Embedded Systems

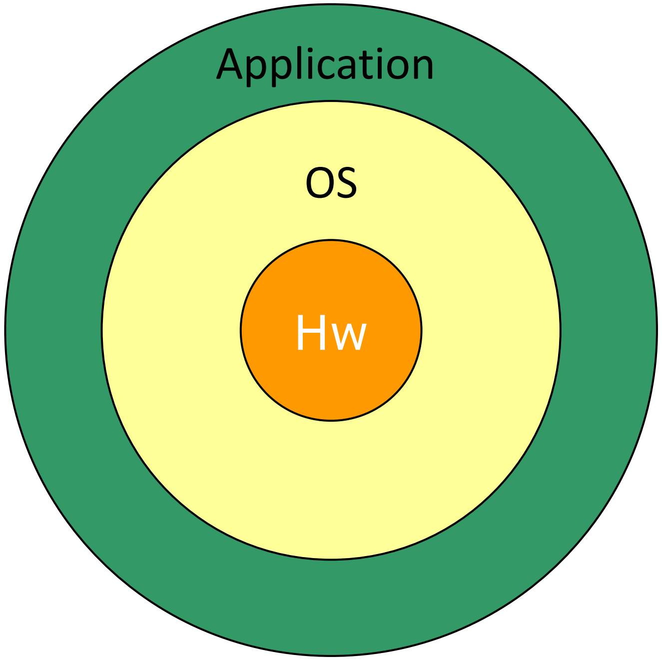
Master 's Degree in Informatics Engineering

Real Time Systems

- Computational systems classification

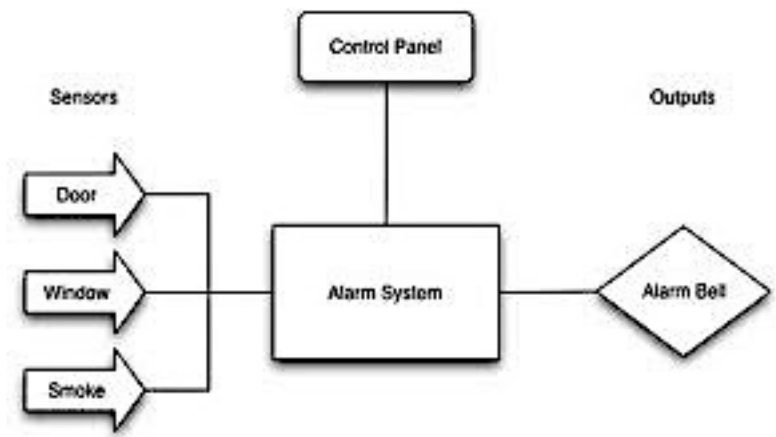
  - General Purpose System
  - Embedded System
  - Real Time System

- Computational systems classification
  - **General Purpose System**
  - There's no hardware restrictions
  - Standard Hardware: PC
  - General Purpose OS
  - Goal: Support all kind of applications



- Computational systems classification
  - General Purpose System
  - **Embedded System**

- Limited resources
  - Computational capabilities
  - Memory
  - Energy consumption
- Environmental dependency and interaction
  - Sensors
  - Actuators
- Different applications
- Real Time response



- Computational systems classification

  - General Purpose System
  - Embedded System
  - **Real Time Systems**

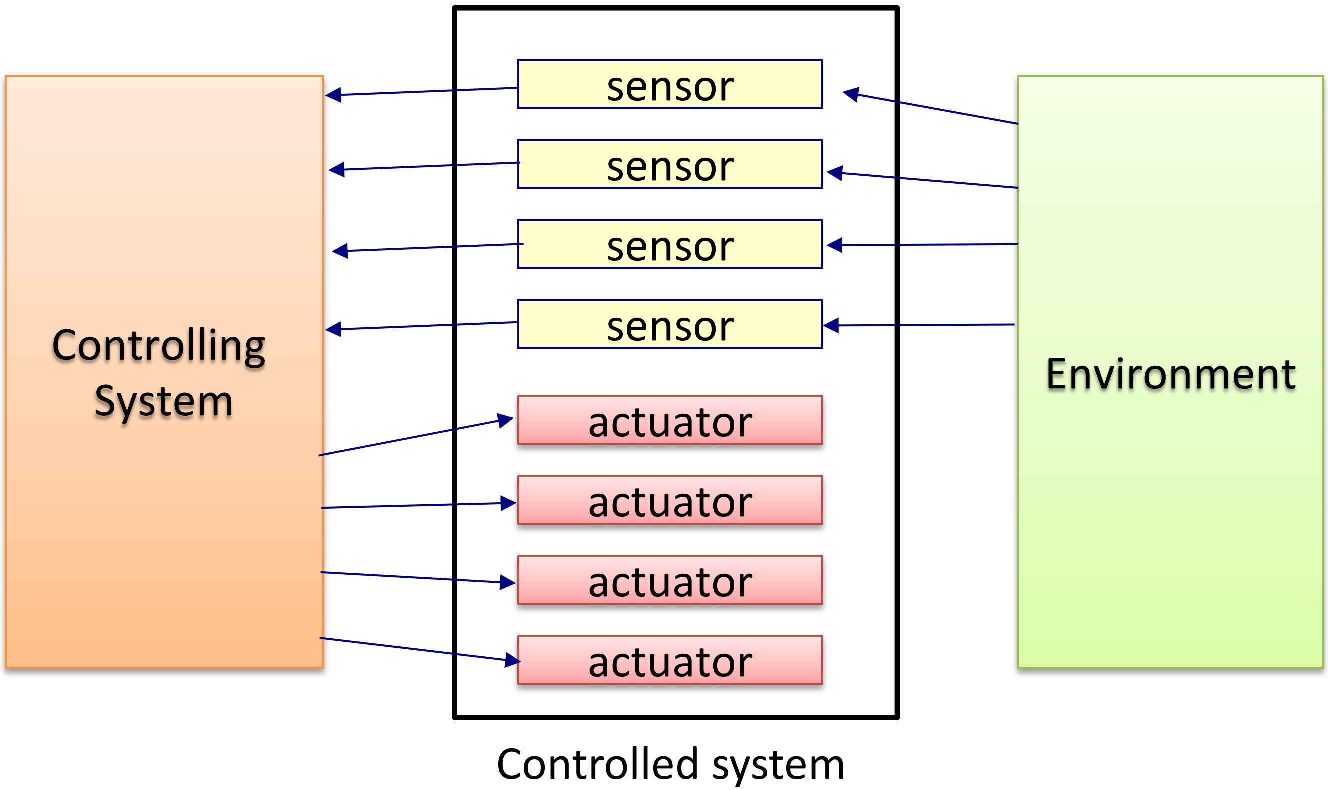
  - Tasks have to – Correctness result & hard deadline
  - Example:
    - A vehicle – 90Km/h (25m/s)
    - Any sensor detects an obstacle 75m far away
    - The vehicle needs 25m to stop
    - The system has to react on:

$$t = \frac{e}{v} = \frac{50m}{25m/s} = 2s$$

    - If there are many tasks to control it could be difficult to stop the vehicle: control the gear, wheel, abs, radio, air conditioner, ….

- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - Definitions
      - Real-time systems often are comprised of a *controlling* system, *controlled* system and *environment*.
        - ✓ *Controlling* system: acquires information about environment using *sensors* and controls the environment with *actuators*.
      - Timing constraints derived from physical impact of controlling systems activities. Hard and soft constraints.
        - ✓ Periodic Tasks: Time-driven recurring at regular intervals.
        - ✓ Aperiodic: event-driven.

- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - Definitions



- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - Definitions

Job completed → Inputs are processed and outputs generated

- **Timing constraint:** constraint imposed on timing behavior of a job: hard or soft.
- **Release Time**: Instant of time job becomes available for execution. If all jobs are released when the system begins execution, then there is said to be no release time
- **Deadline**: Instant of time a job's execution is required to be completed. If deadline is infinity, then job has no deadline.
- **Response time**: Length of time from release time to instant job completes.

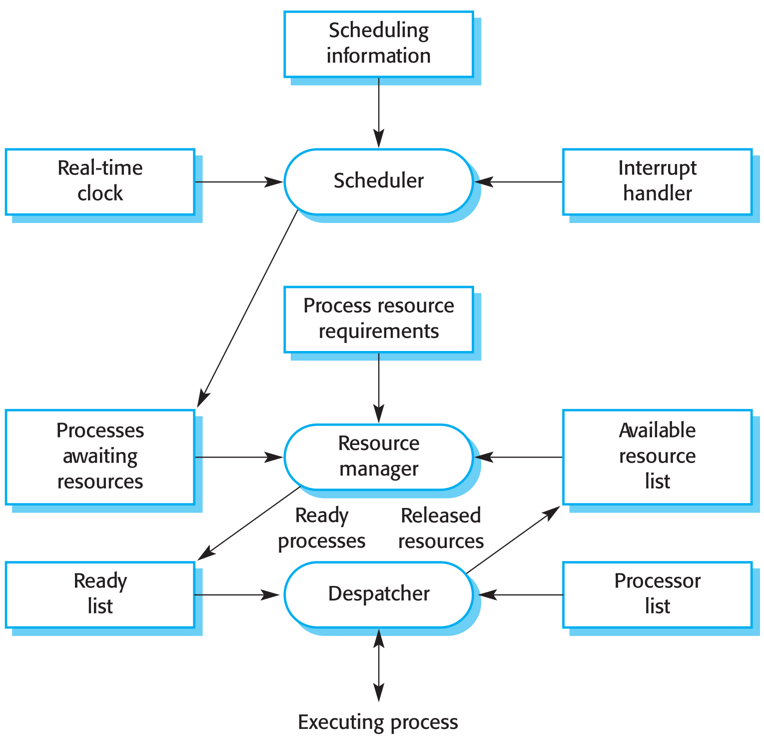
- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - System design
      - Design both the *hardware and the software* associated with system. Partition functions to either hardware or software.
      - Design *decisions* should be made on the basis on non-functional system requirements.
      - Hardware delivers better performance but potentially longer development and less scope for change.

- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - System design
      - Identify the *stimuli* to be processed and the required responses to these stimuli.
      - For each stimulus and response, identify the *timing constraints*.
      - Aggregate the stimulus and response processing into *concurrent processes*. A process may be associated with each class of stimulus and response.
      - Design *algorithms* to process each class of stimulus and response. These must meet the given timing requirements.
      - Design a scheduling system which will ensure that processes are started in time to meet their *deadlines*.
      - Integrate using a *real-time operating system*.

Key Points

- Real-time system correctness depends not just on what the system does but also on how fast it reacts.
- A general RT system model involves associating processes with sensors and actuators.
- Real-time systems architectures are usually designed as a number of concurrent processes.
- Real-time operating systems are responsible for process and resource management.
- Monitoring and control systems poll sensors and send control signal to actuators.
- Data acquisition systems are usually organised according to a producer consumer model.

- Computational systems classification
  - General Purpose System
  - Embedded System
  - **Real Time Systems**
    - Real-Time Operative Systems - Components



Why Use a Real-Time kernel?

Hard coded embedded systems are faster however:

- Abstracting away timing information. This allows the structure of the application code to be simplier and smaller.
- Maintainability/Extensibility. Fewer dependencies between modules.
- Task modularity.
- Event-driven means improved effiency.
- Easier power management when idle task is detected.
- Flexible interrupt handling.

Real Time Operative Systems

Master 's Degree in Informatics Engineering

FreeRTOS

Free embedded RTOS

https://freertos.org/

Considerations

- Static design
  - Everything in the kernel is static, nowhere memory is allocated or freed
  - Allocator subsystems are optionally available, built as a layer on top of the fully static kernel
- No error conditions
  - System APIs have no error conditions
  - All the static core APIs always succeed if correct parameters are passed
- Simple, fast and compact
  - Each API function should have the parameters you would expect
  - Note, first “fast” then “compact”
- Portable
  - Efficient layered architecture
  - Non portable parts are small and enclosed in well defined layers

Background Information

1. It supports >25 architecture ports

2. FreeRTOS RT kernel is portable, open source, royalty free

3. OpenRTOS is a commercialized version

4. Widely used in real projects (aircrafts, rockets, …)

5. It includes a kernel/scheduler on top of wich MCU applications can be built to meet their hard real-time constraints

6. Applications are organized as independent tasks based on priorities.

Benefits of using real-time kernel

1. Abstracting away timing information

   The kernel is responsible for execution timing. This allows the structure of the application code to be simpler, and the overall code size to be smaller.

2. Maintainability/Extensibility

   Fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Application performance is less susceptible to changes in the underlying hardware.

3. Modularity

   Tasks are independent modules, each of which should have a well-defined purpose.

