

FreeRTOS - Tasks

Tasks are jobs will be done in an application. These special jobs are managed by scheduler with task's stack, priority.

[#arm](#) [#stm32](#) [#rtos](#)

Last update: 2021-08-05 23:53:34

Table of Content

1. A Task
2. Startup
3. The IDLE Task
4. Task Lists
5. Task Functions
6. Task states
7. Task priorities
8. Task context switching
9. Lab 0: Tasks
 - 9.1. Enable UART redirection
 - 9.2. Define a param for tasks
 - 9.3. Add 2 Tasks
 - 9.4. Build and Run
 - 9.5. Change Priority
 - 9.6. Use OS Delay
10. Other Tasks functions

1. A Task

A task will do a specific functionality, such as toggling an LED, reading an input. The task function usually is in infinite loop, it means a task will continuously run and never returns.

The Task Function is declared as:

```
void taskFunctionName(void* argument) {
    for(;;) {
        // do things over and over
    }
}
```

In freeRTOS, every task has its own stack that stores TCB (Task Control Block) and other stack-related operations while the task is being executed. It also stores processor context before a context switch (switching to other task). Stack size must be sufficient to accommodate all local variables and processor context.

2. Startup

1. Main Application sets up the System Clock and necessary Peripherals
2. FreeRTOS is started by the `osKernelStart()` function in the `main.c` file from CMSIS_OS APIs:
 1. This calls to the `vTaskStartScheduler()` function from FreeRTOS APIs
 2. `vTaskStartScheduler()` creates an IDLE task using `xTaskCreate()`, then disables all interrupts by calling `portDISABLE_INTERRUPTS()` to be sure that no tick will happened before or during the call to `xPortStartScheduler()` function from FreeRTOS API
 3. `xPortStartScheduler()` configures the lowest priority level for `SysTick` and `PendSV` interrupts, then it starts the timer that generates the SysTick, enables FPU if presented (e.g. in CortexM4) and starts the first task using the `prvPortStartFirstTask()` function
 4. `prvPortStartFirstTask()` function (usually written in assembler) locates the stack and set `MSP` (used by the OS) to the start of the stack, then enables all interrupts. After this, it triggers software interrupt `SVC 0`
 5. As the result of SVC interrupt, `vPortSVCHandler()` is called
 6. `vPortSVCHandler()` restores the context, loads TCB (Task Block Control) of the first task (the highest priority one) from the Ready list and starts executing this task

3. The IDLE Task

The Idle task is created automatically when the scheduler is started:


- It is the `portTASK_FUNCTION()` function in the `task.c` file
- It performs the following operations (in endless loop):
 1. Check for deleted tasks to clean the memory
 2. Call `taskYIELD()` if not using preemption `configUSE_PREEMPTION = 0`
 3. Call `taskYIELD()` if there is another task waiting and `configIDLE_SHOULD_YIELD = 1` to not waste time
 4. Executes `vApplicationIdleHook()` if `configUSE_IDLE_HOOK = 1`
 5. Perform low power entrance if `configUSE_TICKLESS_IDLE != 0`

The default task

When using STM32CubeIDE to add FreeRTOS to project, there is a *default task* which is marked as *can be modified, not be removed*. This not the IDLE task mentioned above. User can configure it as a normal task, or even remove it in the main source code.

4. Task Lists

Task List	Description
<code>ReadyTasksList[]</code>	Prioritized ready tasks lists separate for each task priority up to <code>configMAX_PRIORITIES</code>
<code>TasksWaitingTermination</code>	List of tasks which have been deleted but their memory pools are not freed yet.
<code>SuspendedTaskList</code>	List of tasks currently suspended
<code>PendingReadyTaskList</code>	Lists of tasks that have been read while the scheduler was suspended
<code>DelayedTaskList</code>	List of delayed tasks
<code>OverflowDelayedTaskList</code>	List of delayed tasks which have overflowed the current tick count

 There is no dedicated list for task in Running mode (as there is only one task in this state at the moment), but the currently run task ID is stored in variable `pxCurrentTCB`.

