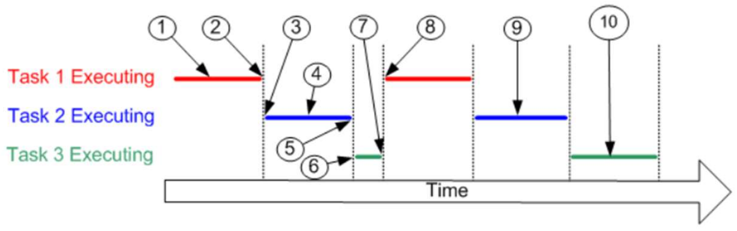
Scheduling

The scheduler is the part of the kernel that is responsible for deciding which task to execute at any particular time. The kernel can pause and resume a task multiple times during the life cycle of the task.

A scheduling strategy is an algorithm used by the scheduler to decide which task to perform at any point in time. A (non-real-time) multi-user system strategy is likely to give each task a "fair" proportion of processor time.

In addition to being involuntarily suspended by the kernel, a task can also choose to suspend itself. If it wants to delay (sleep) for a period of time, or wait (block) for a resource to be available (such as a serial port) or an event (such as a key press), it will do so.



Introduction to Scheduler in FreeRTOS

All operating systems use schedulers to allocate CPU time to tasks or threads. Because only one process can run at a given time on the CPU in the case single-core processor. A low-cost microcontroller comes with a single CPU only and FreeRTOS is specifically designed for resource and memory-constrained microcontrollers. Hence, the scheduler provides the mechanism to decide which task should run on the CPU.

A task can be in one of these four possible states -running, ready, blocked, or suspended. Only tasks that are in a ready state can be picked by a scheduler depending on the scheduling policy. Furthermore, tasks that are in a blocked state (waiting for events such as mutex or semaphores) or in a suspended state cannot be scheduled.

Scheduling Policy

A scheduling policy is an algorithm used by a scheduler to decide which task to run on a processor. Different types of scheduling algorithms are used in real-time operating systems. Mainly the scheduling algorithms are of two types time-slicing and pre-emptive.

FreeRTOS Scheduling Policy

FreeRTOS kernel supports two types of scheduling policy:

Time Slicing Scheduling Policy: This is also known as a round-robin algorithm. In this algorithm, all equal priority tasks get CPU in equal portions of CPU time.

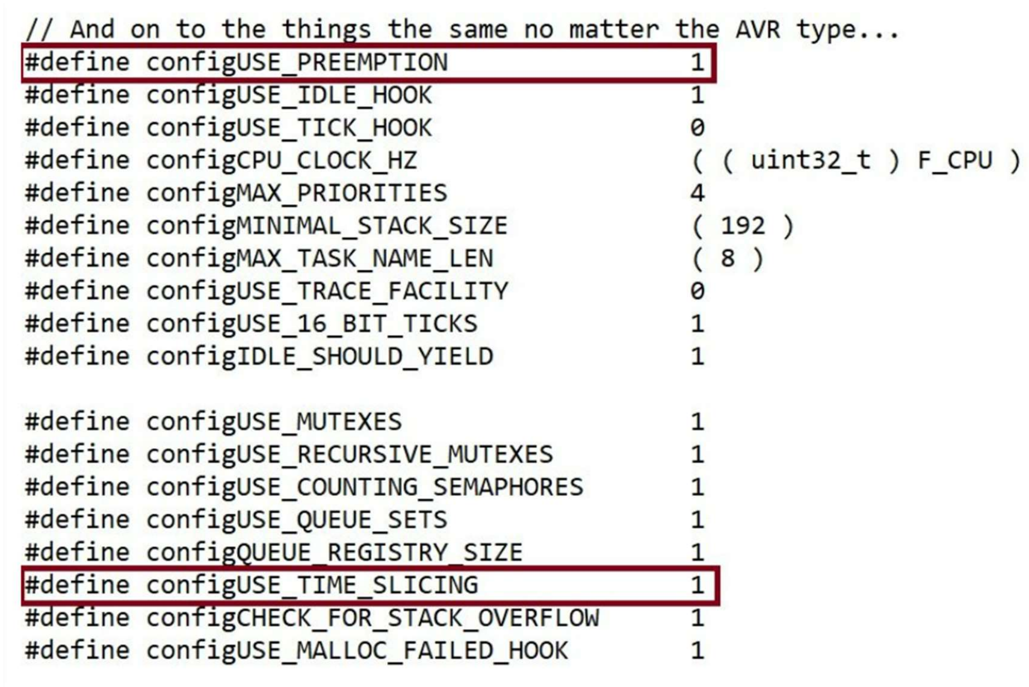
Fixed Priority Pre-emptive Scheduling: This scheduling algorithm selects tasks according to their priority. In other words, a high-priority task always gets the CPU before a low-priority task. A low-priority task gets to execute only when there is no high-priority task in the ready state.