4. Improved efficiency

   Applications are completely event-driven. Code executes only when there is something that must be done.

Standard FreeRTOS features

- Fixed priority Pre-emptive and co-operative scheduler
- Very flexible task priority assignment and management
- Flexible, fast and light weight task notification mechanism
- Queues
- Binary and counting semaphores, Mutexes and Recursive Mutexes
- Software timers
- Event groups
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering

Configuration

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

Some examples:

`configUSE_PREEMPTION`

`configCPU_CLOCK_HZ` – CPU clock frequency, not necessarily the bus frequency.

`configTICK_RATE_HZ` – RTOS tick frequency that dictates interrupt frequency.

`configMAX_PRIORITIES` – Total number of priority levels. Each level creates a new list, so memory sensitive machines should keep this to a minimum.

….

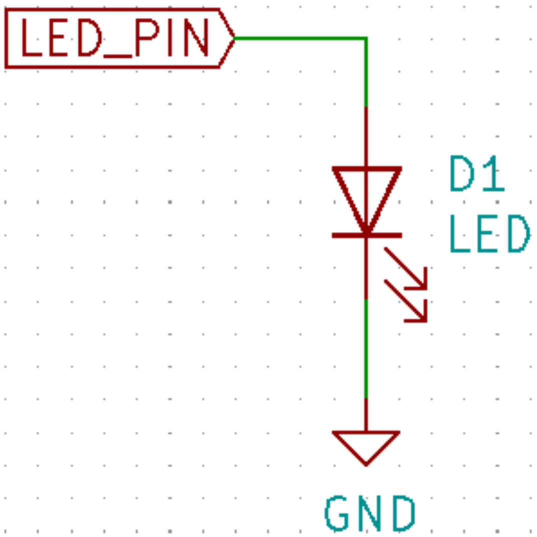
FreeRTOS Source Files common to All Ports

```
FreeRTOS
    │
    └─Source
        │
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

- `tasks.c` – Main application code.
- `list.c` – Structures to maintain scheduler and other functions
- `queue.c` – Queue and Semaphore services.
- `timers.c` – Time functionalities.
- `event_groups.c` – Allows event group interactions.
- `croutine.c` – Just used for really tiny MCU. It implements co-routines capabilites, they are based on macros.

Running example – LED Blink

```
#define LED_PIN 10
#define ARDUINO_RUNNING_CORE 0

void TaskBlink( void *pvParameters );

void setup() {

  xTaskCreatePinnedToCore(
    TaskBlink       // Associated function
    ,  "TaskBlink"  // Assigned label
    ,  1024         // Stack size
    ,  NULL
    ,  2            // Priority
    ,  NULL
    ,  0);          //  ARDUINO_RUNNING_CORE

}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskBlink(void *pvParameters)
{
  (void) pvParameters;

  pinMode(LED_PIN, OUTPUT);

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(100);
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(100);
  }
}
```



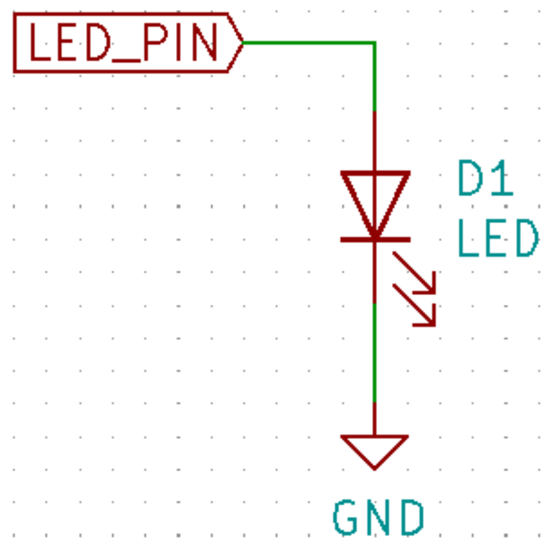
Running example – LED Blink

```
#define LED_PIN 10
#define ARDUINO_RUNNING_CORE 0

void TaskBlink( void *pvParameters );

void setup() {

  xTaskCreatePinnedToCore(
    TaskBlink       // Associated function
    , "TaskBlink"   // Assigned label
    , 1024          // Stack size
    , NULL
    , 2             // Priority
    , NULL
    , 0);           //  ARDUINO_RUNNING_CORE

}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskBlink(void *pvParameters)
{
  (void) pvParameters;

  pinMode(LED_PIN, OUTPUT);

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(100);
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(100);
  }
}
```

Global declarations , initialization,...

Running example – LED Blink

```
#define LED_PIN 10
#define ARDUINO_RUNNING_CORE 0

void TaskBlink( void *pvParameters );

void setup() {

  xTaskCreatePinnedToCore(
    TaskBlink        // Associated function
    ,  "TaskBlink"   // Assigned label
    ,  1024          // Stack size
    ,  NULL
    ,  2             // Priority
    ,  NULL
    ,  0);           //  ARDUINO_RUNNING_CORE

}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskBlink(void *pvParameters)
{
  (void) pvParameters;

  pinMode(LED_PIN, OUTPUT);

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(100);
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(100);
  }
}
```



User code defined as concurrent threads/tasks that will be executed. They are event driven.

Running example – LED Blink

```
#define LED_PIN 10
#define ARDUINO_RUNNING_CORE 0

void TaskBlink( void *pvParameters );

void setup() {

  xTaskCreatePinnedToCore(
    TaskBlink       // Associated function
    ,  "TaskBlink"  // Assigned label
    ,  1024         // Stack size
    ,  NULL
    ,  2            // Priority
    ,  NULL
    ,  0);          //  ARDUINO_RUNNING_CORE

}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskBlink(void *pvParameters)
{
  (void) pvParameters;

  pinMode(LED_PIN, OUTPUT);

