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

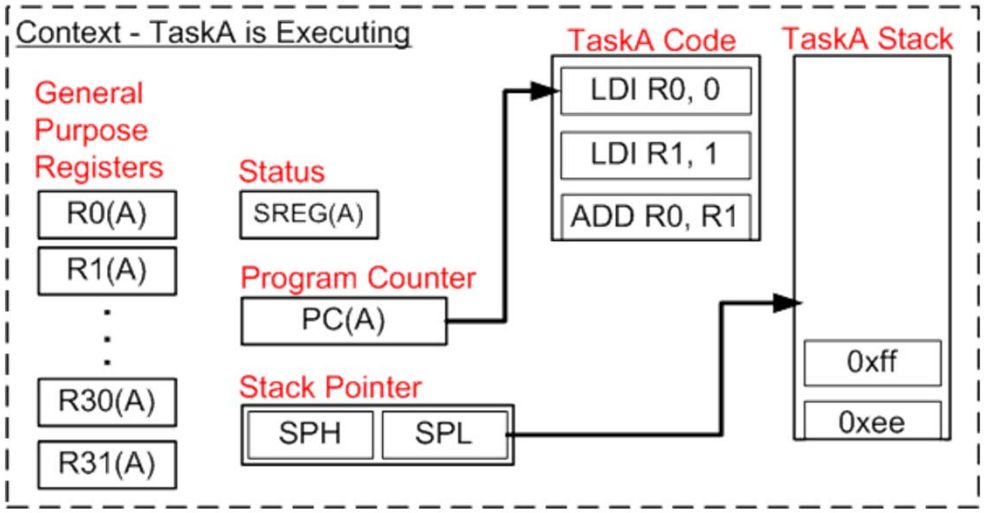
```
87  #ifndef configUSE_PREEMPTION
88          #error Missing definition:  configUSE_PREEMPTION should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configura
89  #endif
90
91  #ifndef configUSE_IDLE_HOOK
92          #error Missing definition:  configUSE_IDLE_HOOK should be defined in FreeRTOSConfig.h 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
00          #error  Missing definition:  configUSE_CO_ROUTINES should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Configu
01  #endif
02
03  #ifndef INCLUDE_vTaskPrioritySet
04          #error Missing definition:  INCLUDE_vTaskPrioritySet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Confi
05  #endif
06
07  #ifndef INCLUDE_uxTaskPriorityGet
08          #error Missing definition:  INCLUDE_uxTaskPriorityGet should be defined in FreeRTOSConfig.h as either 1 or 0.  See the Conf
09  #endif
10
11  #ifndef INCLUDE_vTaskDelete
12          #error Missing definition:  INCLUDE_vTaskDelete         should be defined in FreeRTOSConfig.h as either 1 or 0.  See the C
13  #endif
```

RTOS Fundamentals

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

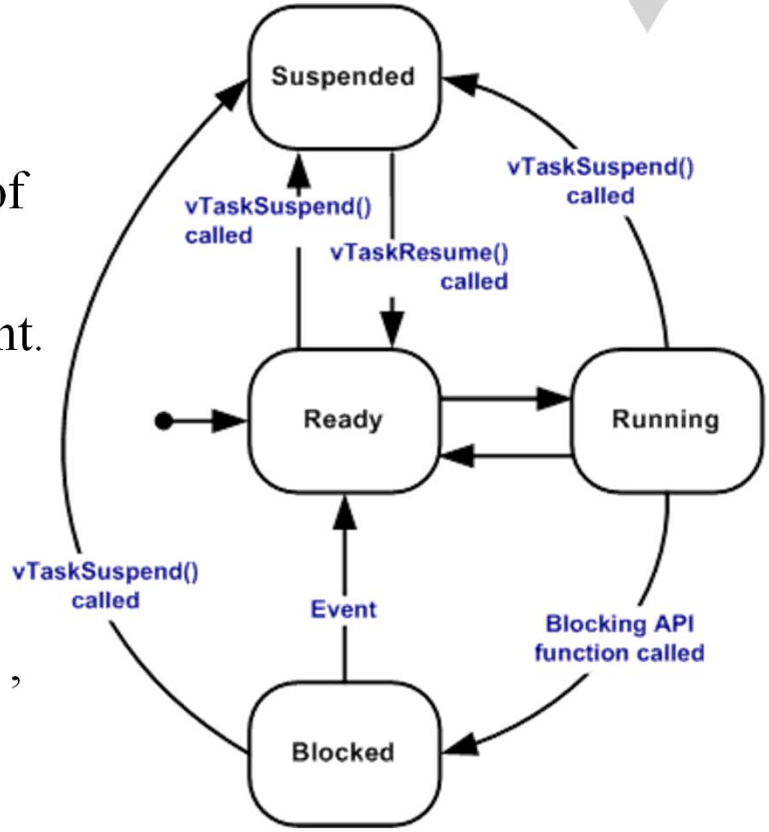
Tasks in FreeRTOS

- 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



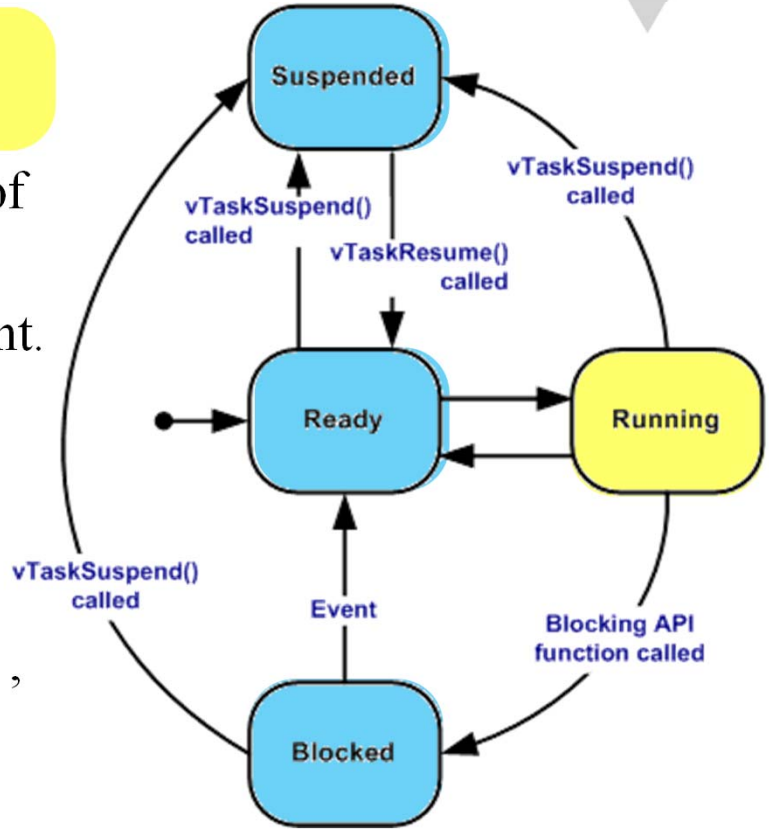
Task States

- **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.



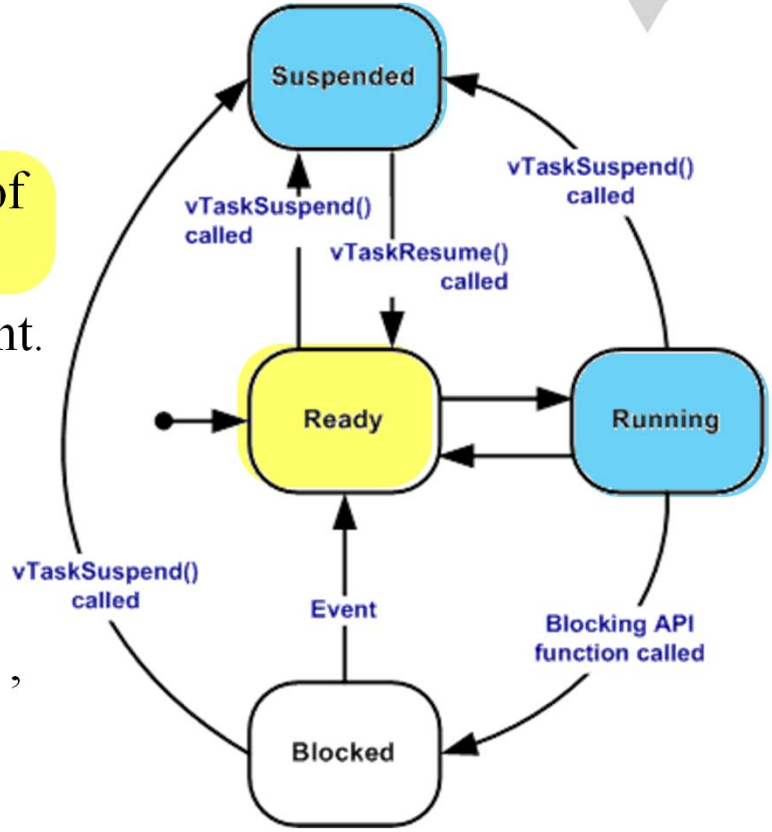
Task States

- **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.



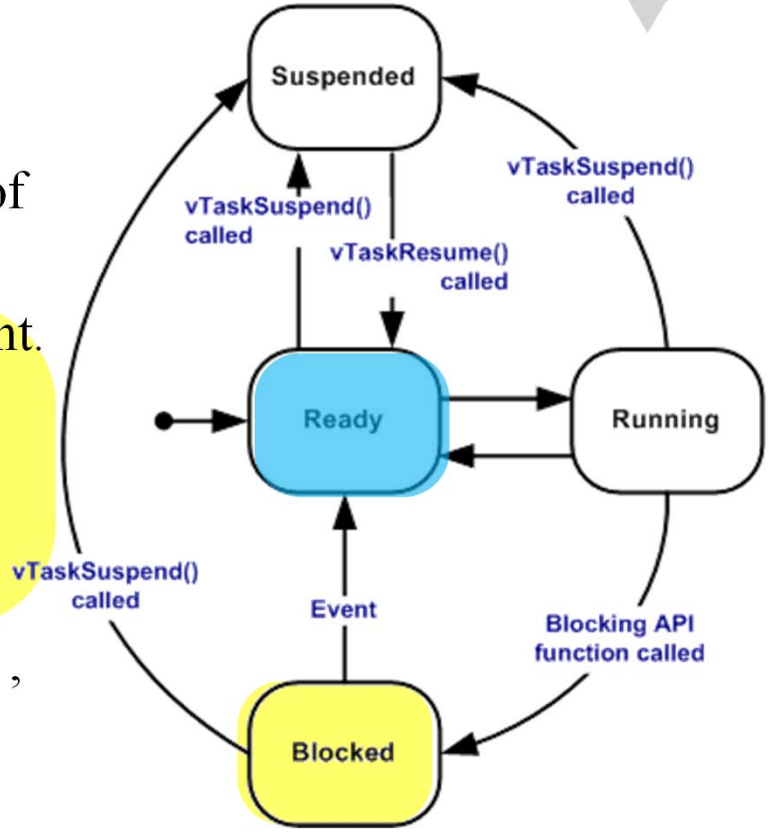
Task States

- **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.



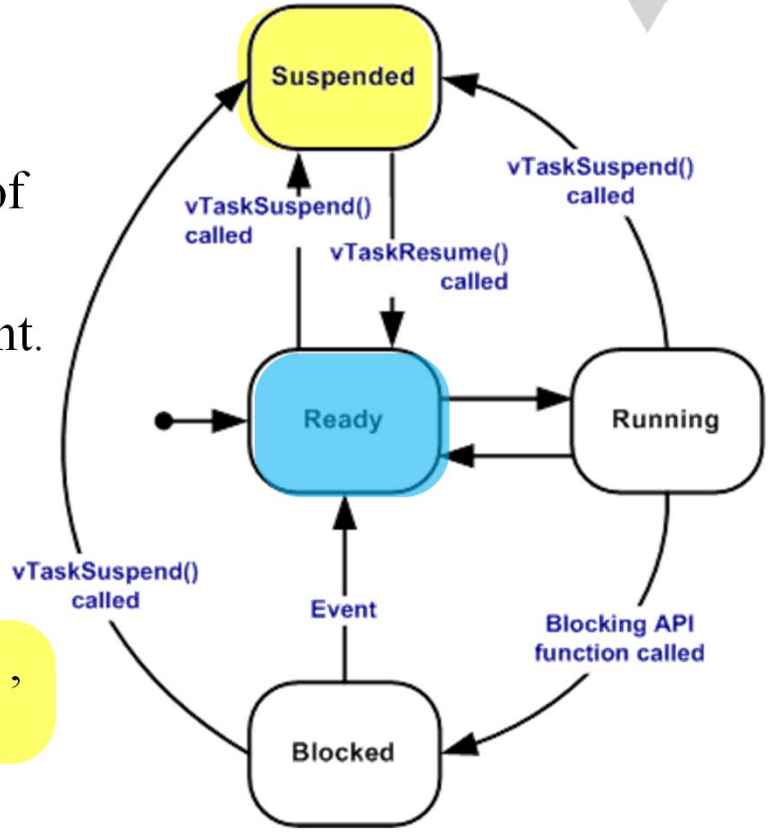
Task States

- **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.



