ESP32, Espressif IoT Development Framework and FreeRTOS

FreeRTOS - introduction

Real-Time systems are designed to do something within a certain amount of time; they guarantee that stuff happens when it's supposed to.

Embedded systems are computer systems that are designed to do only a few things and which are typically smaller and slower than general purpose computer systems.

FreeRTOS is an open source Real-Time Operating System (RTOS) for embedded systems, which supports many different architectures and compiler *toolchains*, and is designed to be "small, simple, and easy to use".

FreeRTOS is an essential part of the Espressif IoT Development Framework API.

The ESP32 is **dual core** containing a Protocol CPU (known as CPU 0 or PRO_CPU) and an Application CPU (known as CPU 1 or APP_CPU). The two cores are identical in practice and share the same memory. This allows the two cores to run tasks interchangeably between them.

The ESP-IDF FreeRTOS is a modified version of vanilla FreeRTOS which supports symmetric multiprocessing (SMP) on the two cores of ESP32.

FreeRTOS - architecture

Like all Operating Systems, FreeRTOS's main job is to run tasks: most of FreeRTOS's code involves prioritizing, scheduling, and running user-defined tasks.

FreeRTOS is a relatively small application: the minimum core of FreeRTOS consist just of few source (.c) and header (.h) files. FreeRTOS's code breaks down into following main areas:

- **tasks**: almost half of FreeRTOS's core code deals with the central concern of managing tasks. A task is a user-defined C function with a given priority. *tasks.c* and *task.h* do all the heavy lifting for creating, scheduling, and maintaining tasks;

- **communication**: about 40% of FreeRTOS's core code deals with communication. *queue.c* and *queue.h* handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes;

- **hardware**: FreeRTOS core is hardware-independent and sits on top of a hardware-dependent layer, which knows how to talk to whatever supported chip architecture.

Tasks – definition

Tasks are implemented as C functions; the only thing special about them is their prototype:

```
void ATaskFunction( void *pvParameters );
```

Each task is a small program in its own right:

- it has an entry point;
- it will normally run forever within an infinite loop;
- it will not exit → a task must not contain a 'return' statement: if it is no longer required, it should instead be explicitly deleted.

A single task function definition can be used to create any number of tasks — each created task being a separate execution instance, with its own stack and its own copy of any automatic (stack) variables defined within the task itself.

Tasks – definition

The typical structure of a task function is:

```
void taskFunction (void *pvParameters) {

    int32_t lVariableExample = 0;

    for( ; ; ) {
        /* here the code to implement task functionality */
    }

    vTaskDelete(NULL);
}
```

variables can be declared just as per a normal function. Each instance of a task created using this example function will have its own copy of the `lVariableExample` variable. This would not be true if the variable was declared **static** – in which case only one copy of the variable would exist, and this copy would be shared by each created instance of the task.

a task will normally be implemented as an infinite loop

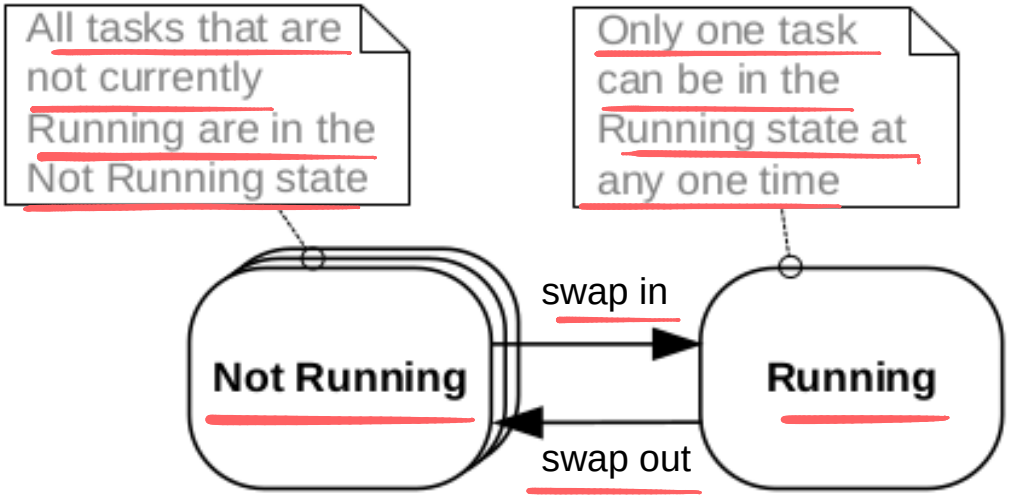
should the task implementation ever break out of the infinite loop, then the task must be deleted before reaching the end of its implementing function. The NULL parameter passed to the vTaskDelete() API function indicates that the task to be deleted is the calling (this) task

Tasks – a simple model

An application can consist of many tasks ⇒ if the processor running the application contains a single (or few) core(s), then only one (few) task(s) can be executing at any given time.