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(100);
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(100);
  }
}
```

Task definition and pinned to core 0.

Considerations

- Source Code Conventions:
  - **Variables prefixes**: ‘c’ for char, ‘s’ for int16_t (short), ‘l’ int32_t (long), and ‘x’ for BaseType_t and any other non-standard types (structures, task handles, queue handles, etc.).

  - Name for **functions** prefixes:
    - `vTaskPrioritySet()` - returns a void and is defined within task.c.
    - `xQueueReceive()` - returns a variable of type BaseType_t and is defined within `queue.c`
    - `pvTimerGetTimerID()` - returns a pointer to void and is defined within `timers.c`

    * File scope (private) functions are prefixed with ‘prv’
  - **Macro Names** - are written in upper case, and prefixed with lower case letters that indicate where the macro is defined.
    - `portMAX_DELAY`, `pdTRUE`, …

Tasks definitions

- Tasks have their own context. No dependencies on other tasks unless defined.
- One task executes at a time.
- Tasks have no knowledge of scheduling activity. The scheduler handles context switching.
- Tasks have their own stack.
- Priorities or Preemptive scheduling options.
- `xTaskCreate` – Allows to create a task
- `vTaskDelete` – Delete the tasks and allows the idle task to free the allocated memory (`vTaskDelete (NULL), vTaskDelete (TaskHandle_t)`)

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

* API Reference >> https://www.freertos.org/a00106.html
* FreeRTOS Configuration >> https://www.freertos.org/a00110.html

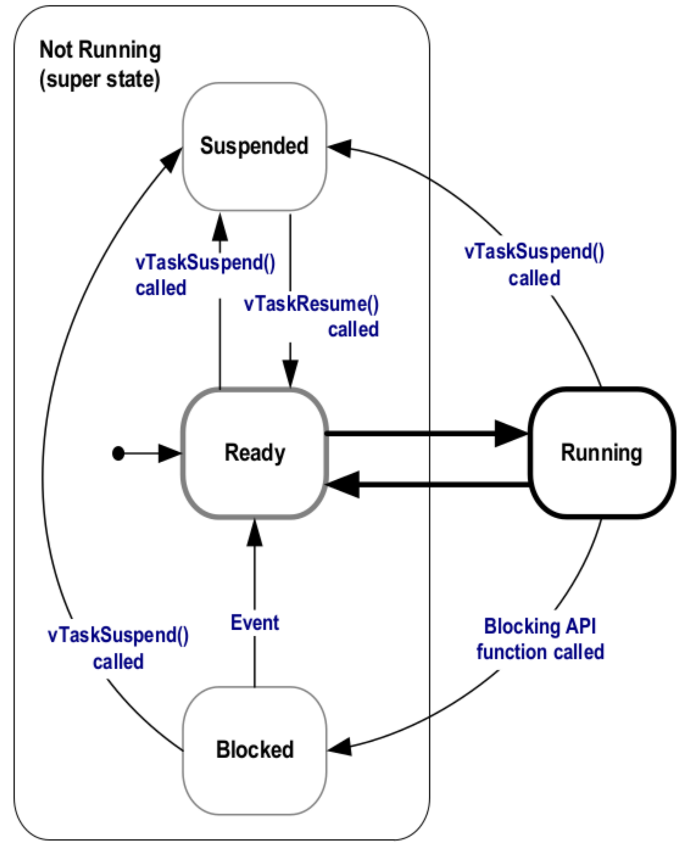
Tasks definitions

- pvTaskCode – Pointer to the function that implements the task
- pcName – Descriptive name for the task. It is not used for FreeRTOS anyway.
- usStasckDepth – Number of words the stack can hold.
- pvParameters – Task function accept a parameter of type pointer to void (void *).
- uxPriority – Defines the task priority. Priorities can be assigned form 0 (the lowest) to configMAX_PRIORITIES -1 (the highest)
- pxCreatedTask – Used to pass out a handle to the task being created, can be used to change priority, delete the task, …
- Returned value – pdPASS or pdFAIL

```
xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                "Task 1",/* Text name for the task.  This is to facilitate
                         debugging only. */
                1000,    /* Stack depth - small microcontrollers will use much
                         less stack than this. */
                NULL,    /* This example does not use the task parameter. */
                1,       /* This task will run at priority 1. */
                NULL );  /* This example does not use the task handle. */

