Tasks must first obtain a semaphore before using a resource

Mutex (1/2)

- Binary semaphores with priority inheritance support
  - ✓ Better for simple mutual exclusion
- Acts like **a token** that is used to guard a resource
  - ✓ Task wishing to access a resource must first obtain ('take') it when it has finished it must 'give' the token back
- Same semaphore access API functions
  - ✓ Support for block timeout



Mutex (2/2)

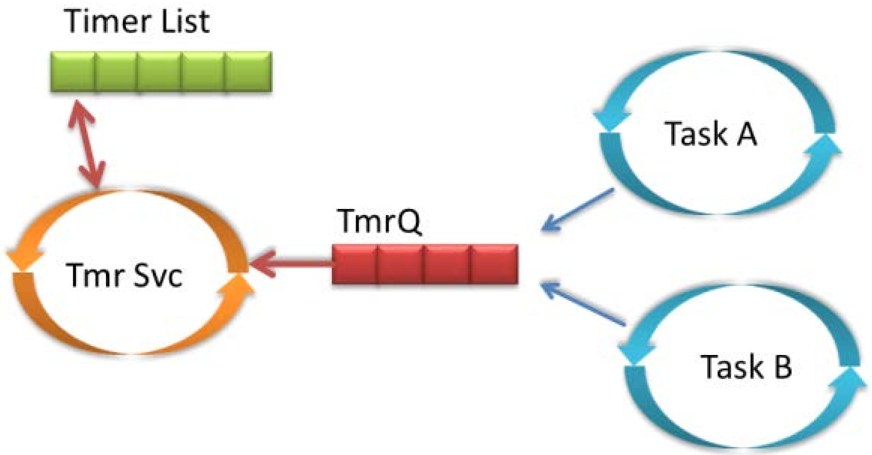
- Priority inheritance
  - If an high priority task blocks while getting a mutex
    - Priority of holding task temporarily raised to that of the blocked task
  - Ensure high priority task are blocked for the shortest time
  - Does not cure priority inversion
    - It just minimizes its effect in some situations
  - **Hard real time applications** should be designed such that priority inversion does not happen in the first place

Recursive Mutex

- Can be 'taken' repeatedly by the owner
- Doesn't become available again until the owner has **completely release** it
- For example
    - if a task successfully ‘takes’ the same mutex 5 times then the mutex will not be available to any other task until it has also ‘given’ the mutex back exactly five times.