In a simplistic model, this implies that a task can exist in one of two states:

- Running;
- Not Running;



→ the FreeRTOS scheduler is the only entity that can switch a task in and out.

Tasks – creation

Tasks are created using the **xTaskCreate()** function, which create a new task and add it to the list of tasks that are ready to run:

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

pvTaskCode is a pointer to the function that implements the task (actually, it is just the function's name)

pcName is just a descriptive name for the task

each task has its own unique stack that is allocated by the kernel to the task when it is created: the **usStackDepth** value tells the kernel how large to make the stack, in terms of bytes

WORD = 2 BYTE

the value assigned to **pvParameters** is the value passed into the task

uxPriority defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.

pxCreatedTask can be used to pass out a handle to the task being created: this handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task.

Tasks – creation

The xTaskCreate() function has two possible return values:

- **pdPASS** - this indicates that the task has been created successfully;
- **pdFAIL** - this indicates that the task has not been created because there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory: the first block is used to hold the task's data structures, the second block is used by the task as its stack.

If a task is created using xTaskCreate() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function.

Tasks – creation

Since in ESP-IDF FreeRTOS, tasks are designed to run on a particular core, new task creation functions have been added to ESP-IDF FreeRTOS implementation.

The function **xTaskCreatePinnedToCore**() is similar to xTaskCreate(), but allows setting task affinity in SMP system.

xTaskCreatePinnedToCore() is nearly identical to xTaskCreate() with the exception of the extra last parameter known as **xCoreID**. This parameter specifies the core on which the task should run on and can be one of the following values:

- 0 pins the task to PRO_CPU;
- 1 pins the task to APP_CPU;
- tskNO_AFFINITY allows the task to be run on both CPUs.

For example `xTaskCreatePinnedToCore(tsk_callback, "APP_CPU Task", 1000, NULL, 10, NULL, 1)` creates a task of priority 10 that is pinned to APP_CPU with a stack size of 1000 bytes.

Actually, xTaskCreate() has been defined in ESP-IDF FreeRTOS as inline function which call xTaskCreatePinnedToCore() with tskNO_AFFINITY as the xCoreID value.

Tasks – creation - example 1

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void vTaskFunction1(void *parameter);
void vTaskFunction2(void *parameter);

void app_main() {
   xTaskCreatePinnedToCore(vTaskFunction1, "Task1", 2048, NULL, 1, NULL, 1);
   xTaskCreatePinnedToCore(vTaskFunction2, "Task2", 2048, NULL, 1, NULL, 1);
}

void vTaskFunction1(void *parameter) {
    volatile uint32_t c;
    const char * task1Message = "Task1\n";
    while(1) {
        printf("%s", task1Message);
        for(c = 500000; c > 0; c--);
    }
}

void vTaskFunction2(void *parameter) {
    volatile uint32_t c;
    const char * task1Message = "Task1\n";
    while(1) {
        printf("%s", task2Message);
        for(c = 500000; c > 0; c--);
    }
}
```

app_main() is the program's entry point: tasks are created and scheduled here

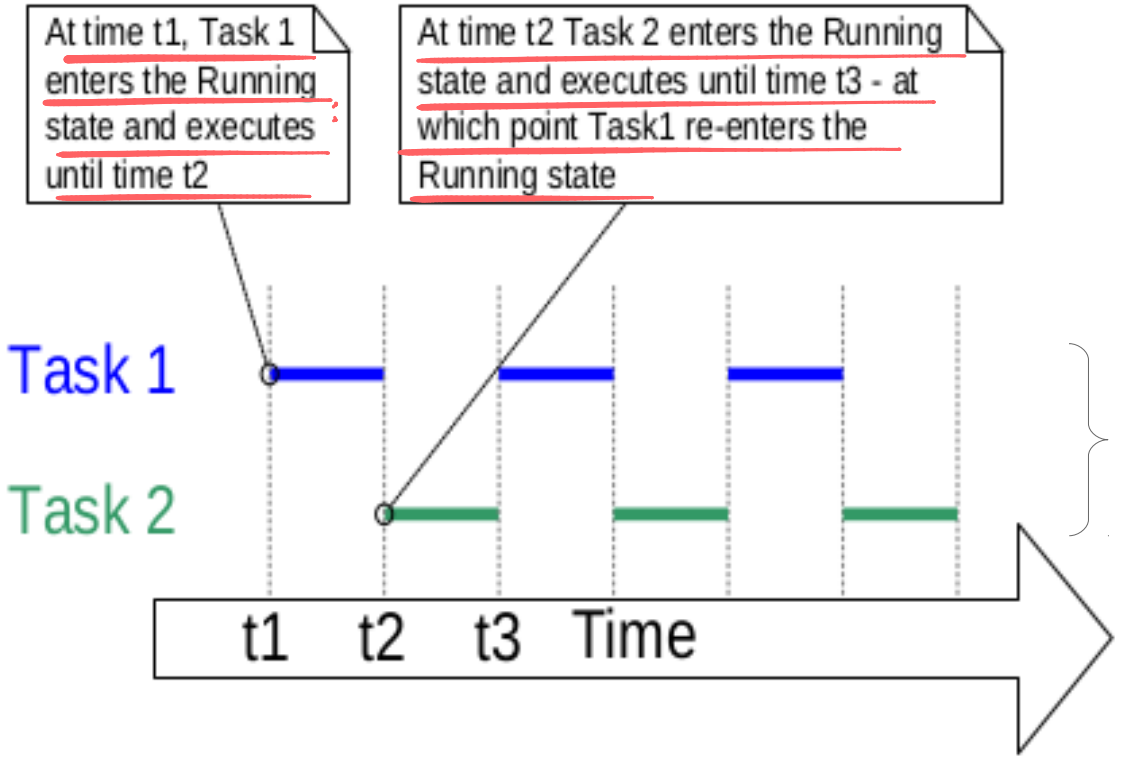
vTaskFunction1 and **vTaskFunction2** are similar task functions

dummy delay implementation

Tasks – creation - example 1

output:

```
--- Miniterm on /dev/ttyUSB0  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T
Parameter: Task1
Parameter: Task2
Parameter: Task1
Parameter: Task2
Parameter: Task1
Parameter: Task2
Parameter: Task1
Parameter: Task2
```



Task1 and **Task2** have the same priority, and both tasks were always able to run. Therefore, each task is executed for a 'time slice', entering the Running state at the start of a time slice, and exiting the Running state at the end of a time slice

Tasks – creation - example 2

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

const char *pcTask1 = "Task1\n";
const char *pcTask2 = "Task2\n";

void vTaskFunction(void *parameter);

void app_main() {
 xTaskCreatePinnedToCore(vTaskFunction, "Task1", 2048, (void *) pcTask1, 1, 1);
 xTaskCreatePinnedToCore(vTaskFunction, "Task2", 2048, (void *) pcTask2, 1, 1);
}

void vTaskFunction(void *parameter)
{
    volatile uint32_t c;
    char *pcTaskName;
    pcTaskName = (char *) parameter;
    while(1) {
        printf("Parameter: %s", pcTaskName);
        for(c = 500000; c > 0; c--);
    }
}
```

two different task instances of the same function are created passing it different parameter's values

FreeRTOS – tick period

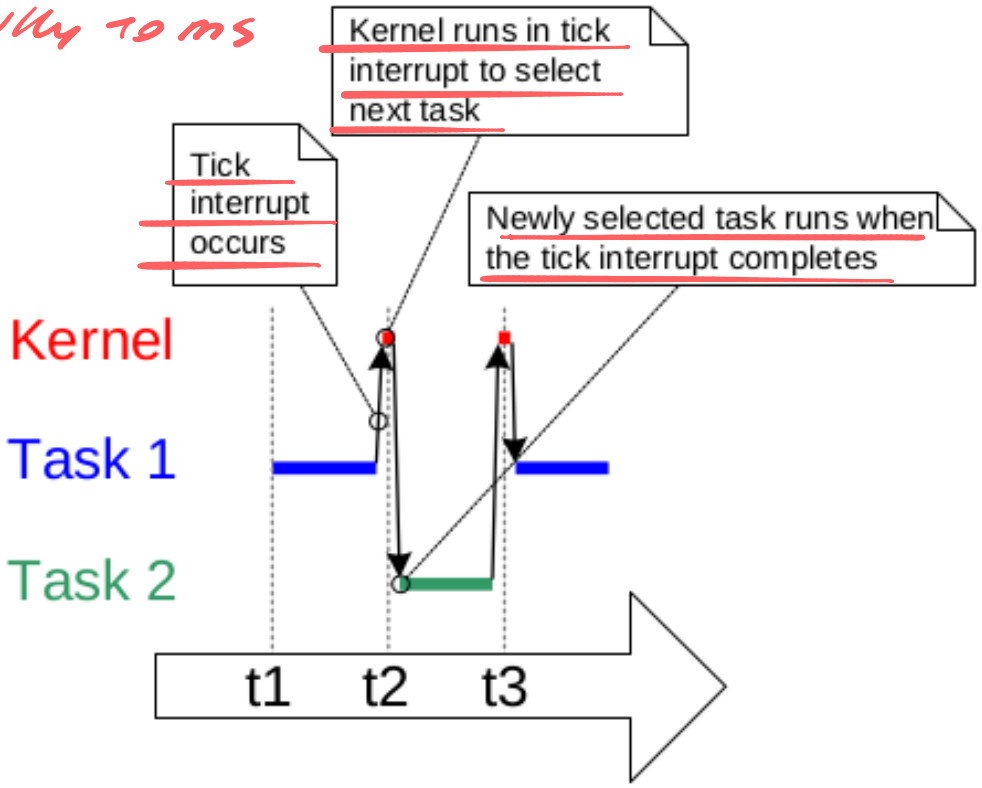
In previous examples, the FreeRTOS scheduler manages two tasks with the same priority, running each one for a time slice.