/* Create the other task in exactly the same way and at the same priority. */
xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );
```

Task definition

- Tasks have their own context. No dependencies on other tasks unless defined.
- One task executes at a time.
- Tasks have no knowledge of scheduling activity. The scheduler handles context switching.
- Tasks have their own stack.
- Priorities or Preemptive scheduling options.
- Tasks can change their own priority, as well as the priority of other tasks.
- Tasks can create new ones.
- `IdleTask` priority = 0. This task is created automatically when the scheduler started.

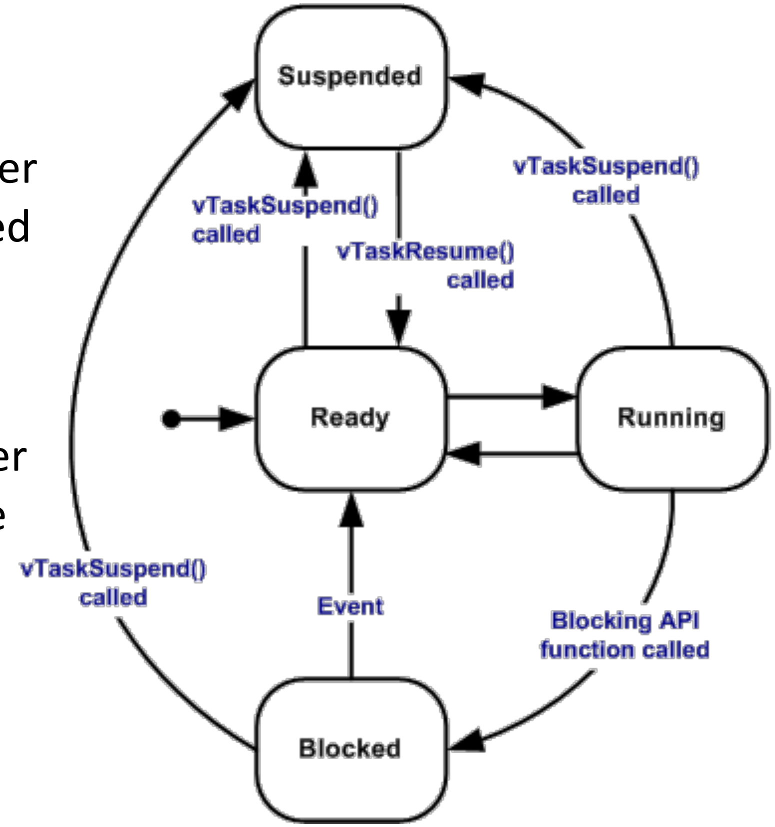


Task Scheduler

FreeRTOSConfig.h > configUSE_PREEMPTION - configUSE_TIME_SLICING

- **Preemptive**. This mode is activated by setting configUSE_PREEMPTION to 1. In this mode the thread using the CPU is preempted by its peers after its time slot has been used. The Quantum is defined in configUSE_TIME_SLICING (default 1).

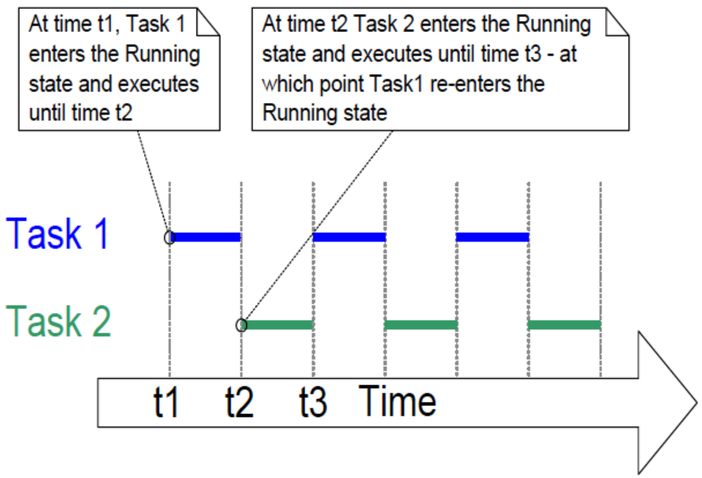
- **Cooperative**. This mode is activated by setting configUSE_PREEMPTION to 0 then the scheduler will still run the highest priority. Cooperative mode is preferable because the kernel becomes *slightly more efficient* because it does not have to handle time slots.

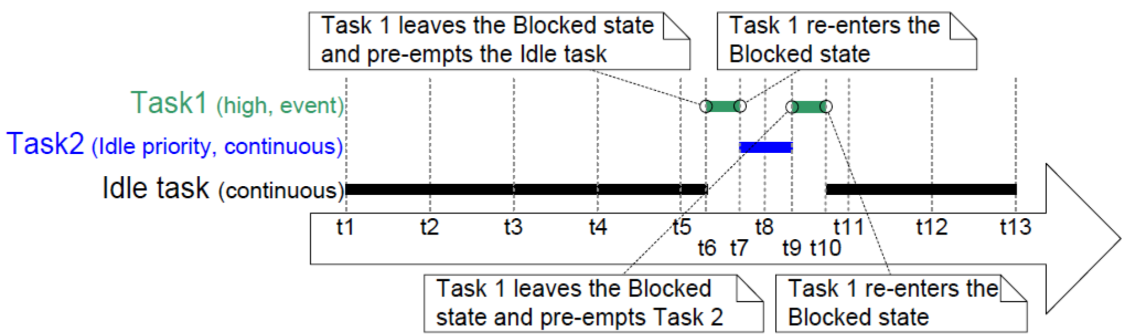


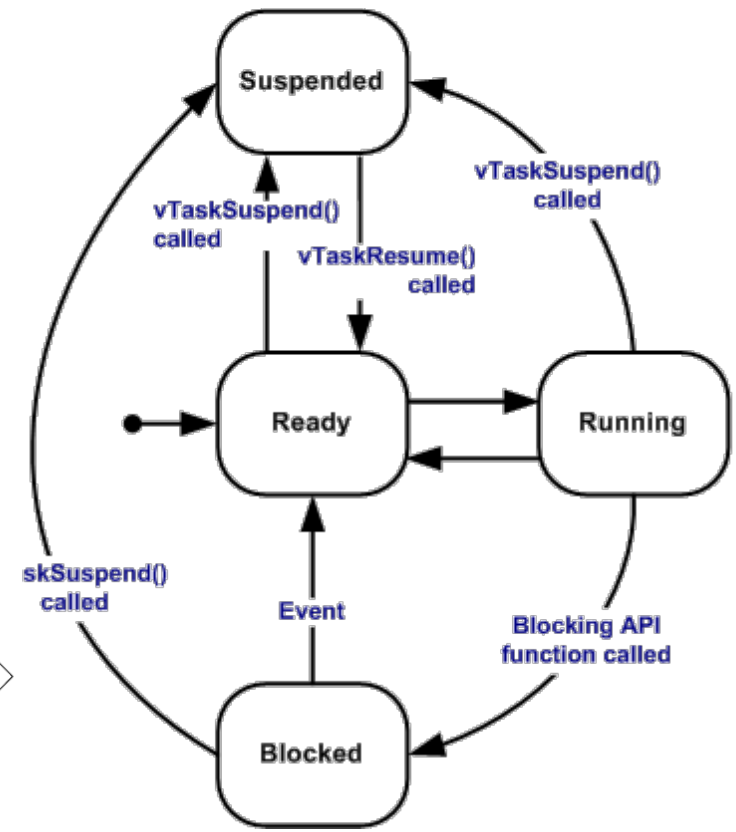
> In time_slicing mode the quantum is being defined by the tick interrupt configTICK_RATE_HZ, by default is 1ms

> configTICK_RATE_MS , convert time delays from ms to number of ticks interrupts

Task Scheduler





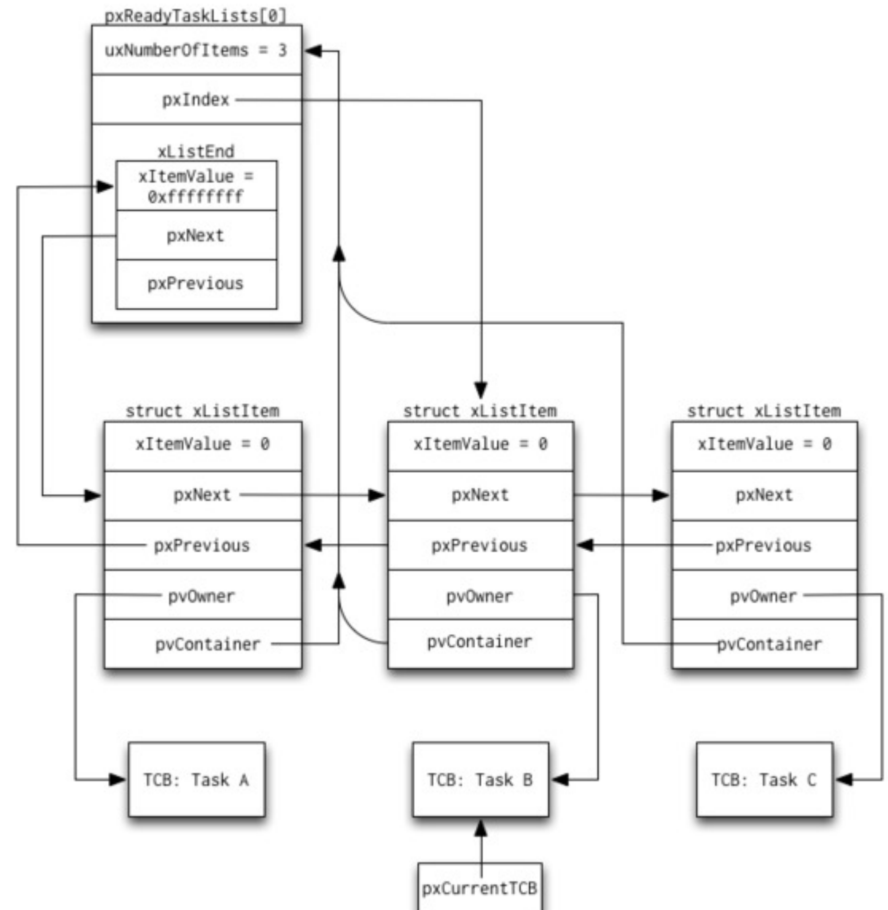


> In time_slicing mode the quantum is being defined by the tick interrupt `configTICK_RATE_HZ`, by default is 1ms

> `configTICK_RATE_MS` , convert time delays from ms to number of ticks interrupts

The ready List queue

It is the most important data structure in FreeRTOS and represents the elegible tasks to be executed.



Tasks – API*

UBaseType_t uxTaskPriorityGet(TaskHandle_t xTask**)** → Obtain the priority of any task.

Example:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
    // ...
    // Use the handle to obtain the priority of the created task.
    // It was created with tskIDLE_PRIORITY, but may have changed
    // it itself.
    if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
    {
        // The task has changed its priority.
    }
    // ...
    // Is our priority higher than the created task?
    if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
    {
        // Our priority (obtained using NULL handle) is higher.
    }
}
```