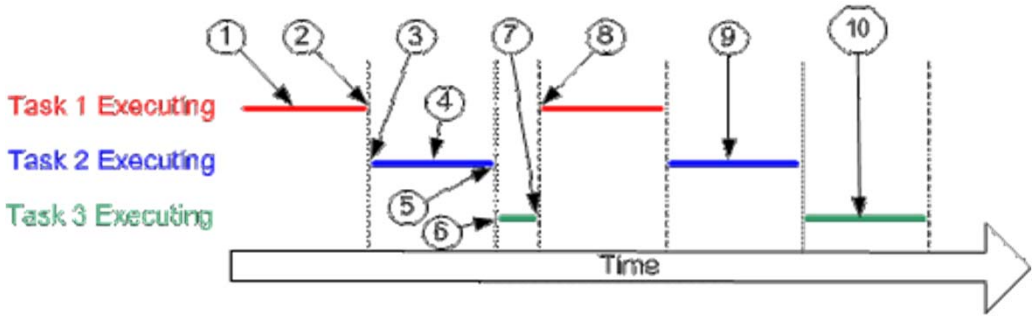
RTOS - Multitasking

- 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

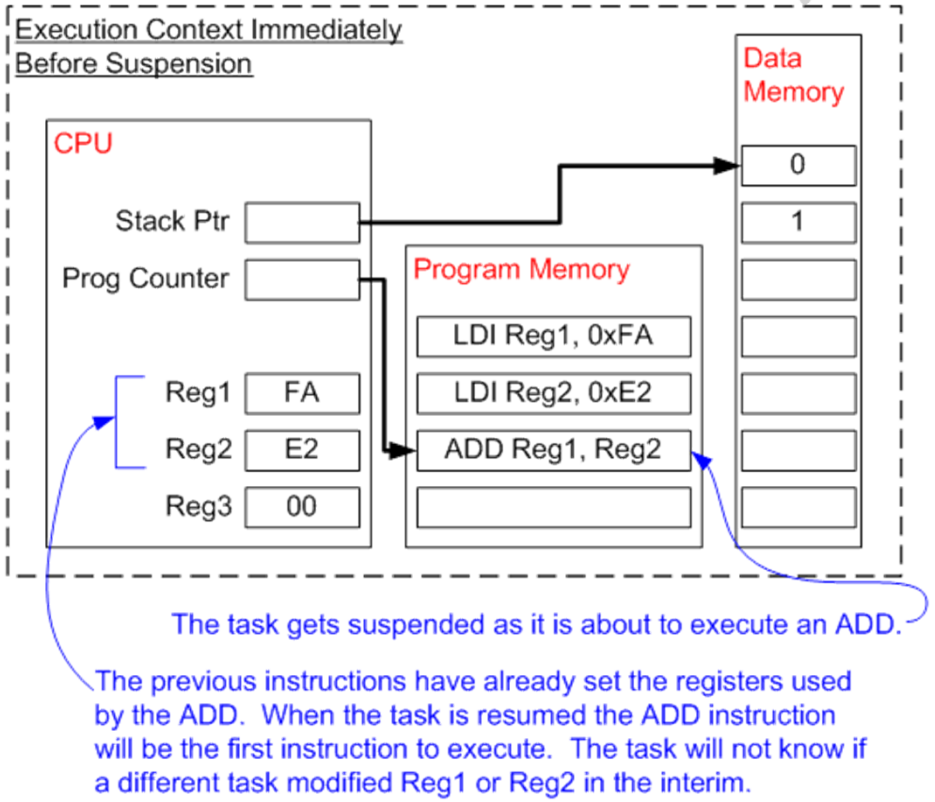
- Algorithm used to **decide the task to execute**
  - ✓ Tasks can be **suspend/resume** many times during its lifetime
  - ✓ "**Fair**" proportions of CPU time are allocated to tasks
  - ✓ A task can also suspend itself (e.g., **delay** (**sleep**) for a fixed period, or **wait** (**block**) for a resource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



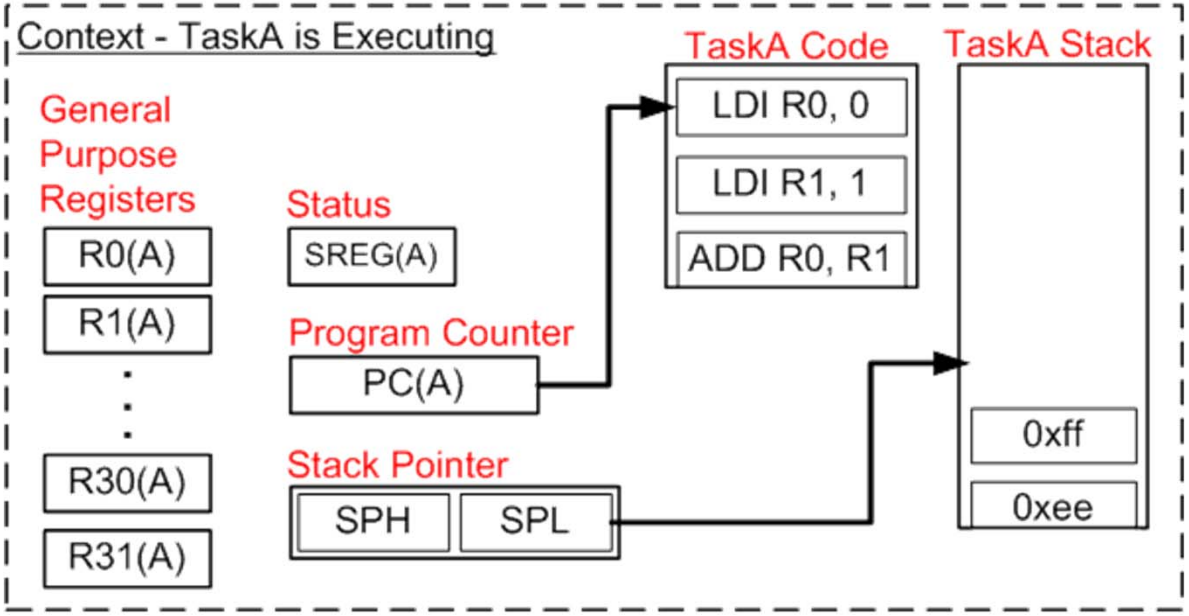
RTOS – Context Switch

- 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



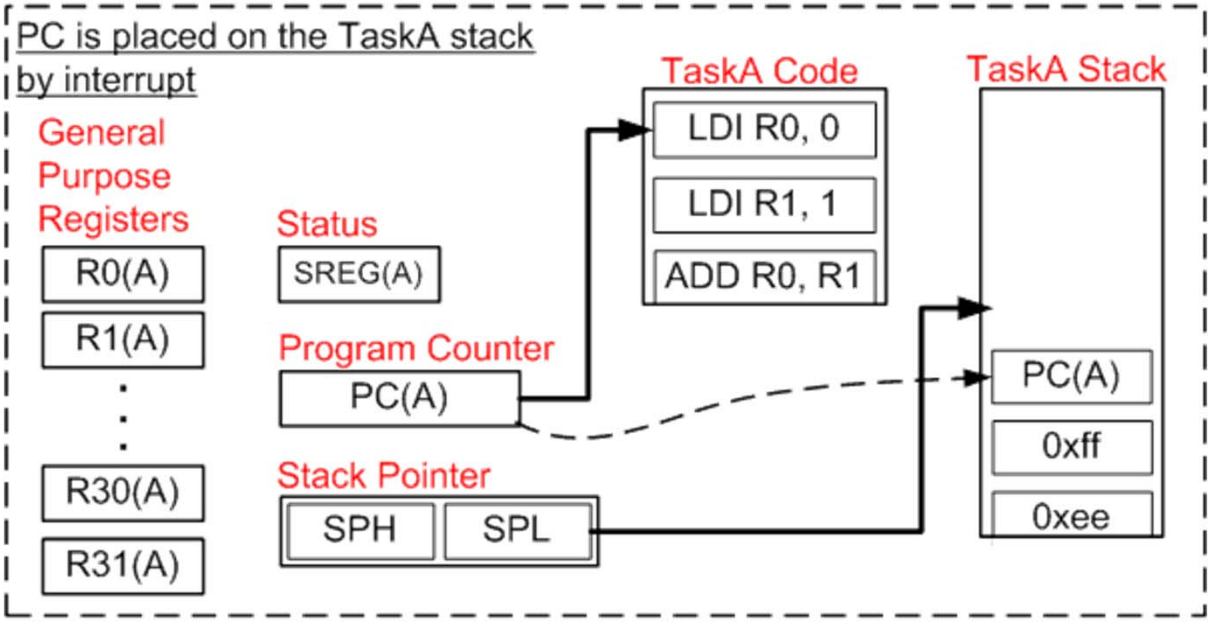
Context Switch in FreeRTOS (Step 1)

- **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)