usually 10 ms

To be able to select the next task to run, the scheduler itself must execute at the end of each time slice. A periodic interrupt, called the '**tick interrupt**', is used for this purpose.

The length of the time slice is effectively set by the tick interrupt frequency, which is configured by the configuration constant: **CONFIG_FREERTOS_HZ**.

Typically CONFIG_FREERTOS_HZ is set to 100 (Hz), then the time slice will be 10 milliseconds. The time between two tick interrupts is called the '**tick period**': one time slice equals one tick period.



Tasks – creation - example 2a

Now, let's modify example 2 in order to give a higher priority to one of the two tasks (which run on the same core):

both tasks run on APP_CPU

```
void app_main() {
 xTaskCreatePinnedToCore(vTaskFunction, "Task1", 2048, (void *) pcTask1, 1, 1);
 xTaskCreatePinnedToCore(vTaskFunction, "Task2", 2048, (void *) pcTask2, 2, 1);
}
```

Task2 has a higher priority than Task1

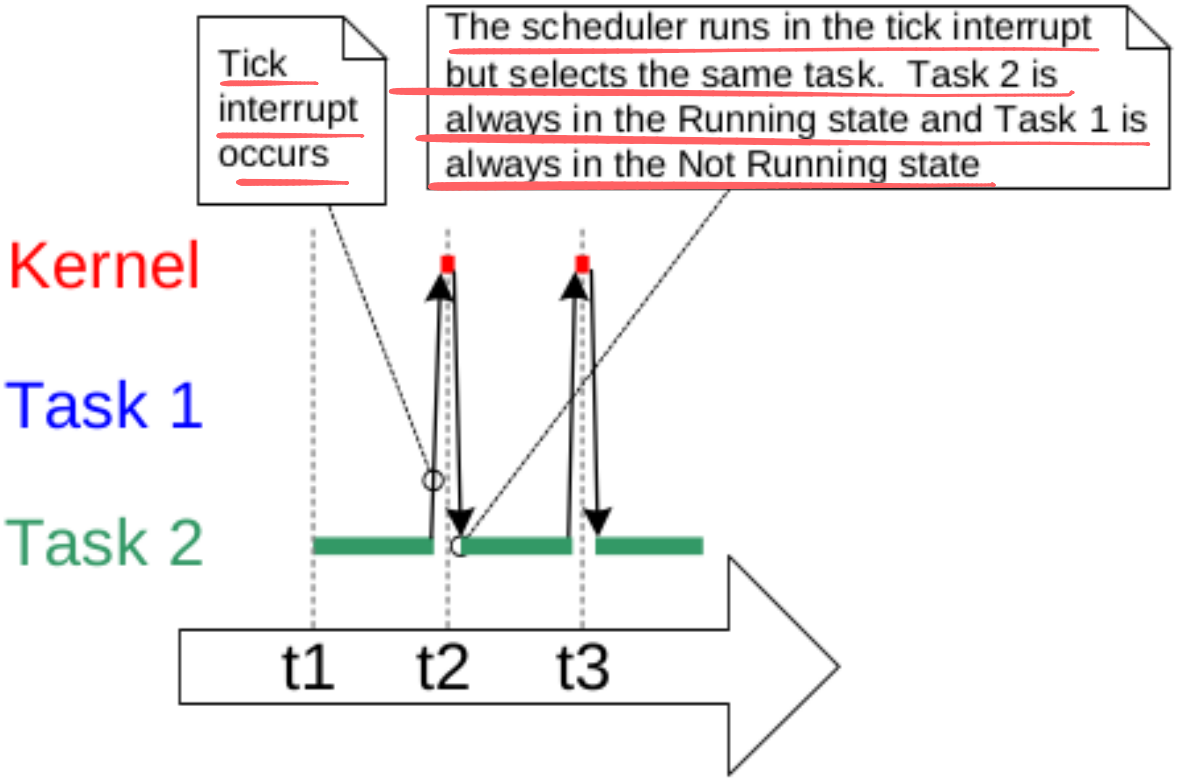
output:

```
--- Miniterm on /dev/ttyUSB0  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T
Parameter: Task2
Parameter: Task2
Parameter: Task2
Parameter: Task2
Parameter: Task2
Parameter: Task2
Parameter: Task2
```

Tasks – creation - example 2a

The reason why in previous example Task2 becomes predominant is that the scheduler (which in this case is also forced to use only one core) always select the highest priority task that is able to run.