Tasks – API*

void vTaskPrioritySet(TaskHandle_t xTask, UBaseType_t uxNewPriority) → Set the priority of any task

Example:

```
void vAFunction( void )
{
TaskHandle_t xHandle;
    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
    // ...
    // Use the handle to raise the priority of the created task.
    vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 )
    // ...
    // Use a NULL handle to raise our priority to the same value.
    vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```


* API Reference >> https://www.freertos.org/a00106.html

Tasks – API*

void vTaskSuspend(TaskHandle_t xTaskToSuspend**)** → Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Example:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
    // ...
    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );
    // ...
    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).
    //...
    // Suspend ourselves.
    vTaskSuspend( NULL );
    // We cannot get here unless another task calls vTaskResume
    // with our handle as the parameter.
}
```


* API Reference >> https://www.freertos.org/a00106.html

Tasks – API*

void vTaskResume (TaskHandle_t xTaskToResume**)** → Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Example:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
    // ...
    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );
    // ...
    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).
    //...
    // Resume the suspended task ourselves.
    vTaskResume( xHandle );
    // The created task will once again get microcontroller processing
    // time in accordance with its priority within the system.
}
```


* API Reference >> https://www.freertos.org/a00106.html

Tasks – API*

vTaskDelay(portTickType xTicksToDelay**)** → Number of **ticks** to remain blocked

vTaskDelayUntil(portTickType *pxPreviousWakeTime, portTickType xTimeIncrement**)**

→ Specify the exact tick count value to move the task from blocked to ready state

→ *pxPreviousWakeTime is obtained at the moment the function is executed

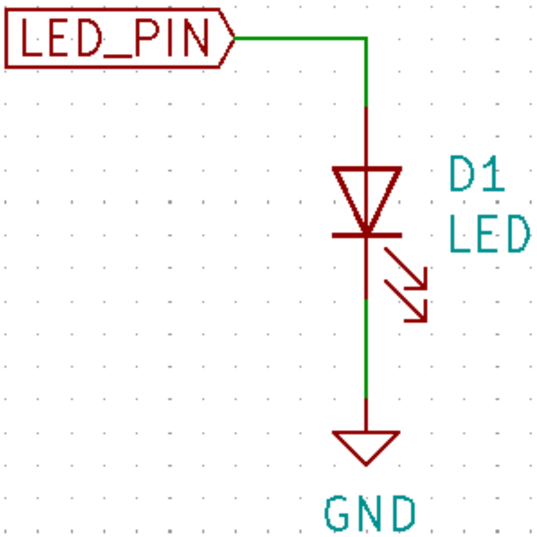
Example:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Block for 10ms
        vTaskDelay(xFrequency);

        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```


* API Reference >> https://www.freertos.org/a00106.html

Running example – LED Blink

```
#define LED_PIN 10
#define ARDUINO_RUNNING_CORE 0

void TaskBlink( void *pvParameters );

void setup() {

  xTaskCreatePinnedToCore(
    TaskBlink       // Associated function
    ,  "TaskBlink"  // Assigned label
    ,  1024         // Stack size
    ,  NULL
    ,  2            // Priority
    ,  NULL
    ,  0);          //  ARDUINO_RUNNING_CORE

}

void loop()
{
  // Empty. Things are done in Tasks.
}

void TaskBlink(void *pvParameters)
{
  (void) pvParameters;

  pinMode(LED_PIN, OUTPUT);

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(100);
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(100);
  }
}
```



Activity 1

1. Implement another Task controlling a LED in `GPIO_10` with 500ms cycle period.

2. Modify the Priority of the new Task to a `higher priority`. When running again, has changed the behaviour in any way? Why?

3. Exchange the code in the new Task with the code present in the box below maintaining the `higher priority`. Observe the behaviour differences.

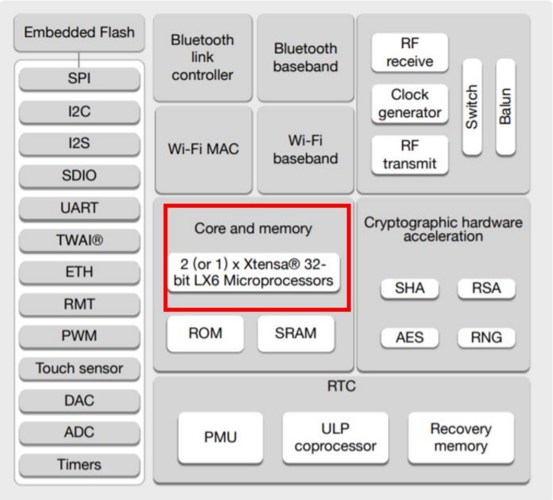
4. Is there any difference if the Thread returns to its original priority?