5. Task Functions

A task will do a specific functionality, such as toggling an LED, reading an input. The task function usually is in infinite loop, it means a task will continuously run and never returns. The Task Function is declared as:

```
void taskFunctionName(void* argument) {
    // do something first
    for(;;) {
        // do things over and over
    }
}
```

In freeRTOS, every task has its own stack that stores TCB (Task Control Block) and other stack-related operations while the task is being executed. It also stores processor context before a context switch (switching to other task). Stack size must be sufficient to accommodate all local variables and processor context.

A Task is created by calling `osThreadNew()` which indeed calls to:

- `xTaskCreateStatic()` if `configSUPPORT_STATIC_ALLOCATION == 1`, or calls to
- `xTaskCreate()` if `configSUPPORT_DYNAMIC_ALLOCATION == 1`.

The argument passed to the Task Function is declared as a `void*` pointer. This help to pass any type of data into the function handler.

6. Task states

Ready

Task is ready to be executed but is not currently executing because a different task with equal or higher priority is running

Running

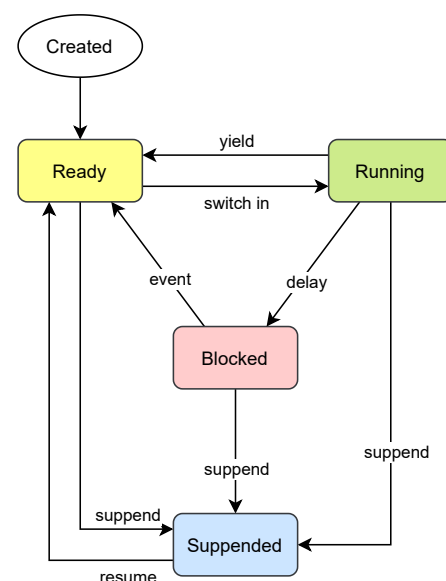
Task is actually running (only one can be in this state at the moment)

Blocked

Task is waiting for either a temporal or an external event

Suspended

Task not available for scheduling, but still being kept in memory



Task states

8. Task context switching

The process of saving the context of a task that is being suspended and restoring the context of a task being resumed is called context switching.

There are two 2 triggers that cause context switching:

1. **SysTick interrupt** This interrupt is used as a periodical signal to check if system needs to switch to another task.

- The interrupt causes `xPortSysTickHandler()` run
- `xPortSysTickHandler()` will:
 - blocks all interrupts (as its own priority is the lowest one) using `portDISABLE_INTERRUPTS()`
 - activates the PendSV bit to trigger `xPortPendSVHandler()` :
 - calls to `vTaskSwitchContext()` function which selects the highest priority task in the READY List using the macro `taskSELECT_HIGHEST_PRIORITY_TASK()`
 - unblocks all interrupts using `portENABLE_INTERRUPT()`

2. **Task yield** When a task has done its work, it can trigger a context switching by send a yield by calling to `portYIELD()` which will set the PendSV bit to trigger `xPortPendSVHandler()` and start the scheduler (see above sequence).

Here are some cases that a task yield is called:

- The idle task is done its internal task, it calls to `taskYIELD()`
- When a task is in the block state, such as it has a delay or delayUntil function call, using `portYIELD_WITHIN_API()`
- When an interrupt unblocks a task using `portYIELD_FROM_ISR()`

9. Lab 0: Tasks

Star a new project with FreeRTOS included in STM32CubeIDE. Refer to the [Lab: Overview](#).

At this time, tasks will be added manually in code, without using the code generation in CubeMX. This project can be targeted on any MCU which has enough RAM to hold FreeRTOS.