Task 2 is always able to run because it never has to wait for anything — it is either cycling around a null loop, or printing to the terminal, so it is the only task to ever enter the Running state and Task1 is said to be '**starved**' of processing time by Task2.



Tasks – a more detailed model

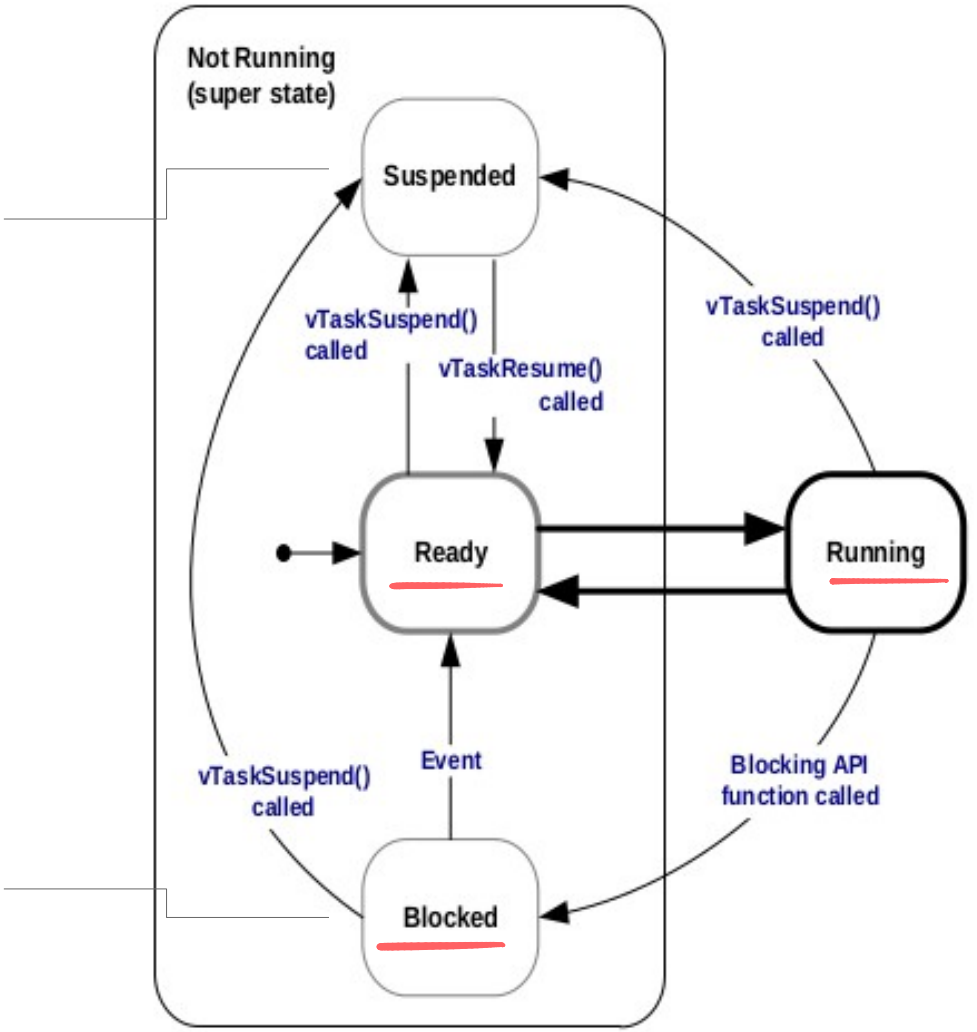
The ‘continuous processing’ tasks like the ones of examples 1 and 2 have limited usefulness, because they can only be created at the very lowest priority.

To make the tasks useful they must be re-written to be event-driven: an **event-driven task** has work (processing) to perform only after the occurrence of the event that triggers it, and is not able to enter the Running state before that event has occurred.

⇒ using event-driven tasks means that tasks can be created at different priorities without the highest priority tasks starving all the lower priority tasks of processing time.

Tasks – a more detailed model

Tasks in the **Suspended** state are not available to the scheduler.
The only way into the Suspended state is through a call to the vTaskSuspend() API function, the only way out being through a call to the vTaskResume() or xTaskResumeFromISR() API functions (most applications do not use the Suspended state).