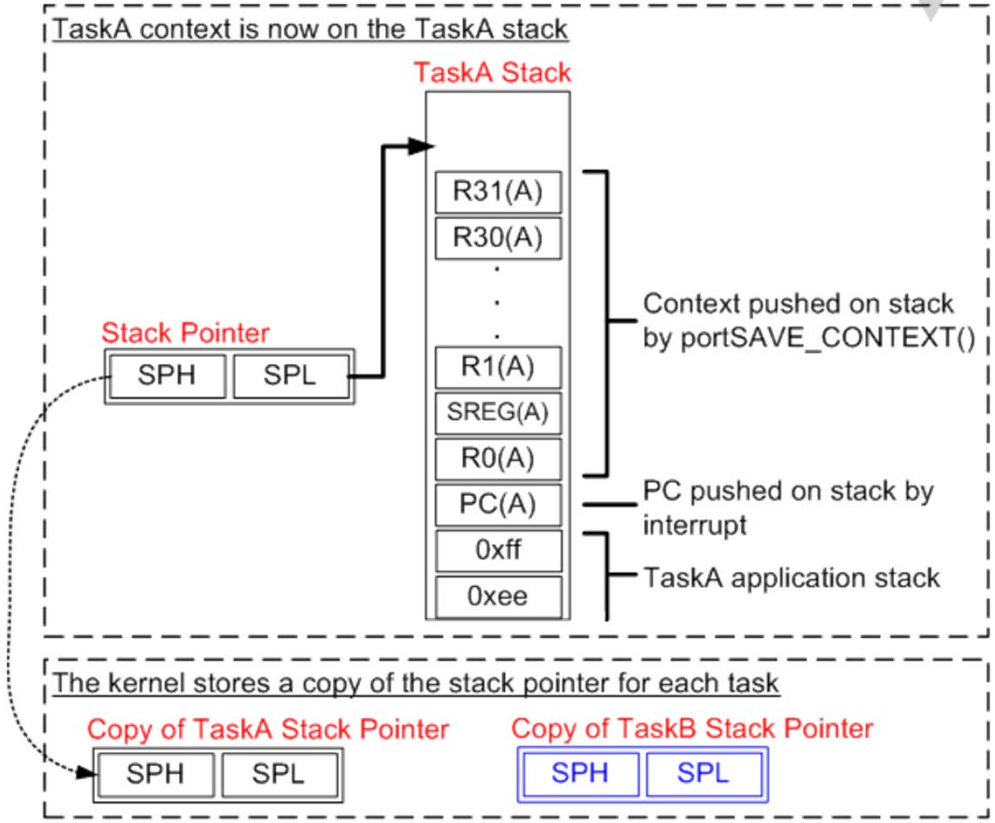
- **Status:** Jump to Interrupt service routine (ISR).
- **Now**: `portSAVE_CONTEXT()` pushes the entire AVR execution context onto the stack of TaskA.

```
/* 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.

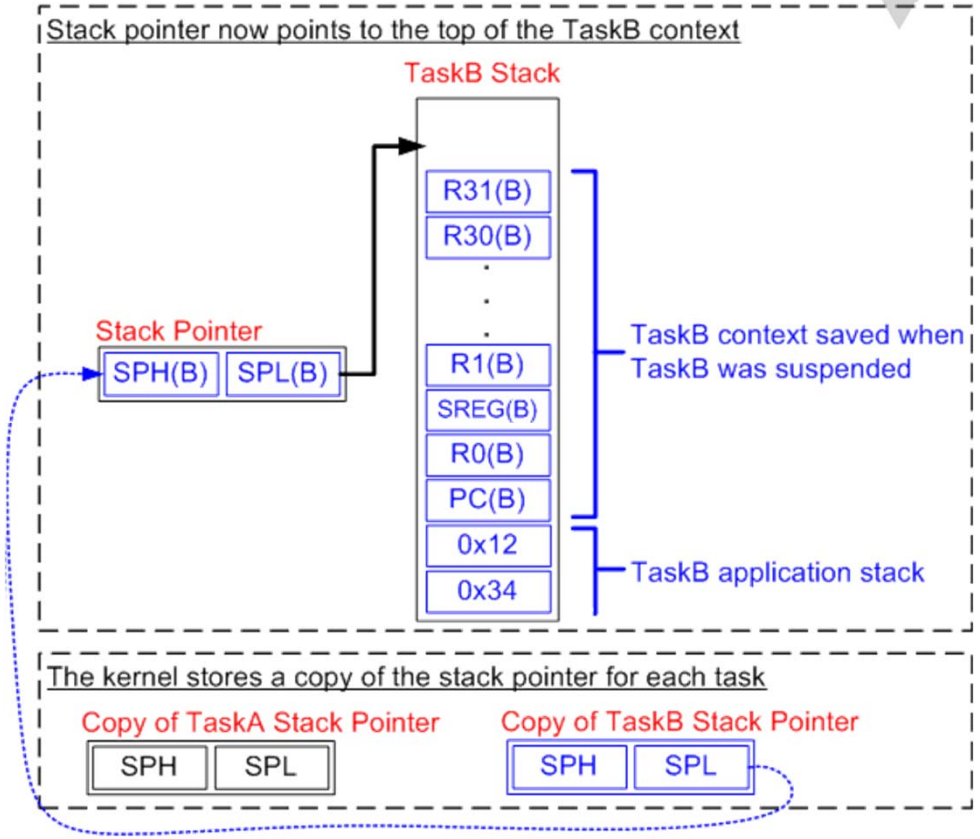
- Some assumptions:
  - ✓ Incrementing the tick count has caused TaskB to become **ready to run**
  - ✓ **TaskB** has a higher priority than **TaskA**

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

Context Switch in FreeRTOS (Step 5)

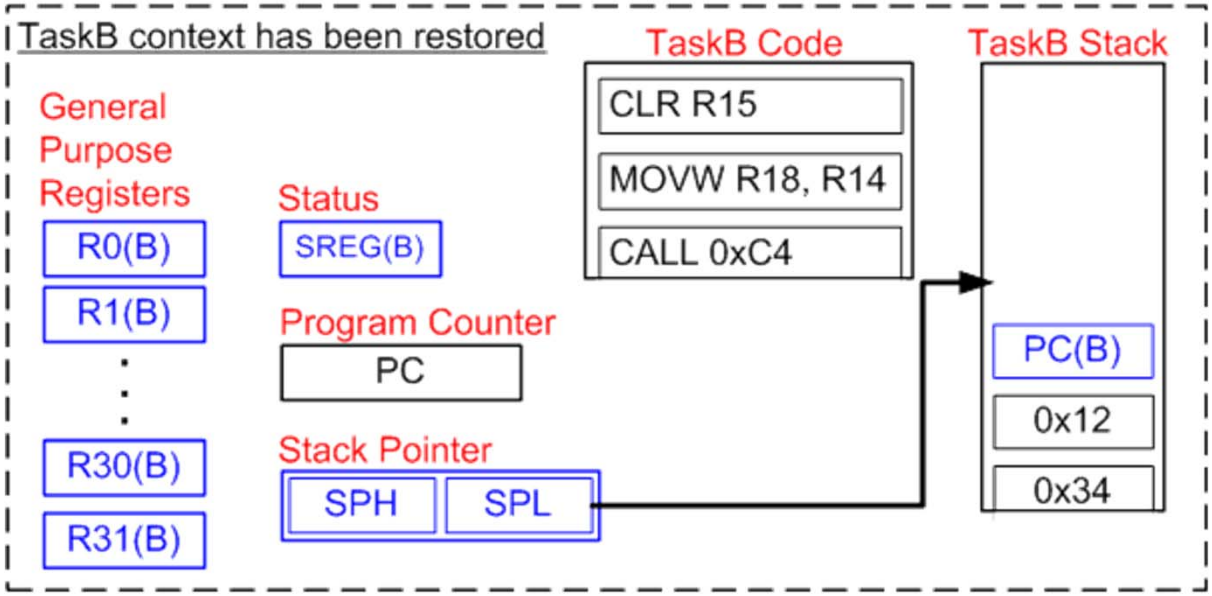
- **Status:** The TaskB context must be restored..

- **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)

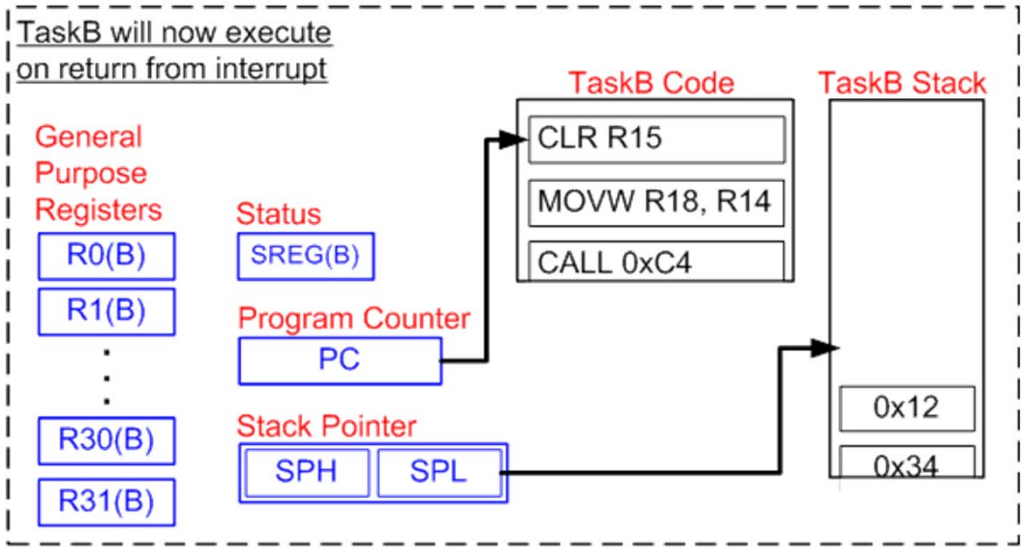
- **Status:** `portRESTORE_CONTEXT()` completes by restoring the TaskB context from its stack into the appropriate processor registers..
- **Now**: Only the program counter remains on the stack.



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

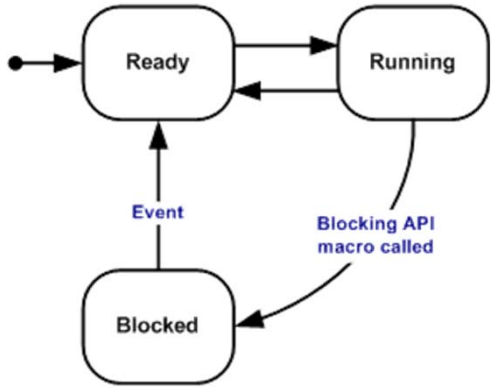
The Idle Task

- 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

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 grant these deadlines
- How to achieve such a behavior?
  - ✓ Software engineer must first assign a priority to each task
  - ✓ 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]
    }
}
```

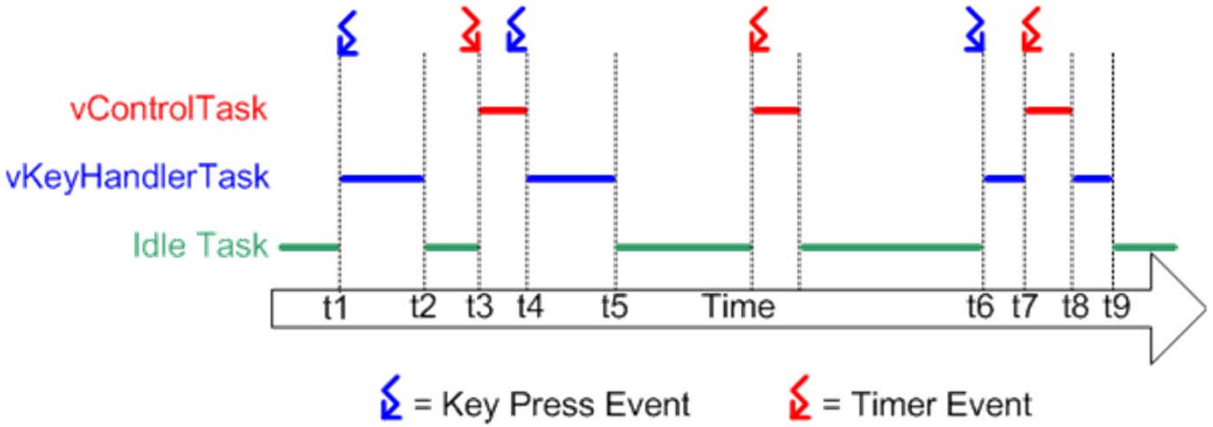
Deadline missing has less severe consequances

Stricter (critical) deadline

```
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]
    }
}
```

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
  - ✓ The higher priority task 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

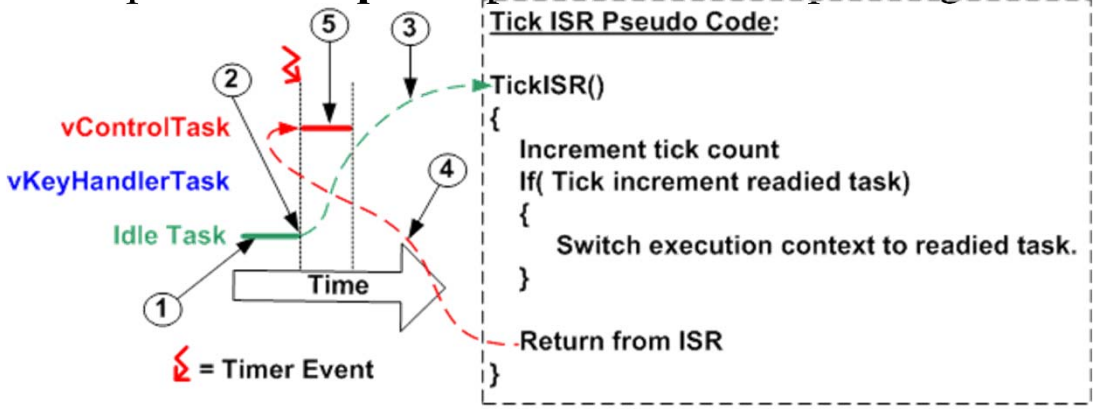
The Tick Timer (1/2)

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

- 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 is an ISR
  - ✓ 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