Configure FreeRTOS Scheduler

But in FreeRTOS, we usually use a scheduling policy by mixing both the above-mentioned algorithms and it is known as Prioritized Pre-emptive Scheduling with Time Slicing.

But you can select scheduling algorithms by setting corresponding configuration bits in a FreeRTOSConfig.h file.

This configUSE_PREEMPTION configuration constant is used to select a fixed priority scheduling policy and configUSE_TIME_SLICING is used to select the time slicing scheduler. But if you want to select both, you can set this configuration bit to one as shown in the figure below:



```
// And on to the things the same no matter the AVR type...
#define configUSE_PREEMPTION                1
#define configUSE_IDLE_HOOK                 1
#define configUSE_TICK_HOOK                 0
#define configCPU_CLOCK_HZ                  ( ( uint32_t ) F_CPU )
#define configMAX_PRIORITIES                4
#define configMINIMAL_STACK_SIZE            ( 192 )
#define configMAX_TASK_NAME_LEN             ( 8 )
#define configUSE_TRACE_FACILITY            0
#define configUSE_16_BIT_TICKS              1
#define configIDLE_SHOULD_YIELD             1

#define configUSE_MUTEXES                   1
#define configUSE_RECURSIVE_MUTEXES         1
#define configUSE_COUNTING_SEMAPHORES       1
#define configUSE_QUEUE_SETS                1
#define configQUEUE_REGISTRY_SIZE           1
#define configUSE_TIME_SLICING              1
#define configCHECK_FOR_STACK_OVERFLOW      1
#define configUSE_MALLOC_FAILED_HOOK        1
```

FreeRTOS Prioritized Preemptive Scheduling with Time Slicing

In this case, a scheduler cannot change the priority of scheduled tasks. But tasks can change their own priority by using FreeRTOS priority changing API

Because this scheduling algorithm also uses Preemptive scheduling, therefore, a high priority task can immediately preempt a low priority running task. Whenever a low priority task gets preempted by a high priority task, a low priority task enters the ready state and allows a high priority task to enter the running state.

The last part of this scheduling algorithm is a time-slicing feature. In this case, all equal priority tasks get shared CPU processing time. Time slicing is achieved by using FreeRTOS tick interrupts.

This picture shows the timing diagram about the execution sequence of high, medium and low priority tasks.

This diagram shows the execution sequence of tasks according to the time-slicing policy.

FreeRTOS Arduino : How to Delete Tasks with vTaskDelete() API

In real-time operating systems, multitasking is achieved by dividing each application into useful tasks. But not all tasks execute throughout the execution of the program. Some tasks are more important and execute more frequently than others. Therefore, in RTOS, some tasks can be deleted during program execution when there is no need to call them in the near future or when we do not need them anymore in the program. Hence, in this tutorial, we will learn how to **delete tasks** using **FreeRTOS API** with Arduino.

Before preceding further with the tutorial, you should read to know how to use FreeRTOS with Arduino. You must read this getting started tutorial:

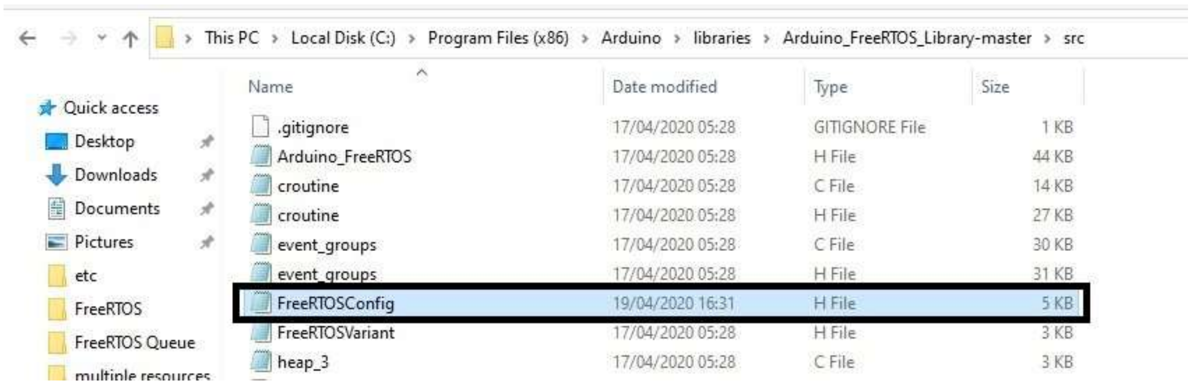
How to Create Tasks with FreeRTOS an Arduino

FreeRTOS Task deleting API

vTaskDelete() FreeRTOS API function is used to delete tasks. With the help of this API function, any task can delete itself and can also delete other tasks by passing reference by a handler to **vTaskDelete()** function.