A task that is waiting for an event is said to be in the '**Blocked**' state. Tasks can enter the Blocked state to wait for two different types of event:

1. **time-related** events - the event being either a delay period expiring, or an absolute time being reached;

2. **synchronization** events - where the events originate from another task or interrupt, for example, a task may enter the Blocked state to wait for data to arrive on a queue;

Tasks – example 2b

Let’s review example 2a, introducing a time-related blocked state in each task:

```
…
void app_main()
{
  xTaskCreatePinnedToCore(vTaskFunction, "Task1", 2048, (void *)pcTask1, 1,
NULL, 1);
  xTaskCreatePinnedToCore(vTaskFunction, "Task2", 2048, (void *)pcTask2, 2,
NULL, 1);
}

void vTaskFunction(void *parameter)
{
  char *pcTaskName;
  pcTaskName = (char *) parameter;
  while(1)
  {
    printf("Parameter: %s", pcTaskName);
    vTaskDelay(1000/portTICK_PERIOD_MS);
  }
}
```

the function **vTaskDelay**() blocks the task for 1[s]: the task does not use any processing time while it is in the Blocked state, so it uses processing time only when there is actually work to be done!

Tasks – example 2b

output:

```
--- Miniterm on /dev/ttyUSB0  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T
Task2
Task1
Task2
Task1
Task2
Task1
Task2
```

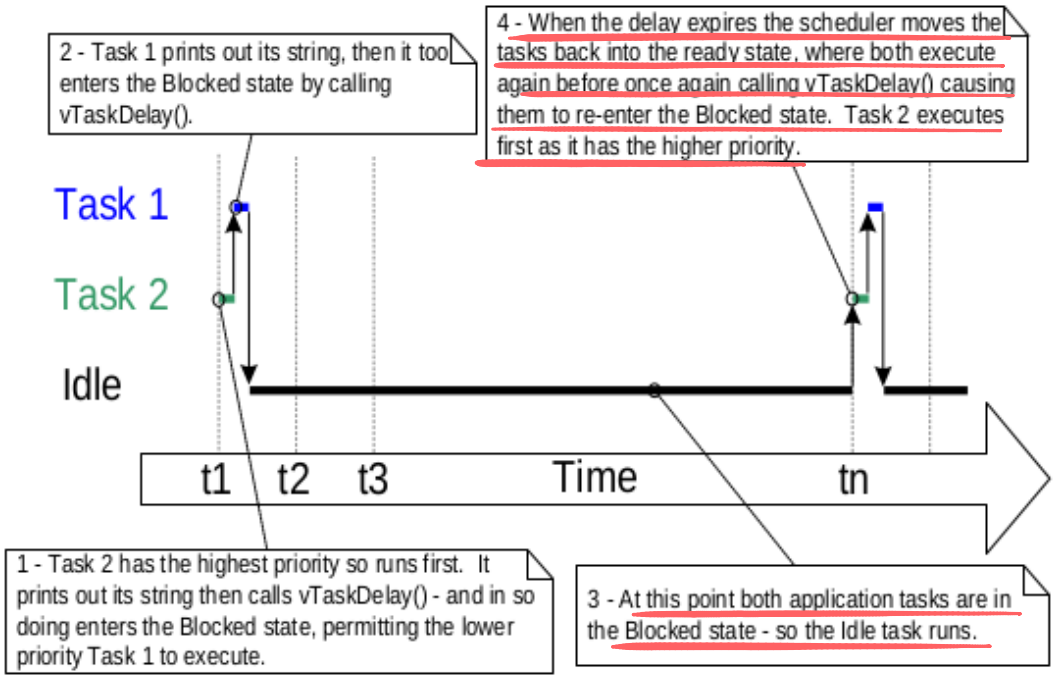
In example 2b, Task2, which has the highest priority, executes first: it prints out a message, then it blocks, so that Task1, at lower priority, can be executed too and it has the same behavior.

Each time one task leaves the Blocked state it runs for a fraction of a tick period before re-entering the Blocked state, so that most of the time there are no tasks that are able to run (no tasks in the Ready state) and, therefore, no tasks can be selected to enter the Running state.

Tasks – idle task

Actually, there must always be at least one task that can enter the Running state: to ensure this is the case, an **Idle task** is automatically created when the scheduler starts.

The idle task does very little more than sit in a loop - so, it is always able to run and it has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.



In example 2b, both tasks run for a fraction of time before staying blocked for many tick periods.

While this is the case, the idle task will run, just waiting to find tasks in the Ready state again.

The amount of processing time allocated to the idle is a measure of the spare processing capacity in the system: using RTOS can significantly increase the spare processing capacity simply by allowing an application to be completely event driven.

Tasks – xTaskDelay()

The function *void* **vTaskDelay**(*const TickType_t xTicksToDelay*), blocks a task for a given number of ticks.

The actual time that the task remains blocked depends on the tick rate. The constant **portTICK_PERIOD_MS** can be used to calculate real time from the tick rate - with the resolution of one tick period.