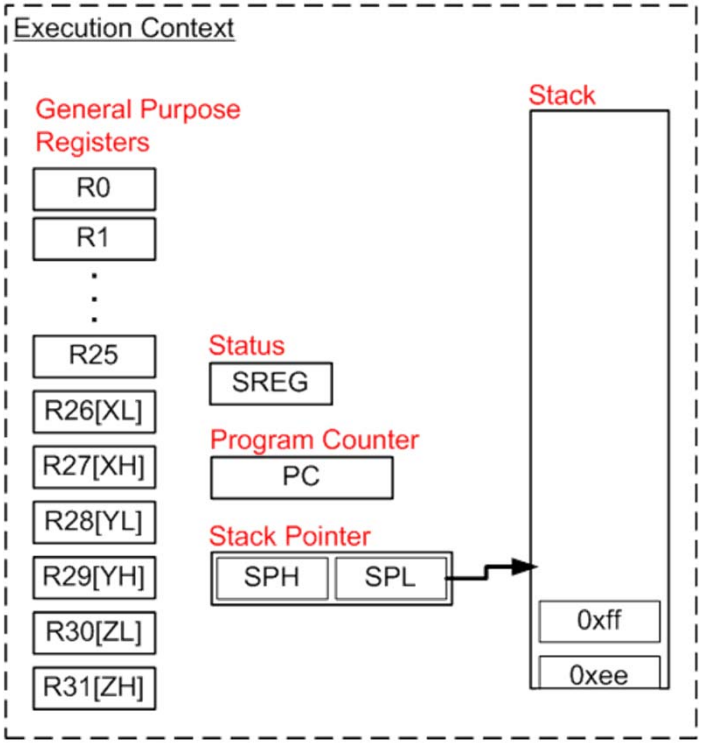
```
;void SIG_OUTPUT_COMPARE1A( void )
;{
    ; --------------------------------------
    ; NO COMPILER GENERATED CODE HERE TO SAVE
    ; THE REGISTERS THAT GET ALTERED BY THE
    ; ISR.
    ; --------------------------------------

    ; 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`

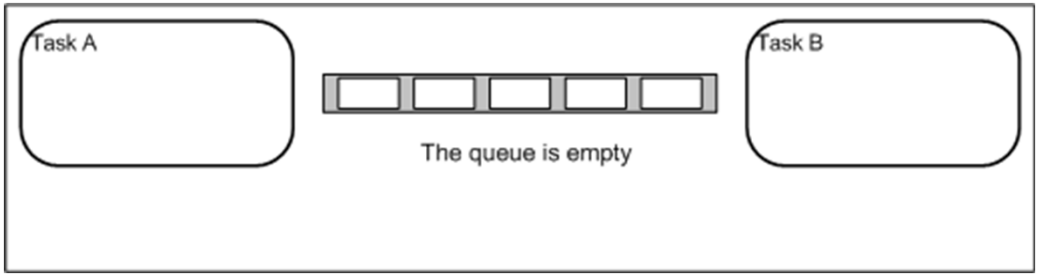
```
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
portTASK_FUNCTION( vATaskFunction, pvParameters ) {
        For( ;; ) {
            // Task application code here
        }
    }
```

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

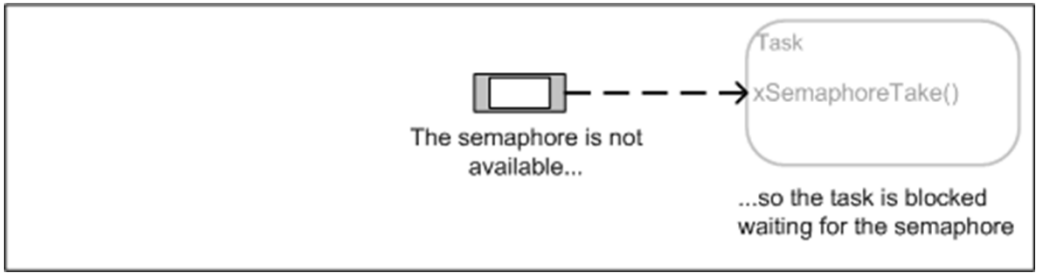
Queues (2/2)

- 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

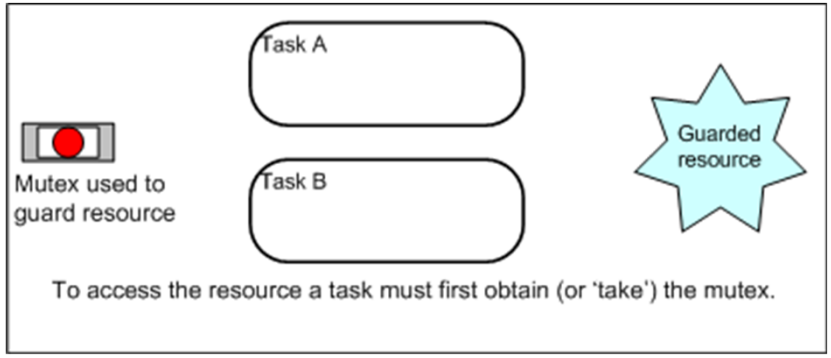


Counting Semaphores

- 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
  - ✓ Counting events 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
  - ✓ Resource management 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)

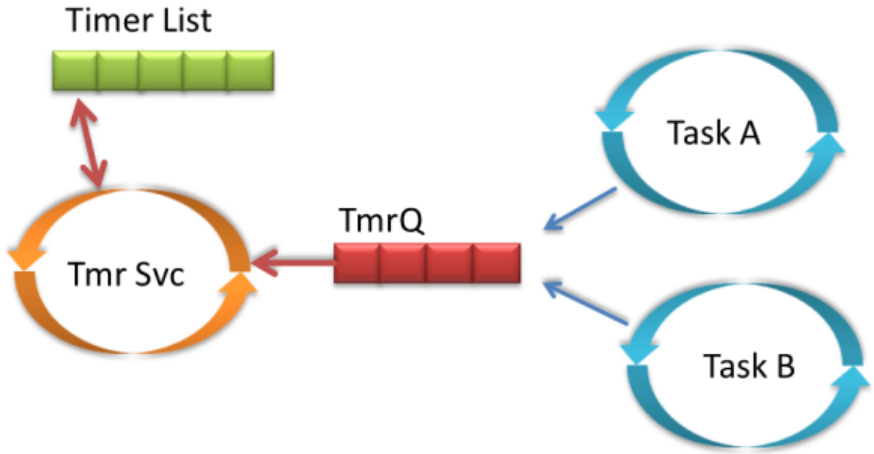
- Priority inheritance
  - ✓ If an high priority task blocks while getting a mutex
    - Priority of holding task temporarily raised to that of the blocked task
  - ✓ Ensure high priority task are blocked for the shortest time
  - ✓ Does not cure priority inversion
    - It just minimizes its effect 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.

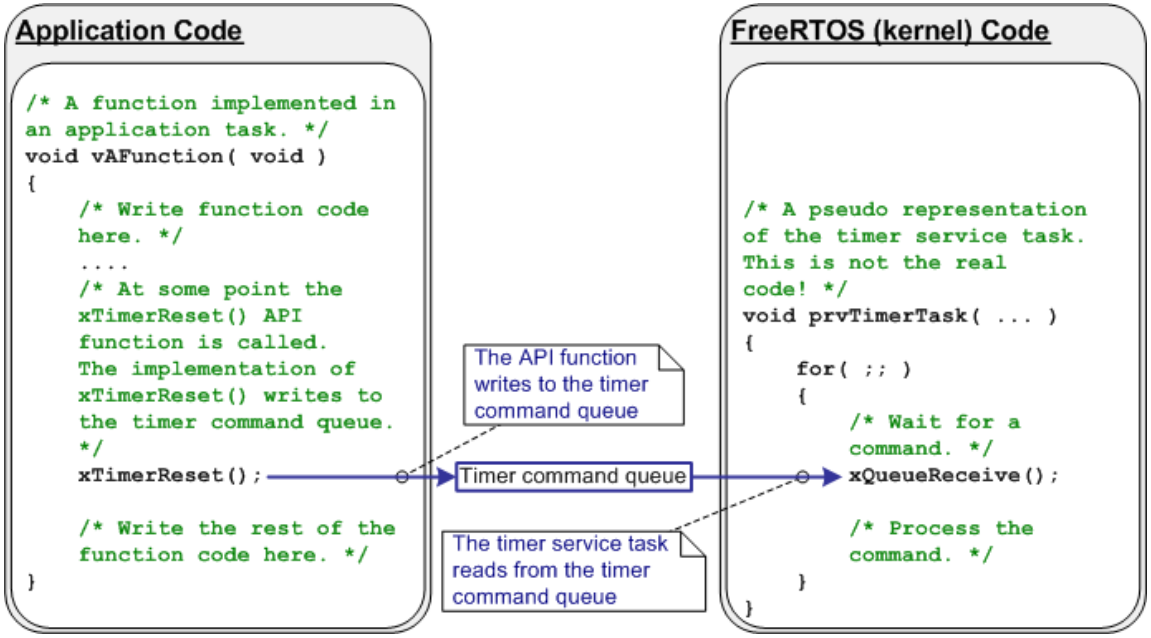
Software Timers

- 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

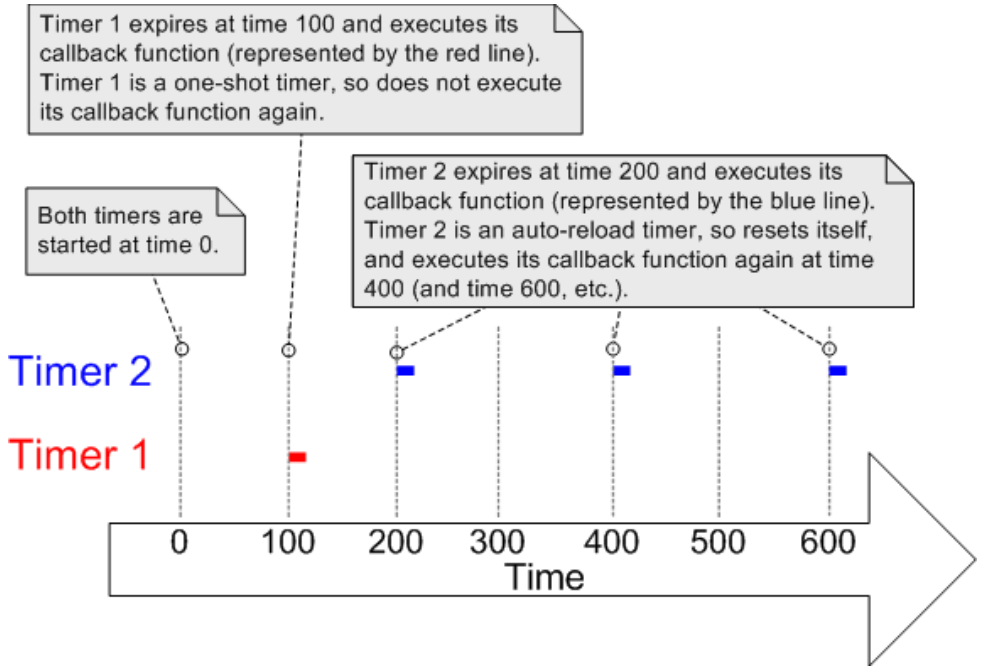


Software Timers - Usage

- 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

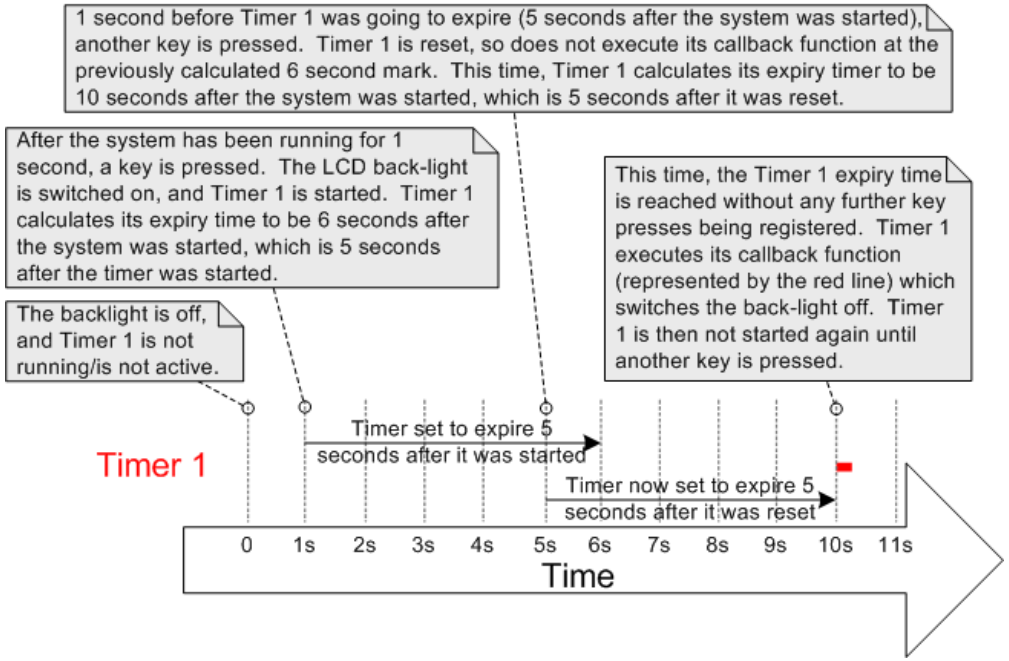
Software Timers - Types

- 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



Why choose FreeRTOS

- **One solution** fits many different architectures
- Undergoing continuous **active development**
- **Minimal** ROM, RAM and processing **overhead**
  - ✓ Typically kernel binary image in the range of 4 to 9 [KB]
- The core of the kernel is contained in only 3 C files
  - ✓ Very simple to study, understand... hack...
- Smaller and **easier real time processing** alternative
- **No software restriction**
  - ✓ On number of tasks and priorities that can be used

Features Overview

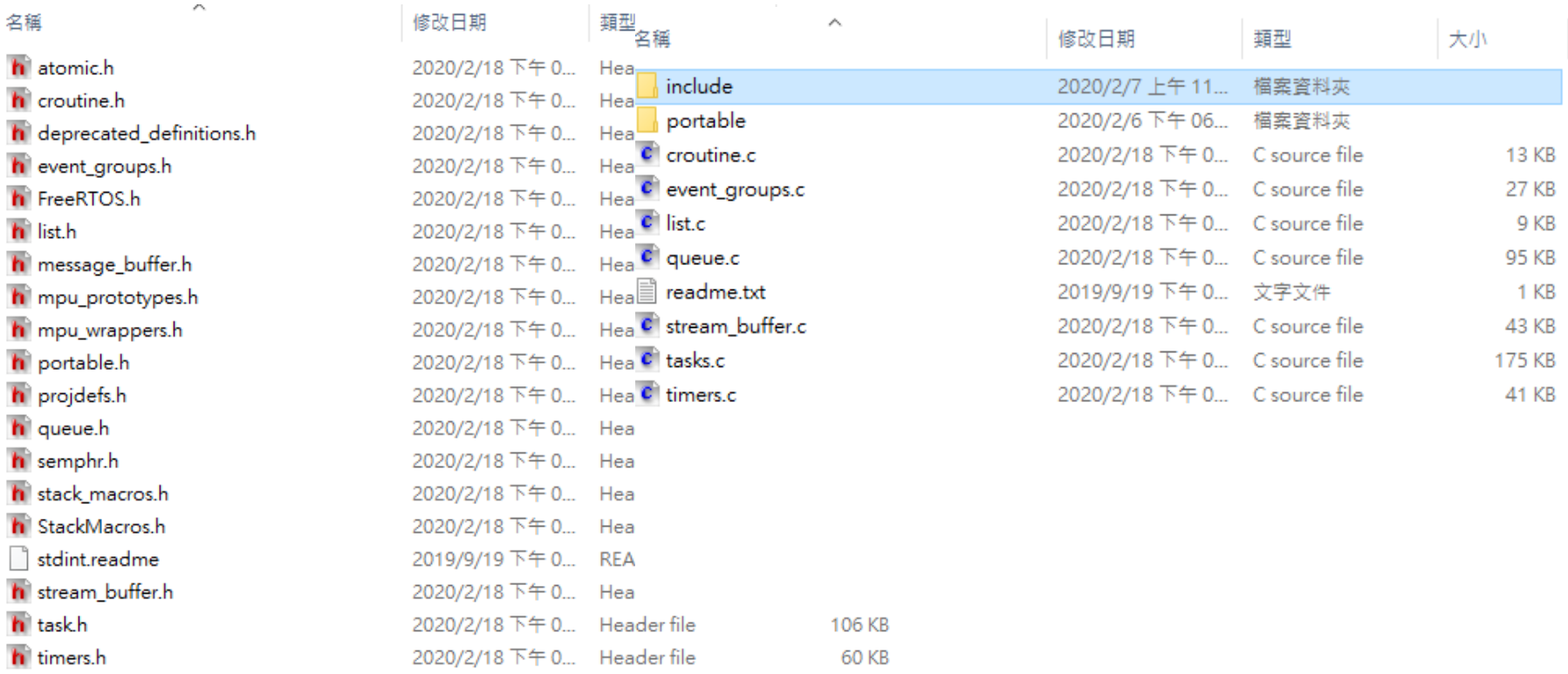
- **RTOS kernel**
  - ✓ Preemptive, cooperative and hybrid configuration options
- Supports Cortex M3 Memory Protection Unit (MPU)
- Supports both **tasks** and **co-routines**
- Efficient **communication** and **synchronization**
  - ✓ Between tasks, or between tasks and interrupts queues, binary semaphores, counti ng semaphores, recursive semaphores & mutex (with priority inheritance)

Basic Directory Structure

- The FreeRTOS download includes source code for every **processor port**, and every **demo application**
  - ✓ Greatly **simplifies distribution**, very **simple directory structure** the RT kernel is contained in just few sources

Platform Independent Code

- Sources common to every processor architecture

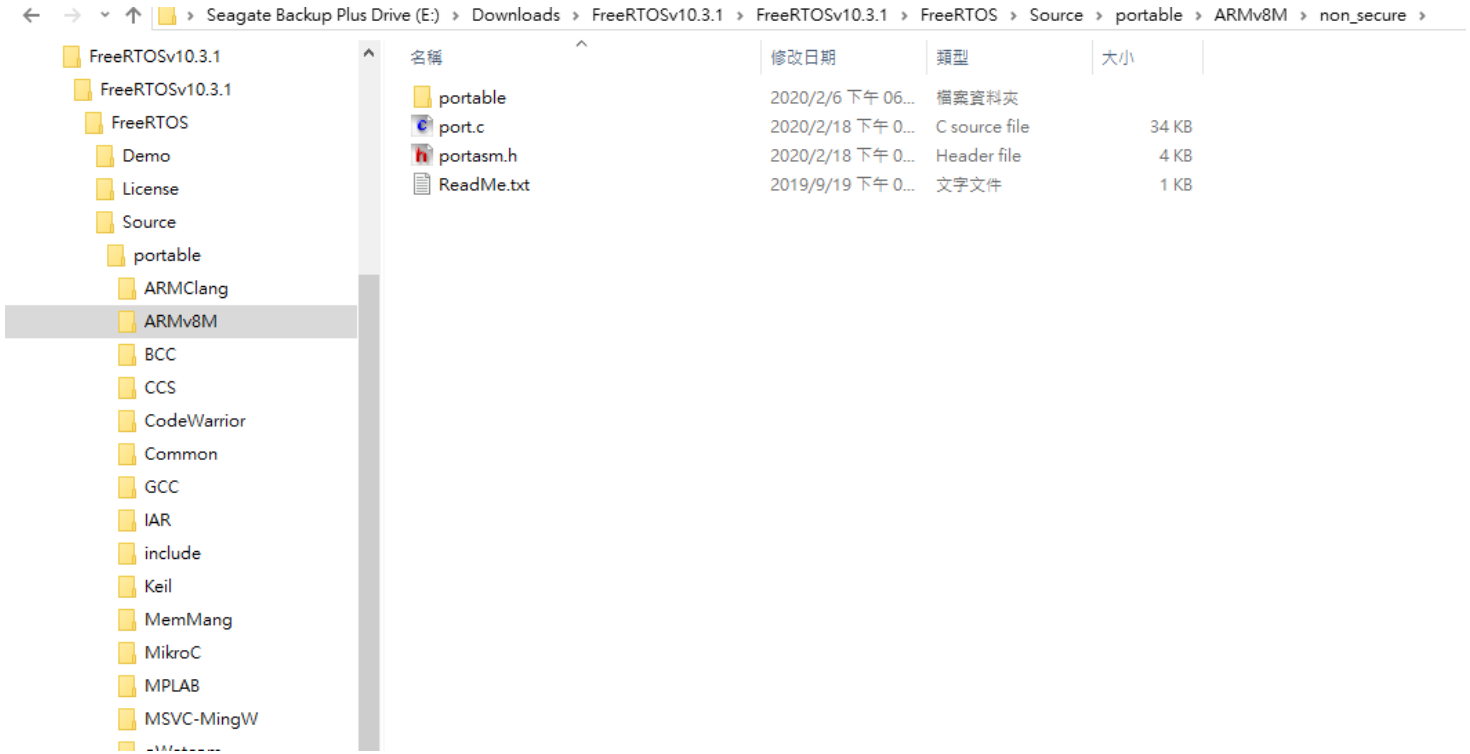


名稱	修改日期	類型	
atomic.h	2020/2/18 下午 0...	Hea	
croutine.h	2020/2/18 下午 0...	Hea	
deprecated_definitions.h	2020/2/18 下午 0...	Hea	
event_groups.h	2020/2/18 下午 0...	Hea	
FreeRTOS.h	2020/2/18 下午 0...	Hea	
list.h	2020/2/18 下午 0...	Hea	
message_buffer.h	2020/2/18 下午 0...	Hea	
mpu_prototypes.h	2020/2/18 下午 0...	Hea	
mpu_wrappers.h	2020/2/18 下午 0...	Hea	
portable.h	2020/2/18 下午 0...	Hea	
projdefs.h	2020/2/18 下午 0...	Hea	
queue.h	2020/2/18 下午 0...	Hea	
semphr.h	2020/2/18 下午 0...	Hea	
stack_macros.h	2020/2/18 下午 0...	Hea	
StackMacros.h	2020/2/18 下午 0...	Hea	
stdint.readme	2019/9/19 下午 0...	REA	
stream_buffer.h	2020/2/18 下午 0...	Hea	
task.h	2020/2/18 下午 0...	Header file	106 KB
timers.h	2020/2/18 下午 0...	Header file	60 KB

名稱	修改日期	類型	大小
include	2020/2/7 上午 11...	檔案資料夾	
portable	2020/2/6 下午 06...	檔案資料夾	
croutine.c	2020/2/18 下午 0...	C source file	13 KB
event_groups.c	2020/2/18 下午 0...	C source file	27 KB
list.c	2020/2/18 下午 0...	C source file	9 KB
queue.c	2020/2/18 下午 0...	C source file	95 KB
readme.txt	2019/9/19 下午 0...	文字文件	1 KB
stream_buffer.c	2020/2/18 下午 0...	C source file	43 KB
tasks.c	2020/2/18 下午 0...	C source file	175 KB
timers.c	2020/2/18 下午 0...	C source file	41 KB

Platform Specific Code

- Processor architecture requires a small amount of kernel code specific to that architecture



Creating a New FreeRTOS Port

- To yet unsupported microcontroller is **not trivial**
  - ✓ Very dependent on the processor and tools being used
- Within same processor family more straight forward
- Use existing ports as a reference

Setting up the Directory Structure

- Create directory containing '**port**' files for the [architecture]
  - ✓ FreeRTOS/Source/portable/[compiler name]/[processor name]
  - ✓ Copy stub **port.c** and **portmacro.h** into that directory

Start here | portmacro.h | port.c