```
void TaskBlink2(void *pvParameters)
{
  (void) pvParameters;

  pinMode(10, OUTPUT);
  long start_time;

  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    start_time = xTaskGetTickCount();
    while ((xTaskGetTickCount()-start_time)<500);
    start_time = xTaskGetTickCount();
    digitalWrite(LED_BUILTIN, LOW);
    while ((xTaskGetTickCount()-start_time)<500);
  }
}
```

Dual Core – ESP-IDF

- ESP32 is a 32 bit dual-core chip*
- The scheduler assigns the core for each task, by default it is core 1
- Task creation adds a new parameter determining the assigned core

  >> xTaskCreatePinnedToCore




```
xTaskCreatePinnedToCore(Task2code,"Task2",10000,NULL,1,&Task2,1);
```

- It is possible to know the task assigned core:

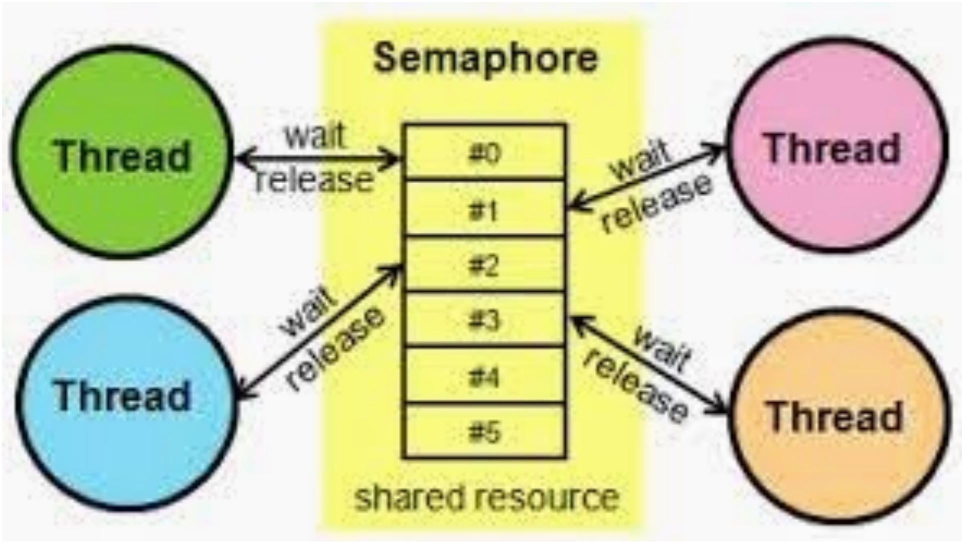
  >> xPortGetCoreID()

```
void Task2code( void * pvParameters ){
 Serial.print("Task2 running on core ");
 Serial.println(xPortGetCoreID());

     for(;;){
       digitalWrite(led_2, HIGH);
       delay(1000);
       digitalWrite(led_2, LOW);
       delay(1000);
     }
}
```

* There are some ESP32 variants with just only one core
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

Semaphores

- They are really useful in RTOS as ensure controlled accesses to the resources
- There are two implementations; **Counter** and **Binary** semaphores
- Multiple tasks can use binary sempahores but only one task can acquire it at a time.
- Usage:
  - Task synchronization – Binary access
  - Control the number of concurrent access to resources - Count semaphore



Binary Semaphore

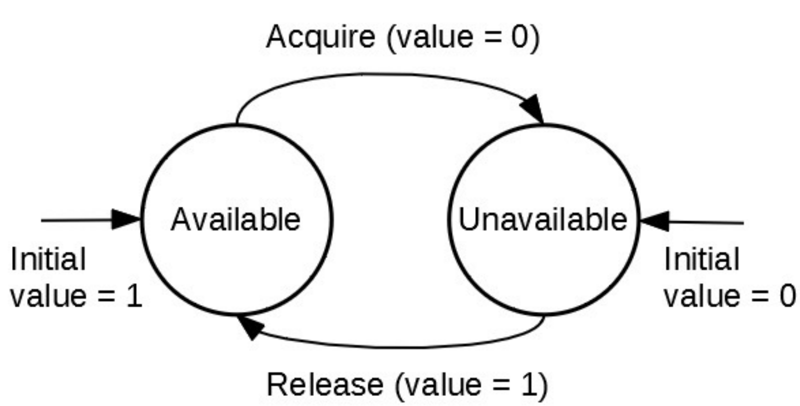
Just only one task is allowed to access

```
SemaphoreHandle_t xSemaphoreCreateBinary(void)
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,
                          TickType_t xTicksToWait);
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

> Just in case the semaphore should be released from an interrupt service

```
BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
                                 BaseType_t *pxHigherPriorityTaskWoken)
```

HigherPriorityTaskWoken > ensures that the interrupt returns directly to the highest priority Ready state task



Binary Semaphores

```
#define  LED  13
SemaphoreHandle_t xBinarySemaphore;
void setup()
{
  Serial.begin(115200);
  pinMode(LED ,OUTPUT);
  xBinarySemaphore = xSemaphoreCreateBinary();
  xTaskCreate(LedOnTask, "LedON",1000,NULL,1,NULL);
  xTaskCreate(LedoffTask, "LedOFF", 1000,NULL,1,NULL);
  xSemaphoreGive(xBinarySemaphore);
}

void loop(){}
void LedOnTask(void *pvParameters)
{
  while(1)
  {
   if(xSemaphoreTake(xBinarySemaphore,portMAX_DELAY)==pdTRUE){
    Serial.println("Inside LedOnTask");
    digitalWrite(LED,LOW);
    xSemaphoreGive(xBinarySemaphore);
   }
   vTaskDelay(500);
  }
}
void LedoffTask(void *pvParameters)
{
  while(1)
  {
   if(xSemaphoreTake(xBinarySemaphore,portMAX_DELAY)==pdTRUE){
    Serial.println("Inside LedffTask");
    digitalWrite(LED,HIGH);
    xSemaphoreGive(xBinarySemaphore);
   }
   vTaskDelay(500);
  }
}
```