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called, it is therefore difficult to use vTaskDelay () by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTaskDelay () may not be fixed (the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes).

Tasks – xTaskDelayUntil()

To wake up tasks from blocked state at desired time, the function vTaskDelayUntil() is a better choice.

The function *void* **vTaskDelayUntil**(*TickType_t *const pxPreviousWakeTime, const TickType_t xTimeIncrement*) specifies the absolute (exact) time at which it wishes to unblock, therefore this function can be used by periodic tasks to ensure a constant execution frequency.

```
void vTaskFunction(void *parameter)
{
  char *pcTaskName;
  pcTaskName = (char *) parameter;
  TickType_t xLastWakeTime;

  xLastWakeTime = xTaskGetTickCount ();
  while(1)
  {
   printf("%s", pcTaskName);
   vTaskDelayUntil(&xLastWakeTime,1000/portTICK_PERIOD_MS);
  }
}
```

ESP-IDF FreeRTOS scheduler – in-depth analysis

The FreeRTOS implements scheduling in the `vTaskSwitchContext()` function: this function is responsible for selecting the highest priority task to run from a list of tasks in the Ready state.

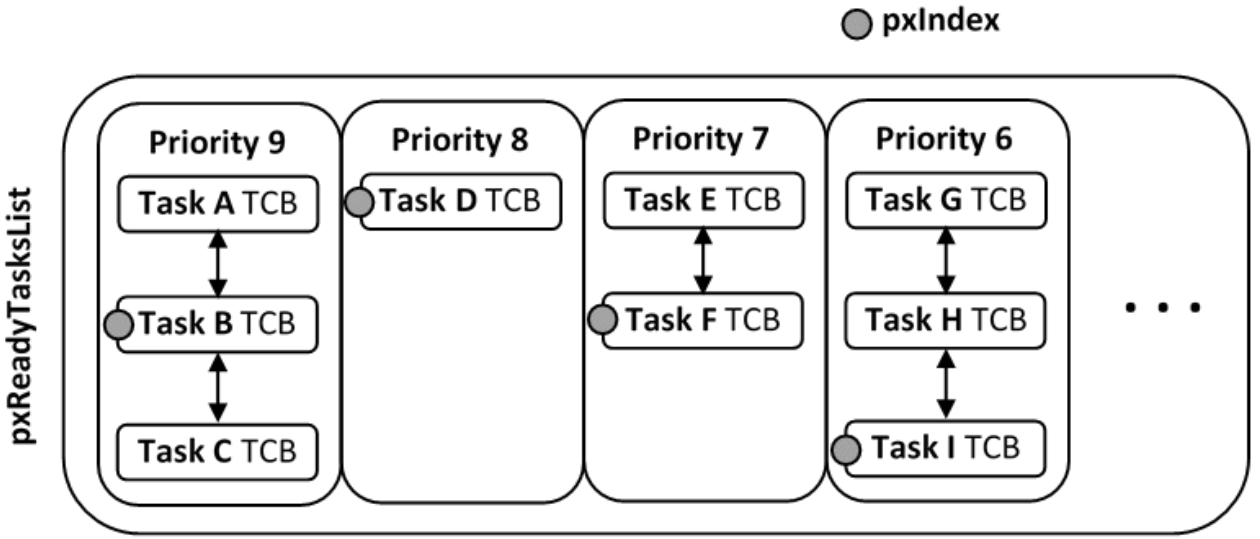
In ESP-IDF FreeRTOS, each core will call `vTaskSwitchContext()` independently to select a task to run from the Ready Tasks List which is shared between both cores.

Given multiple tasks in the Ready state and of the same priority, FreeRTOS implements **Round Robin** scheduling between each task: this will result in running those tasks in turn each time the scheduler is called (e.g. every tick interrupt).

ESP-IDF FreeRTOS scheduler – in-depth analysis

In FreeRTOS, `pxReadyTasksList` is used to store a list of tasks that are in the Ready state.