```
495
496  void vPortEnterCritical( void ) /* PRIVILEGED_FUNCTION */
497  {
498      portDISABLE_INTERRUPTS();
499      ulCriticalNesting++;
500
501      /* Barriers are normally not required but do ensure the code is
502       * completely within the specified behaviour for the architecture. */
503      __asm volatile( "dsb" ::: "memory" );
504      __asm volatile( "isb" );
505  }
506  /*-----------------------------------------------------------*/
507
508  void vPortExitCritical( void ) /* PRIVILEGED_FUNCTION */
509  {
510      configASSERT( ulCriticalNesting );
511      ulCriticalNesting--;
512
513      if( ulCriticalNesting == 0 )
514      {
515          portENABLE_INTERRUPTS();
516      }
517  }
518  /*-----------------------------------------------------------*/
519
520  void SysTick_Handler( void ) /* PRIVILEGED_FUNCTION */
521  {
522  uint32_t ulPreviousMask;
523
524      ulPreviousMask = portSET_INTERRUPT_MASK_FROM_ISR();
```

portmacro.h

```
#define portSTACK_TYPE                          uint32_t
#define portBASE_TYPE                           long

typedef portSTACK_TYPE                          StackType_t;
typedef long                                    BaseType_t;
typedef unsigned long                           UBaseType_t;

#if( configUSE_16_BIT_TICKS == 1 )
    typedef uint16_t TickType_t;
    #define portMAX_DELAY ( TickType_t )        0xffff
#else
    typedef uint32_t TickType_t;
    #define portMAX_DELAY ( TickType_t )        0xffffffffUL

    /* 32-bit tick type on a 32-bit architecture, so reads of the tick count do
     * not need to be guarded with a critical section. */
    #define portTICK_TYPE_IS_ATOMIC             1
#endif
/*-----------------------------------------------------------*/