Here are the settings needed for the lab:

1. HAL Timebase is assigned to a general timer, let FreeRTOS use the SysTick
2. UART1 is enabled for printing debug information, note the pinout.
3. Select CMSIS_OS version 2, and use default FreeRTOS settings

4. Enable re-entrant settings for using FreeRTOS with `newlib`. Read more in [FreeRTOS newlib reentrant](#).

9.1. Enable UART redirection

It is useful to print out debug information, and to make it easier, [UART redirection](#) will be used. In this lab, it can be implemented in a minimal method by overriding the low-level `_write` function in the main file. Inside this function, a blocking call will be used to make sure all data is written in multi-thread. The correct method of synchronization will be discussed later.

main.c

```
int _write(int file, char *ptr, int len) {
    // block write to UART if it is not ready
    while(HAL_UART_GetState(&huart1) != HAL_UART_STATE_READY);

    HAL_StatusTypeDef hstatus =
        HAL_UART_Transmit(&huart1, (uint8_t*) ptr, len, HAL_MAX_DELAY);

    return (hstatus == HAL_OK ? len : 0);
}
```

9.2. Define a param for tasks

In this example, two tasks will count characters in a range which is defined in a structure as below:

```
typedef struct {
    uint8_t start;
    uint8_t end;
} CounterRange_t;
```

The first task will count from `0` to `9`, while the second tasks will count from `A` to `J`.

```
CounterRange_t Task1_CounterRange = {'0', '9'};
CounterRange_t Task2_CounterRange = {'A', 'J'};
```

9.3. Add 2 Tasks

Define two tasks in the main file. At this moment, two tasks will have the same priority as `osPriorityNormal` level.

main.c

```
osThreadId_t Task1_Handle;
const osThreadAttr_t Task1_attributes = {
    .name = "Task1",
    .stack_size = 128 * 4,
```



```

    .priority = (osPriority_t) osPriorityNormal,
};

osThreadId_t Task2_Handle;
const osThreadAttr_t Task2_attributes = {
    .name = "Task2",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};

```

These two tasks will print out a sequence of character by using the same task `CounterTask()` which gets a `CounterRange_t` param from its task function's parameter:

⚠ HAL_Delay()

At the beginning of this lab, `HAL_Delay()` function is used to simulate a heavy work which is consuming CPU. Scheduler will never put these tasks to the Blocked state.

```

void CounterTask(void *argument) {
    char* name = pcTaskGetName(NULL);
    CounterRange_t* range = (CounterRange_t*)argument;
    printf("%s: range = %c : %c\r\n", name, range->start, range->end);
    uint8_t counter = range->start;
    /* Infinite loop */
    for(;;) {
        printf("%s: counter = %c\r\n", name, counter++);
        if(counter > range->end) counter = range->start;
        HAL_Delay(500);
    }
}

```

In the main function, create two tasks using the `CounterTask()` function with different delay params. Note that the pointer pointing to the `int` param is converted to a `void*` pointer.

```

int main(void) {
    // Hardware init
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    // FreeRTOS Init
    osKernelInitialize();

    // Add Tasks
    Task1Handle = osThreadNew(CounterTask, (void*) &Task1_CounterRange,
    &Task1_attributes);
    Task2Handle = osThreadNew(CounterTask, (void*) &Task2_CounterRange,
    &Task2_attributes);

    //Start scheduler
}

```

```
osKernelStart();
}
```

9.4. Build and Run

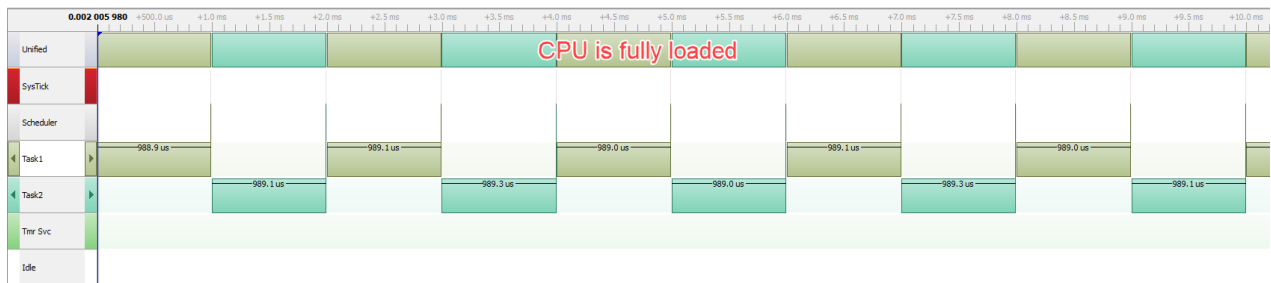
Compile the project and connect the UART1 to PC, and check the result:

- At startup, tasks print out their name and delay to confirm the attributes and params
- Both tasks run and print their counter value
- Task 1 has a smaller delay (500 ms) so it prints out faster than Task 2

```
Task2: range = A : J
Task2: counter = A
Task1: range = 0 : 9
Task1: counter = 0
Task2: counter = B
Task1: counter = 1
Task2: counter = C
Task1: counter = 2
Task2: counter = D
Task1: counter = 3
```

Two tasks are printing

Note that, because two tasks have the same priority, they will be switched at every SysTick interruption (about 1 ms). Both tasks use `HAL_Delay()` which in fact consume



Two tasks have the same priority

9.5. Change Priority

Let change one of two task to use `osPriorityHigh` priority, and see how system works after that modification.

The result is only high priority task can run because higher priority are requesting to use CPU all the time.

```
const osThreadAttr_t Task1_attributes
= {
```

```
const osThreadAttr_t Task2_attributes
= {
```

```
.name = "Task1",
.stack_size = 128 * 4,
.priority = (osPriority_t)
osPriorityHigh,
}
```

```
.name = "Task2",
.stack_size = 128 * 4,
.priority = (osPriority_t)
osPriorityHigh,
}
```

```
Task1: range = 0 : 9
Task1: counter = 0
Task1: counter = 1
Task1: counter = 2
Task1: counter = 3
Task1: counter = 4
Task1: counter = 5
Task1: counter = 6
Task1: counter = 7
Task1: counter = 8
```

Only Task 1 with High Priority is running

```
Task2: range = A : J
Task2: counter = A
Task2: counter = B
Task2: counter = C
Task2: counter = D
Task2: counter = E
Task2: counter = F
Task2: counter = G
Task2: counter = H
Task2: counter = I
```

Only Task 2 with High Priority is running

Change priority in runtime

There are two functions to get and set task priority in runtime:

- The function `osThreadGetPriority()` calls `uxTaskPriorityGet()` or `uxTaskPriorityGetFromISR()` defined in `tasks.c` file
- The function `osThreadSetPriority()` calls `vTaskPrioritySet()` defined in `tasks.c` file. After setting new priority, this function also trigger the task scheduler to re-arrange tasks in the scheduled lists.

Two these functions must be enabled in **Include definitions** settings of FreeRTOS.

9.6. Use OS Delay

Keep one task in high priority, how to run both tasks? The key of RTOS scheduler is whenever a task is not working, it can be moved to block state and other task, even in lower priority, can run.

```
void CounterTask(void *argument) {
    char* name = pcTaskGetName(NULL);
    CounterRange_t* range = (CounterRange_t*)argument;
    printf("%s: range = %c : %c\r\n", name, range->start, range->end);
    uint8_t counter = range->start;
    /* Infinite loop */
    for(;;) {
        printf("%s: counter = %c\r\n", name, counter++);
        if(counter > range->end) counter = range->start;
        osDelay(500);
    }
}
```

[illegible]

1. Calls `vTaskSuspendAll()` to pause the scheduler without disabling interrupts. RTOS tick will be held pending until the scheduler has been resumed.
2. Remove task from event list (running tasks) and move it to delayed list with given delay value using the function `prvAddCurrentTaskToDelayedList()`
3. Resume the scheduler using `xTaskResumeAll()` function
4. Trigger `PendSV` interrupt (using `portYIELD_WITHIN_API()` macro) to switch the context

1. Remove the task from the ready list using `uxListRemove()` and removes the task from waiting on an event tasks list.
2. In case the task is deleting itself, this function will switch execution to the next task calling function by calling `portYIELD_WITHIN_API()`

Memory allocated by the task code is not automatically freed and should be freed before the task is deleted. TCB and its original stack are freed by the IDLE Task.

Task Yield

When a task has done its job, and don't want to wait for a SysTick interruption, it can yield to scheduler to trigger context switching.