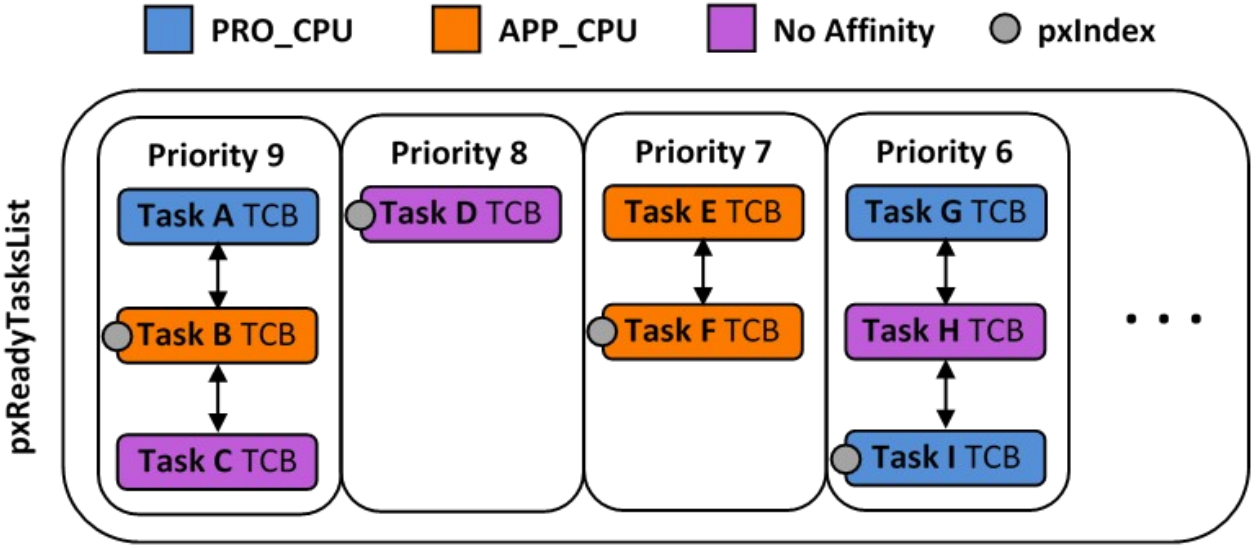
The list is implemented as an array where each element of the array is a linked list which contains TCBs of tasks of the same priority that are in the Ready state.



Each linked list also contains a `pxIndex` which points to the last TCB returned when the list was queried: this index allows the `vTaskSwitchContext()` to start traversing the list at the TCB immediately after `pxIndex` hence implementing Round Robin Scheduling between tasks of the same priority.

ESP-IDF FreeRTOS scheduler – in-depth analysis

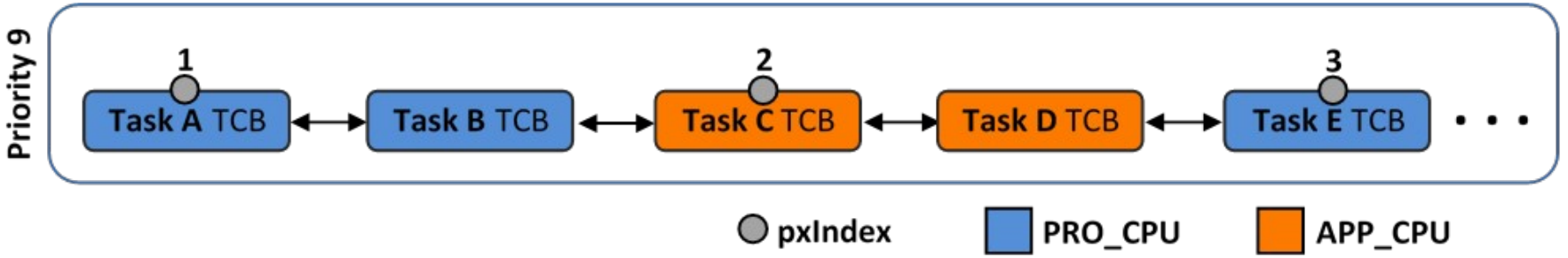
In ESP-IDF FreeRTOS, the Ready Tasks List is shared between cores hence `pxReadyTasksList` will contain tasks pinned to different cores: when a core calls the scheduler, it is able to look at the `xCoreID` member of each TCB in the list to determine if a task is allowed to run on calling the core.



Although each TCB has an `xCoreID` in ESP-IDF FreeRTOS, the linked list of each priority only has a single `pxIndex`: when the scheduler is called from a particular core and traverses the linked list, it will skip all TCBs pinned to the other core and point the `pxIndex` at the selected task.

ESP-IDF FreeRTOS scheduler – in-depth analysis

Therefore, an issue arises, because TCBs skipped on the previous scheduler call from the other core would not be considered on the current scheduler call:



1) **PRO_CPU** calls the scheduler and selects Task A to run, hence moves `pxIndex` to point to Task A;

2) **APP_CPU** calls the scheduler and starts traversing from the task after `pxIndex` which is Task B. However Task B is not pinned to APP_CPU hence it is skipped and Task C is selected instead: `pxIndex` now points to Task C;

3) **PRO_CPU** calls the scheduler and it skips Task D and selects Task E to run. Notice that Task B isn't traversed;

4) The same situation with Task D will occur if **APP_CPU** calls the scheduler again as `pxIndex` now points to Task E.

One solution to the issue of task skipping is to ensure that every task will enter a blocked state so that they are removed from the Ready Task List. Another solution is to distribute tasks across multiple priorities such that a given priority will not be assigned multiple tasks that are pinned to different cores.