/**
 * Architecture specifics.
 */
#define portARCH_NAME                           "Cortex-M23"
#define portSTACK_GROWTH                        ( -1 )
#define portTICK_PERIOD_MS                      ( ( TickType_t ) 1000 / configTICK_RATE_HZ )
#define portBYTE_ALIGNMENT                      8
#define portNOP()
#define portINLINE                              __inline
#ifndef portFORCE_INLINE
    #define portFORCE_INLINE                    inline __attribute__(( always_inline ))
#endif
#define portHAS_STACK_OVERFLOW_CHECKING         1
#define portDONT_DISCARD                        __attribute__(( used ))
/*-----------------------------------------------------------*/
```

Setting up the Directory Structure

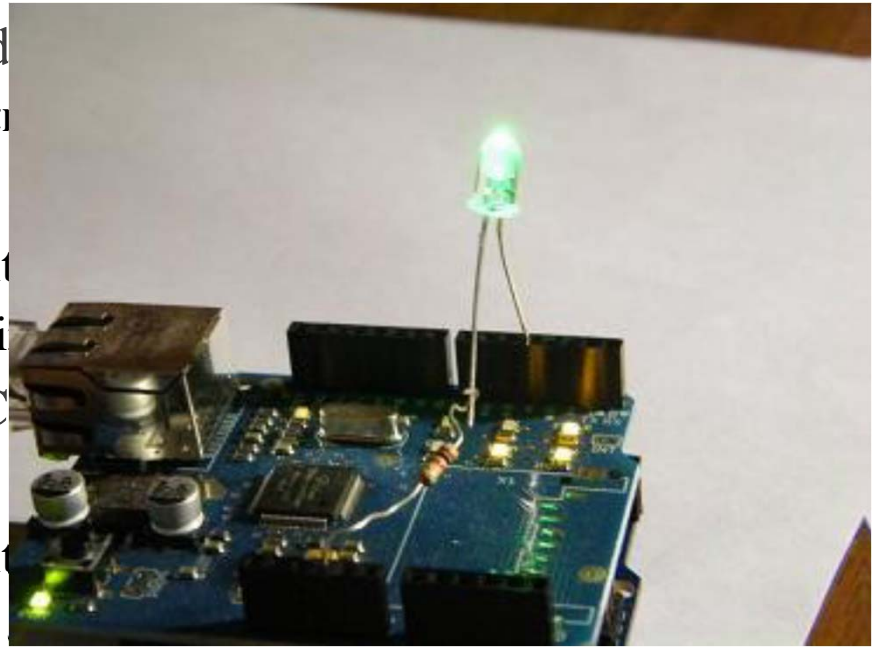
① Create directory containing '**port**' files for the [architecture]
- ✓ FreeRTOS/Source/portable/[compiler name]/[processor name]
- ✓ Copy stub **port.c** and **portmacro.h** into that directory

② Configure [architecture] stack growing d
- ✓ `portSTACK_GROWTH` macro in **portı**

③ Create a demo application directory
- ✓ **For example**, FreeRTOS/Demo/[archit
- ✓ Copy stub FreeRTOSConfig.h and mai

④ Properly setup macros into **FreeRTOSC**

⑤ Create a porting test subdirectory
- ✓ **For example**, FreeRTOS/Demo/[archit
- ✓ Implement required functions to setup



FreeRTOS: Co-Routines

- Supported since **FreeRTOS v4**
- Conceptually similar to tasks, fundamental differences
  - ✓ Stack usage
    - All co-routines of an application share a single stack
  - ✓ Scheduling and priorities
    - They use prioritised cooperative scheduling between them
  - ✓ Macro implementation
    - Provided through a set of macros
  - ✓ Restrictions on use

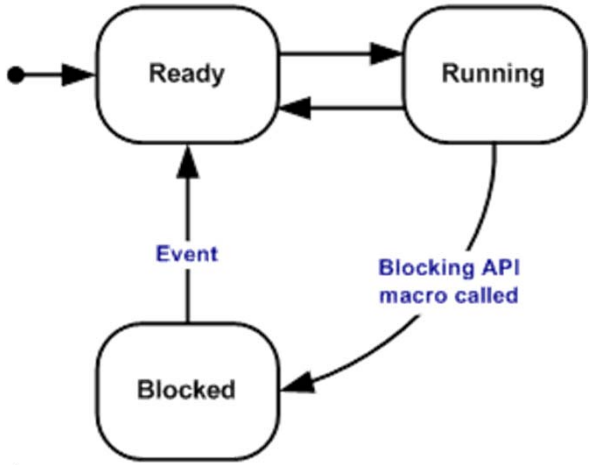
Co-Routines States

- A co-routine can exist in one of **this three states**

① Running
- ✓ Currently utilizing the processor

② Ready
- ✓ Able to execute
- ✓ Not currently executing

③ Blocked
- ✓ Not available for scheduling
- ✓ Currently waiting for temporal or external event



Implementing a Co-Routine

- A co-routine should
  - ✓ Have a function prototype of type `crCOROUTINE_CODE`
    - i.e. returns void and takes exactly two parameters: `xCoRoutineHandle` and **an index**
  - ✓ Start with `crSTART()` and end with `crEND()`
  - ✓ Never return