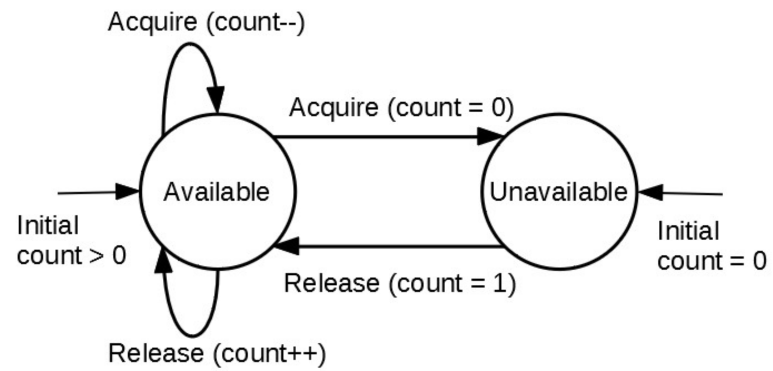
Counter Semaphore

There is a limited number of tasks allowed to access simultaenously to the resource

Usage:

- Counting events: An event increment the semaphore count value, and a handler task decrementing it. The count determint how many events remain unprocessed.
- Resource management: The count value indicates the number of resources available.

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount,
                                           UBaseType_t uxInitialCount)
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,
                          TickType_t xTicksToWait)
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore)
UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore)
```



Counter Semaphores

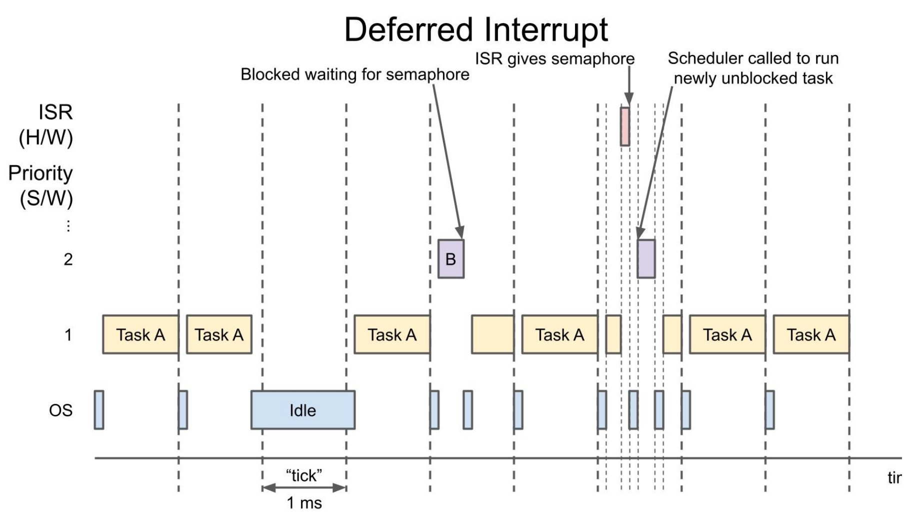
```
#define  LED  13
SemaphoreHandle_t xCountingSemaphore;
void setup()
{
  Serial.begin(115200); // Enable serial communication library.
  pinMode(LED,OUTPUT);

  xCountingSemaphore = xSemaphoreCreateCounting(1,1);
  xSemaphoreGive(xCountingSemaphore);

  xTaskCreate(Task1,"Ledon",1000,NULL,0,NULL );
  xTaskCreate(Task2,"Ledoff",1000,NULL,0,NULL);

}

void loop() {}
void Task1(void *pvParameters)
{
  while(1){
    xSemaphoreTake(xCountingSemaphore, portMAX_DELAY);
    Serial.println("Inside Task1 and Serial monitor Resource Taken");
    digitalWrite(LED,HIGH);
    xSemaphoreGive(xCountingSemaphore);
    vTaskDelay(500);
  }
}
void Task2(void *pvParameters)
{
  while(1){
    xSemaphoreTake(xCountingSemaphore, portMAX_DELAY);
    Serial.println("Inside Task2 and Serial monitor Resource Taken");
    digitalWrite(LED,LOW);
    xSemaphoreGive(xCountingSemaphore);
    vTaskDelay(500);
  }
}
```

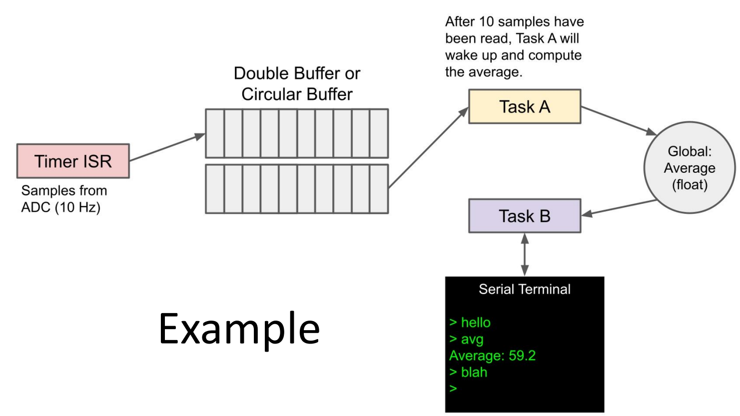
Interrupt Management

Interrupts are events that occur asynchronously and notify the CPU that it should take some action. The CPU stops whatever it was doing and execute the “interrupt service routine” (ISR) that uses queues, mutexes, and semaphores to warn tasks for doing something.

Some considerations:

1. An ISR should never block itself.
2. ISR must be as short as possible.
3. If a variable (such as a global) is updated inside an ISR, you likely need to declare it with the “volatile” qualifier. This lets the compiler know that the “volatile” variable can change outside the current thread of execution.





Example

Interrupt Management

>> **attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)**

> digitalPinToInterrupt(pin): Associated pin that will cause an interrupt.

> ISR: Function name that will be executed . It has not parameters and also returns nothing. Whenever the interrupt will occur this function will be called.

> mode: The triggering action for the interrupt to occur.

LOW: This is used to trigger the interrupt when the pin is in a low state.

CHANGE: This is used to trigger the interrupt when the pin changes its state (HIGH-LOW or LOW-HIGH)

RISING: This is used to trigger the interrupt when the pin goes from LOW to HIGH

```
#define pushButton_pin   15
#define LED_pin   22

void IRAM_ATTR toggleLED()
{
  digitalWrite(LED_pin, !digitalRead(LED_pin));
}

void setup()
{
  pinMode(LED_pin, OUTPUT);
  pinMode(pushButton_pin, INPUT);
  attachInterrupt(pushButton_pin, toggleLED, RISING);
}

void loop(){}
```