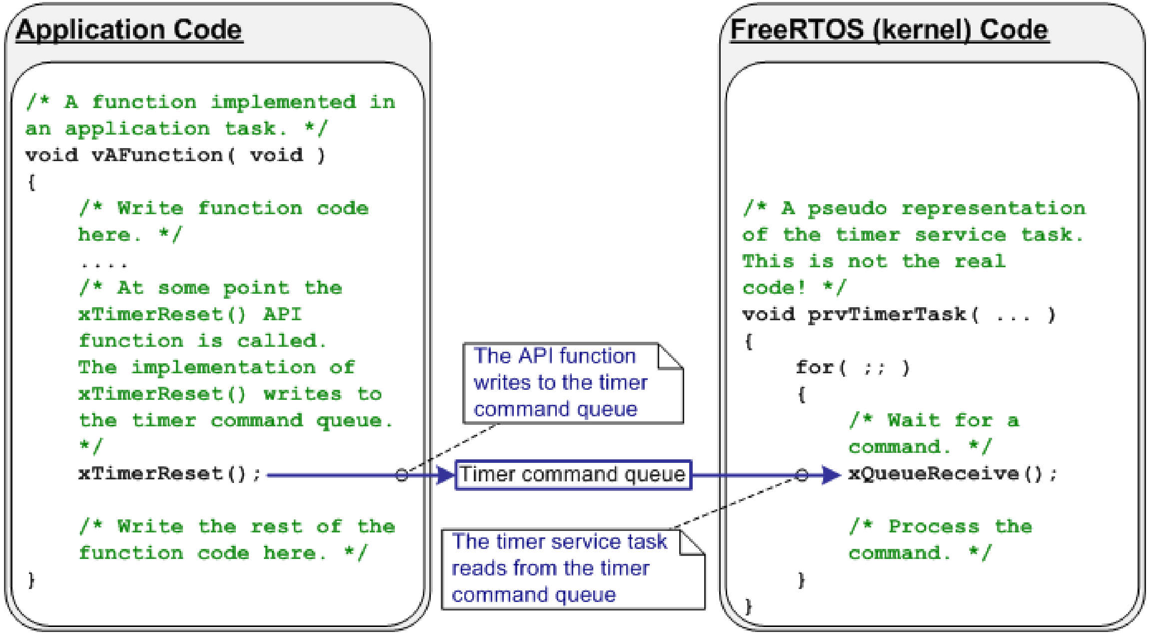
Software Timers

- Allows a **callback function** to be executed at a future timer period
  - ✓ Must be explicitly created before it can be used
  - ✓ **Optional** FreeRTOS functionality
    - Not part of the core FreeRTOS kernel
    - Provided by a timer service (or daemon) task
  - ✓ FreeRTOS provides a set of timer related API



- xTimerCreate
- xTimerCreateStatic
- xTimerIsTimerActive
- pvTimerGetTimerID
- pcTimerGetName
- vTimerSetReloadMode
- xTimerStart
- xTimerStop
- xTimerChangePeriod
- xTimerDelete
- xTimerReset
- xTimerStartFromISR
- xTimerStopFromISR
- xTimerChangePeriodFromISR
- xTimerResetFromISR
- pvTimerGetTimerID
- vTimerSetTimerID
- xTimerGetTimerDaemonTaskHandle
- xTimerPendFunctionCall
- xTimerPendFunctionCallFromISR
- pcTimerGetName
- xTimerGetPeriod
- xTimerGetExpiryTime

Software Timers - Example

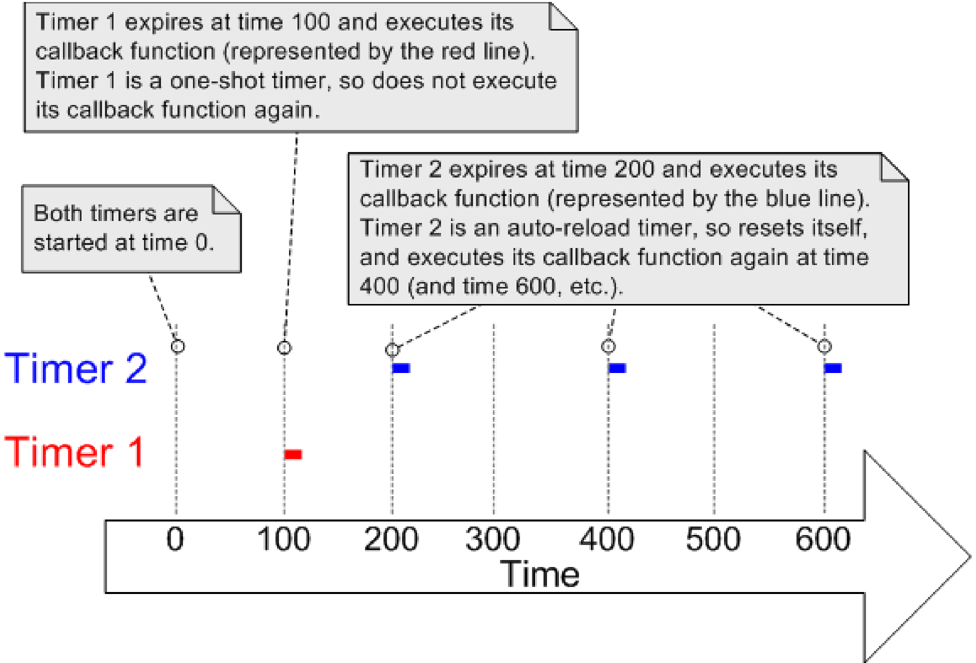


Software Timers - Usage

- Add `Source/timers.c` source file to your project
- Configure proper defines into `FreeRTOSConfig.h`
  - configUSE_TIMERS
  - configTIMER_TASK_PRIORITY
  - configTIMER_QUEUE_LENGTH
  - configTIMER_TASK_STACK_DEPTH
- Write a timer **callback function**
  - Executes in the context of the timer service task
  - Essential that timer callback functions **never** attempt to block

Software Timers - Types

- One-shot timer execute its callback function only once
    - ✓ Can be **manually** re-started, will **not automatically** re-start
- Auto-reload timer **automatically** re-start itself after each execution of its callback function



Software Timers - Resetting

- ➢ Recalculate its expiry time
  - ✓ Expiry time becomes relative to when the timer was reset
- ➢ Example: LCD backlight control