```
void vACoRoutineFunction(
    xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex) {
        crSTART( xHandle );
        For( ;; ) {
            // Co-routine application code here
        }
        crEND();
    }
```

  - ✓ Co-routines are created by calling xCoRoutineCreate()

Co-Routine Scheduling

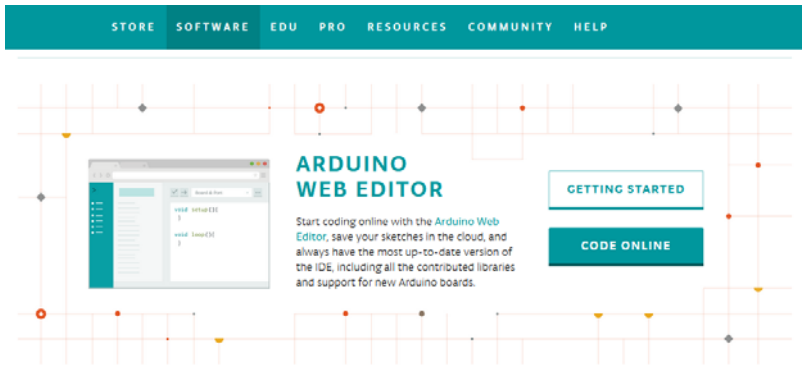
- By repeated calls to `vCoRoutineSchedule()`
- Best placed within **the idle task hook**
  - ✓ Even if your application only uses co-routines
  - ✓ Allows tasks and co-routines to be easily mixed
- Will only execute when there are no higher prio tasks

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

Start Working with Arduino?

- Arduino board
- USB cable
- Computer with USB interface
- USB driver and Arduino application (https://www.arduino.cc/en/Main/Software)



What is Arduino?

- The Arduino is a programmable hardware board that runs an 8-bit/16Mhz microcontroller with **a special bootloader** that allows users to upload programs to the microcontroller
  - ✓ It has **digital input pins** for input from switches and output to Actuators (LEDS or electrical motors)
  - ✓ It also has **analog pins** to accept inputs from voltage-based sensors.

- Arduino can be used to develop **stand-alone interactive objects** or can be **connected to software** on your computer

Why Arduino?

- **Open Source Hardware**
  - ✓ The Arduino system is open source -all hardware (made by Arduino distributors) has the schematics **and PCB layouts available online**

- **Open Source Bootloader**
  - ✓ The bootloaderis what runs on the chip before the program is run. It boots the chip and executes the program

- **Open Source Development Kit**
  - ✓ The development kit - what you use to program an Arduino board - is also available online. It is free, **open-source**

Arduino Terminology (I/II)

- **I/O Board**
  - ✓ The I/O Board is the "**brain**" of the operation (main microcontroller you program it from your computer).

- **Shield**
  - ✓ A Shield is a device that plugs into an I/O Board. These extend the capabilities of the I/O Board

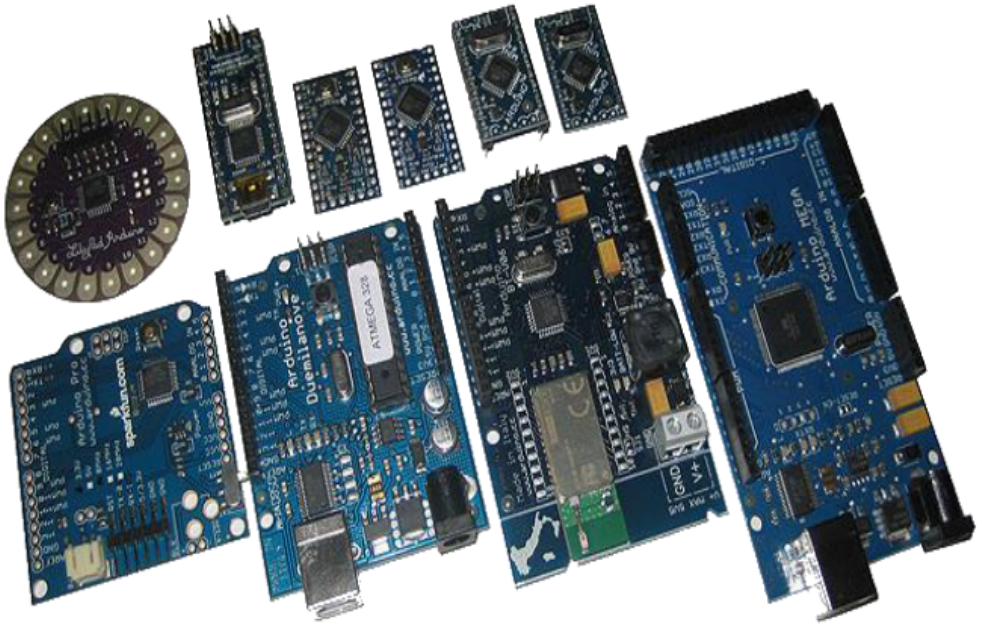
- **Sketch**
  - ✓ A Sketch is a program written for the board and shields.

Arduino Terminology (II/II)

- **Sensor**
  - ✓ Sensing components (Gas, etc.).
- **Modules**
  - ✓ Collecting serial data (GPS module, etc.)
- **Pin**
  - ✓ An input or output connected to something.
- **Digital**
  - ✓ Value is either **HIGH** or **LOW**
- **Analog**
  - ✓ Value ranges, usually from 0-255

Arduino Types

- Many different versions
  - ✓ Number of input/output channels
  - ✓ Processor
- Leonardo
- Due
- Micro/Nano
- LilyPad
- Esplora
- Uno/number one

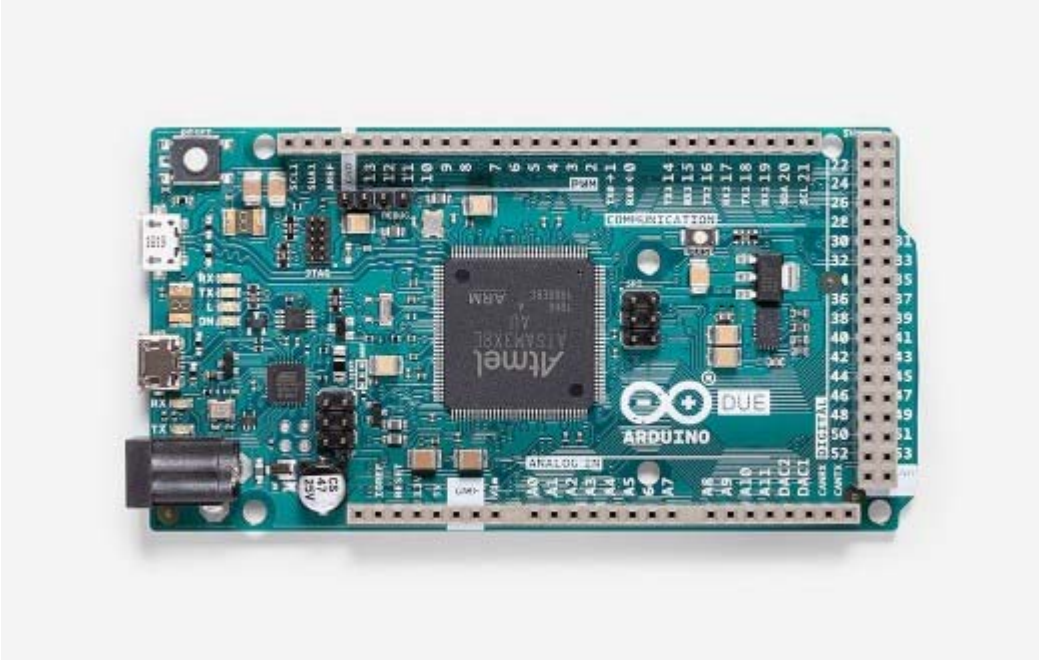


Leonardo

- Compared to the Uno, a slight upgrade
- Built in USB compatibility
- Presents to PC as a mouse or keyboard

Due

- Much faster processor, many more pins
- Operates on 3.3 volts
- Similar to the Mega



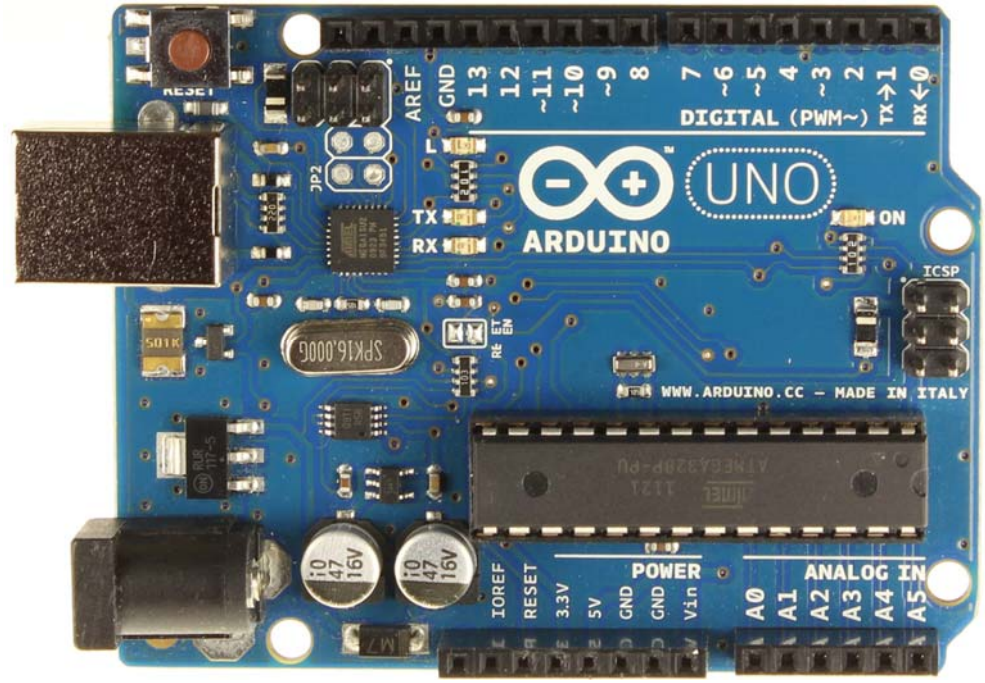
Micro/ Nano

- Arduino Nano is a surface mount breadboard embedded version with integrated USB.

- It is a smallest, complete, and breadboard friendly.

Uno (means one)

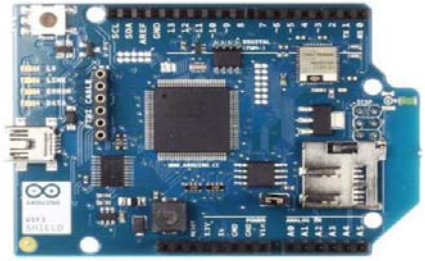
- The pins are in three groups:
  - ✓ 6 analog inputs
  - ✓ 14 digital input/output pins (of which 6 can be used as PWM outputs)
  - ✓ Power



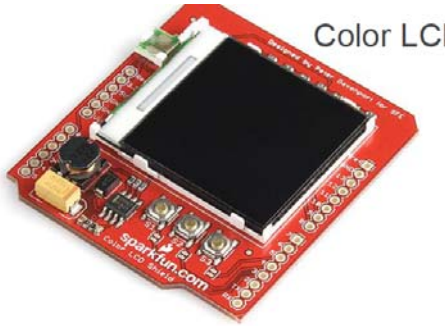
Shields

- Shields connect to the I/O board to extend it's functionality

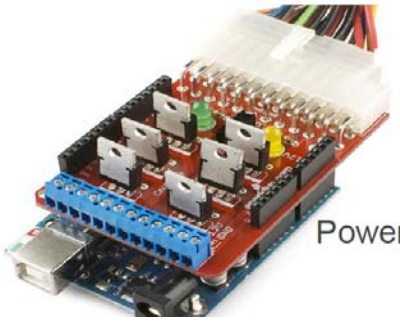
Wireless Network Shield



Color LCD Shield



GPS Shield



Power Driver Shield

More Shields…

- Communication shields -XBee, Ethernet, and Wifi

XBee Shield

Ethernet Shield

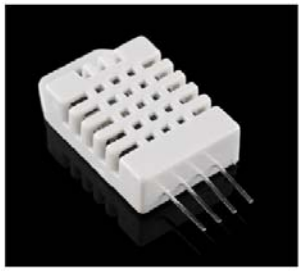
Wifi Shield

Modules

- Modules send serial data strings to the Arduino

GPS Module

Bluetooth Module



Temperature &
Humidity Sensor

RFID Module

Sensors and Modules

- Shields aren't the only way to extend an Arduino board -you can hook sensors to it!

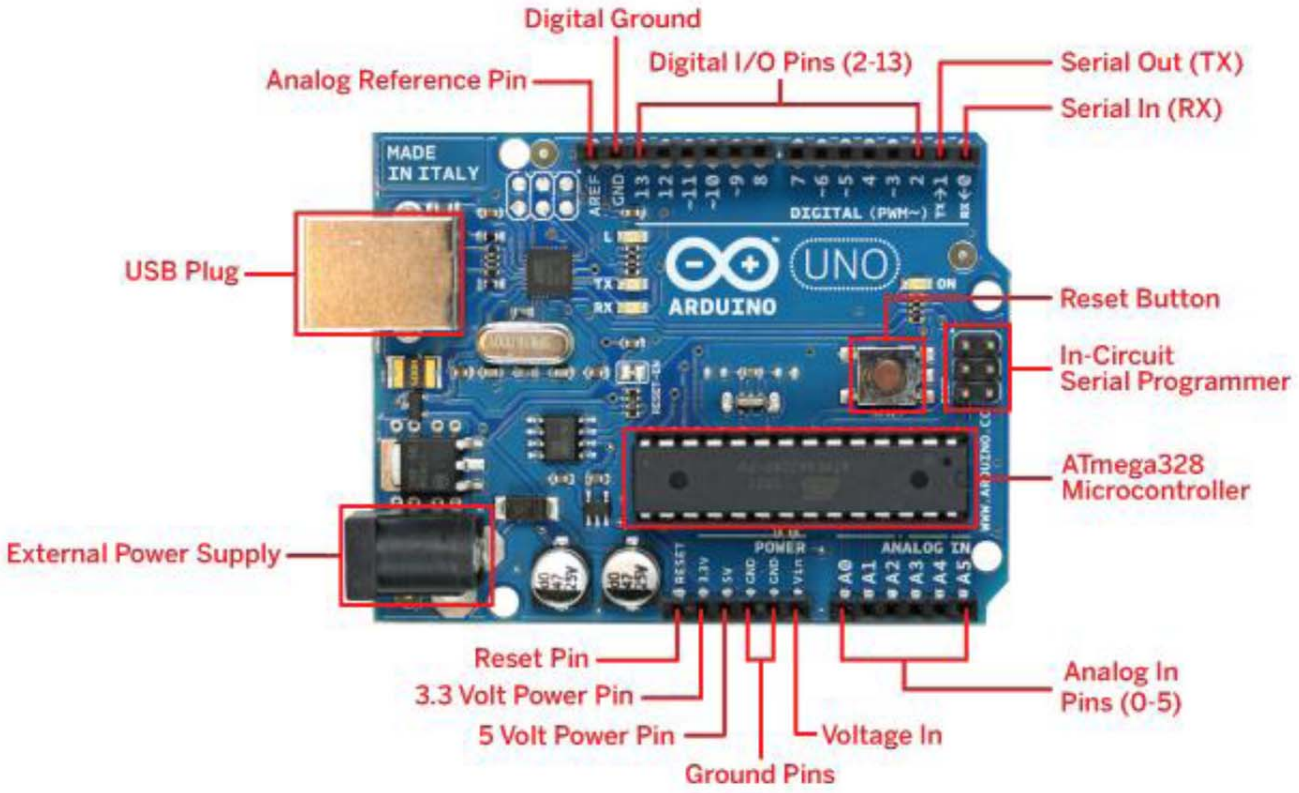
Temp & Humidity

Gas Sensor

Fingerprint Scanner

Flex Sensor

Arduino Uno Board Overview



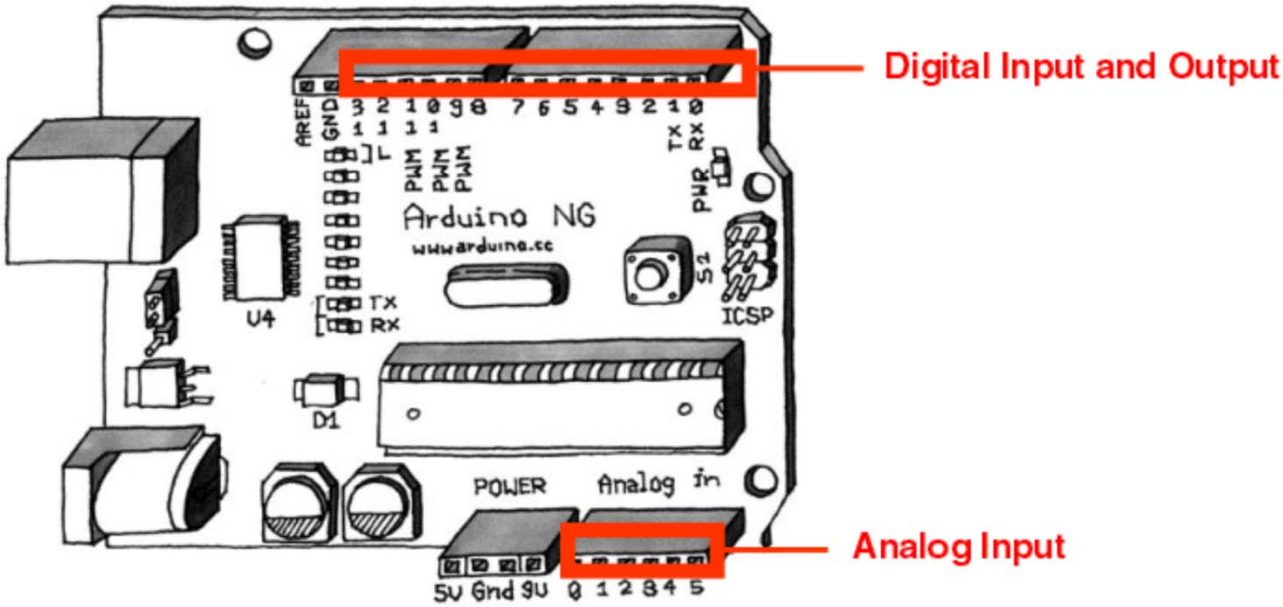
Arduino Uno/ATmega328

- The datasheet of Arduino Uno

Microcontroller	ATmega328
Operating Voltage	5 V
Input Voltage (recommended)	7-12 V
Input Voltage (limits)	6-20 V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 2 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz

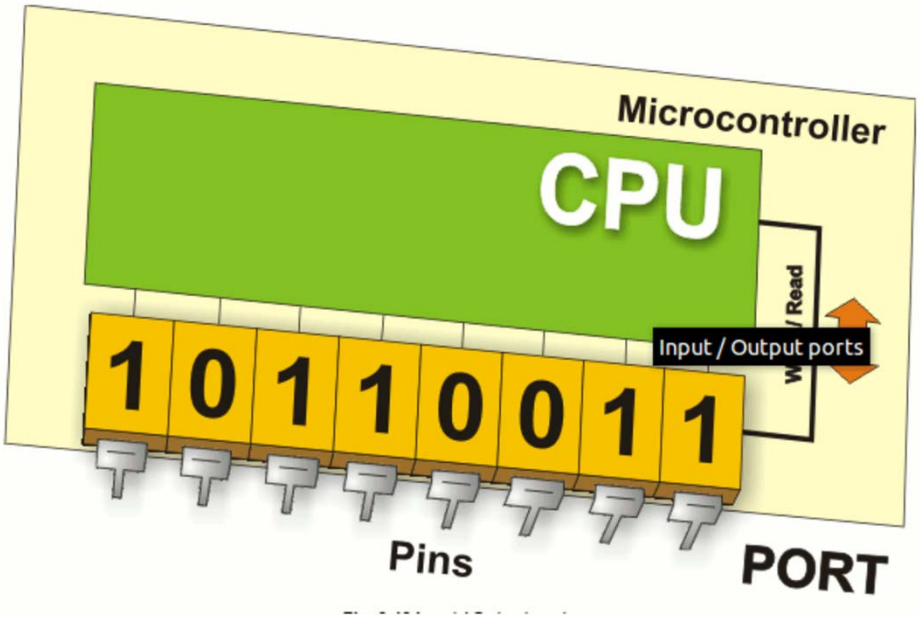
IO Pins

Two states (binary signal) vs. multiple states (continuous signal)



Arduino Digital I/0

- pinMode (pin_no., dir)
  - ✓ Sets pin to either INPUT or OUTPUT
- digitalRead (pin)
  - ✓ Reads HIGH or LOW from a pin
- digitalWrite (pin, value)
  - ✓ Writes HIGH or LOW to a pin



Arduino Analog I/0

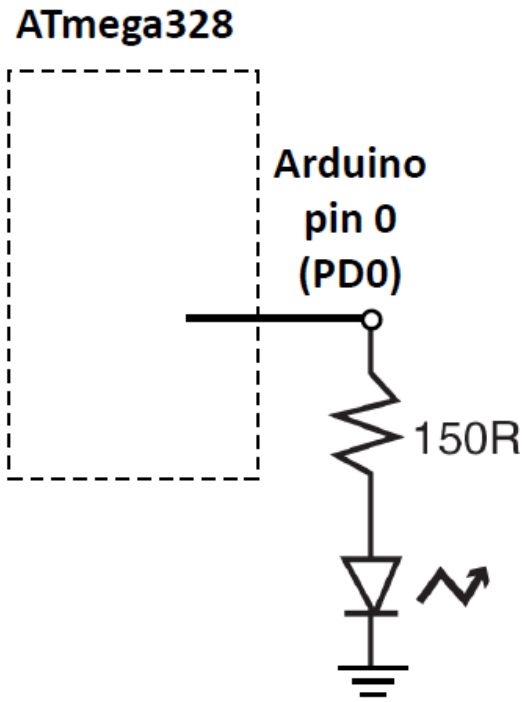
- analogWrite (pin, value)
  - ✓ pin: the pin to write to.
  - ✓ value: PWM the duty cycle: between 0 (always off) and 255

- `int x = analogRead (A0)`

Pin Used as an Output

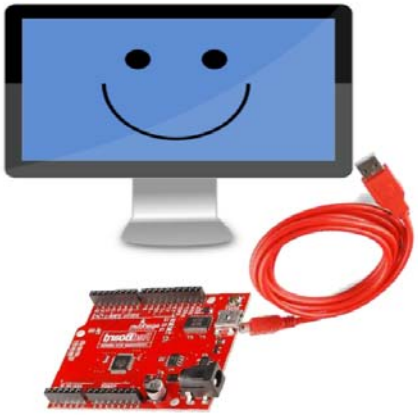
- Turn on LED, which is connected to pin Arduino pin 0 (PD0) (note the resistor!)

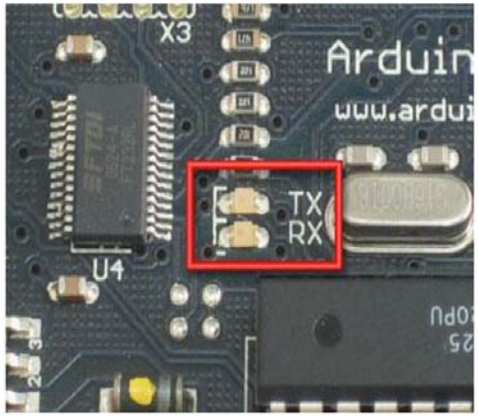
- What should the data direction be for pin 0 (PD0)?
  - ✓ pinMode (0, OUTPUT);
  - ✓ Turn on the LED
  - ✓ digitalWrite (0, HIGH);
  - ✓ Turn off the LED
  - ✓ digitalWrite (0, LOW);



Serial Communication

- RX blinks when the Arduino is receiving data
- TX blinks when the Arduino is transmitting data





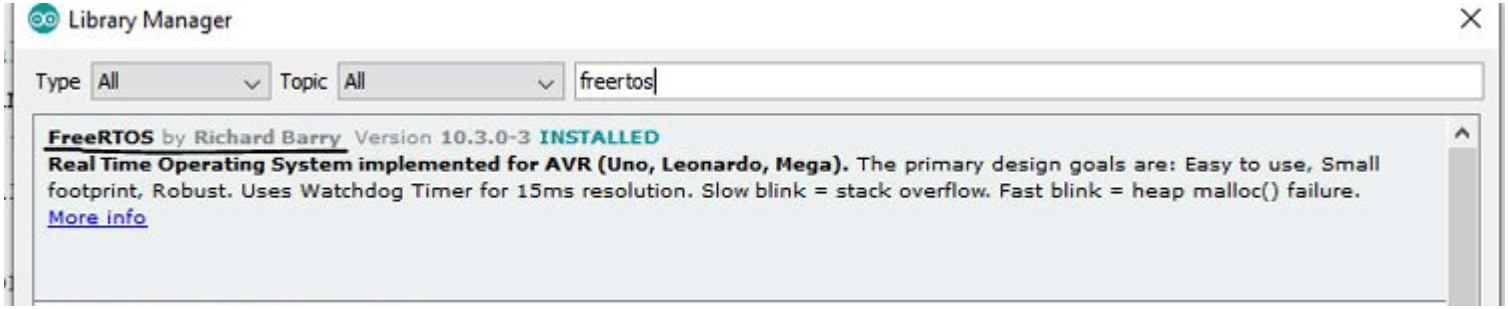
FreeRTOS with Arduino

- FreeRTOS is developed by Real Time Engineers Ltd.
- It is an **open-source** popular Real-Time Operating System kernel.
- It is used for embedded devices which as microcontrollers, Arduino.
- It is mostly written in C but some functions are written in assembly.

Installing Arduino FreeRTOS Library (1/2)

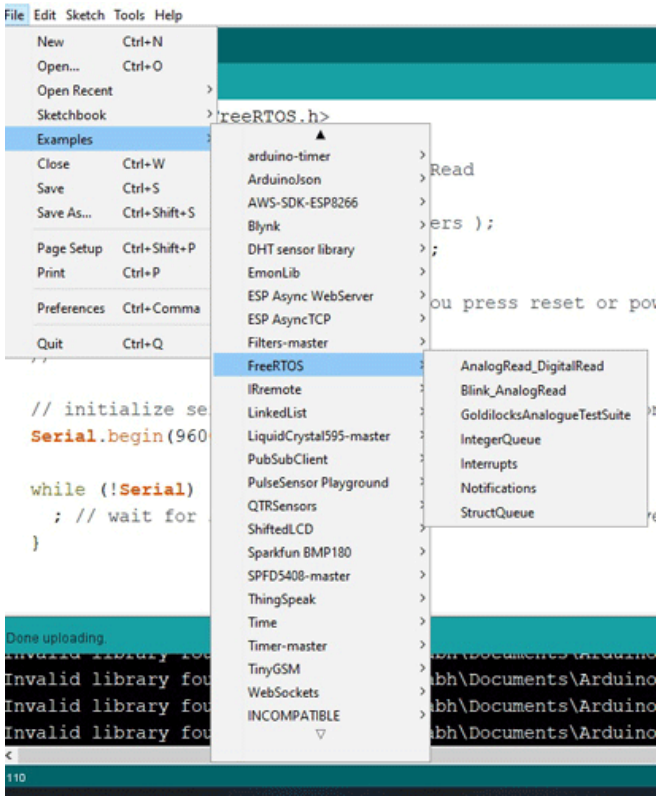
- Open Arduino IDE and go to Sketch -> Include Library -> Manage Libraries
- Search for FreeRTOS and install the library as shown below.

- Or You can download the library from github and Add the .zip file in Sketch-> Include Library -> Add .zip file



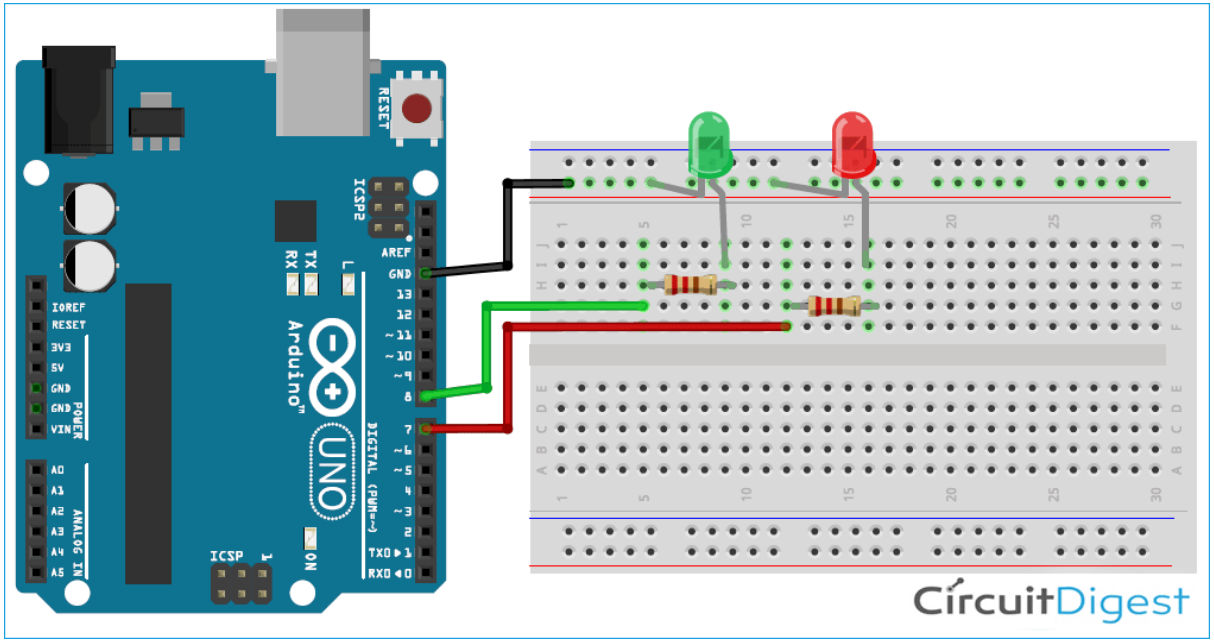
Installing Arduino FreeRTOS Library (2/2)

- Now, restart the Arduino IDE. This library provides some example code, also that can be found in `File -> Examples -> FreeRTOS` as shown below.



Circuit Diagram

- Below is the circuit diagram for creating Blinking LED task using FreeRTOS on Arduino:



Creating FreeRTOS tasks in Arduino IDE

- First, include Arduino FreeRTOS header file as

```
#include <Arduino_FreeRTOS.h>
```

- Give the function prototype of all functions that you are writing for execution which is written as

```
void Task1( void *pvParameters );
void Task2( void *pvParameters );
```

- Now, in void setup() function, create tasks and start the task scheduler

```
xTaskCreate(task1,"task1",128,NULL,1,NULL);
xTaskCreate(task2,"task2",128,NULL,2,NULL);
```

Example Code (1)

- From the above basic structure explanation, include the Arduino FreeRTOS header file. Then make function prototypes. As we have three tasks, so make three functions and it's prototypes.

```
#include <Arduino_FreeRTOS.h>
void TaskBlink1( void *pvParameters );
void TaskBlink2( void *pvParameters );
void Taskprint( void *pvParameters );
```

Example Code (2)

- In void **setup() function**, initialize serial communication at **9600** bits per second and create all three tasks using xTaskCreate() API. Initially, make the priorities of all tasks as '1' and start the scheduler

```
void setup() {
Serial.begin(9600);
xTaskCreate(TaskBlink1,"Task1",128,NULL,1,NULL);
xTaskCreate(TaskBlink2,"Task2 ",128,NULL,1,NULL);
xTaskCreate(Taskprint,"Task3",128,NULL,1,NULL);
vTaskStartScheduler();
}
```

Example Code (3)

- Now, implement all three functions as shown below for task1 LED blink.

```
void TaskBlink1(void *pvParameters)
{
pinMode(8, OUTPUT);
while(1)
{
digitalWrite(8, HIGH);
vTaskDelay( 200 / portTICK_PERIOD_MS );
digitalWrite(8, LOW);
vTaskDelay( 200 / portTICK_PERIOD_MS );
}
}
```

Example Code (4)

- Task 2 LED blink.

```
void TaskBlink2(void *pvParameters)
{
  pinMode(7, OUTPUT);
  while(1)
  {
    Serial.println("Task2");
    digitalWrite(7, HIGH);
    vTaskDelay( 300 / portTICK_PERIOD_MS );
    digitalWrite(7, LOW);
    vTaskDelay( 300 / portTICK_PERIOD_MS );
  }
}
```

Example Code (5)

- Task 3: Print Task.

```
void Taskprint(void *pvParameters)
{
	int counter = 0;
	while(1)
	{
		counter++;
		Serial.println(counter);
		vTaskDelay( 500 / portTICK_PERIOD_MS );
	}
}
```

Result