Note: Before using vTaskDelete(), you should make changes to FreeRTOSConfig.h by setting INCLUDE_vTaskDelete = 1. Otherwise, vTaskDelete() API do not work.

To set INCLUDE_vTaskDelete = 1, go to the libraries folder of Arduino IDE. After that, open Arduino FreeRTOS library folder>>scr

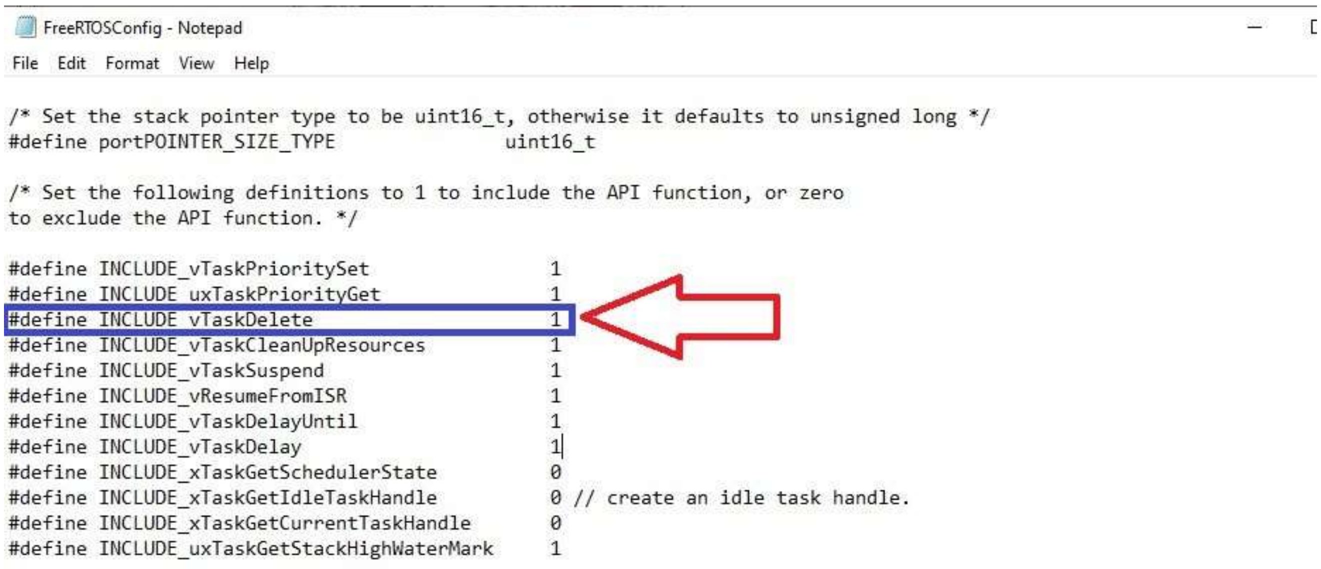

This PC > Local Disk (C:) > Program Files (x86) > Arduino > libraries > Arduino_FreeRTOS_Library-master > src
Quick access
Desktop
Downloads
Documents
Pictures
etc
FreeRTOS
FreeRTOS Queue
multiple resources

Name	Date modified	Type	Size
.gitignore	17/04/2020 05:28	GITIGNORE File	1 KB
Arduino_FreeRTOS	17/04/2020 05:28	H File	44 KB
croutine	17/04/2020 05:28	C File	14 KB
croutine	17/04/2020 05:28	H File	27 KB
event_groups	17/04/2020 05:28	C File	30 KB
event_groups	17/04/2020 05:28	H File	31 KB
FreeRTOSConfig	19/04/2020 16:31	H File	5 KB
FreeRTOSVariant	17/04/2020 05:28	H File	3 KB
heap_3	17/04/2020 05:28	C File	3 KB

C:\Program Files (x86)\Arduino\libraries\Arduino_FreeRTOS_Library-master\src

After that, open to FreeRTOSConfig.h file as a administrator, because you will not able to save changes without opening this file as a administrator.

Now set INCLUDE_vTaskDelete = 1 as shown below. If it is set to one by default, leaving it as it is.



```
/* Set the stack pointer type to be uint16_t, otherwise it defaults to unsigned long */
#define portPOINTER_SIZE_TYPE                    uint16_t

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vResumeFromISR                  1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          0
#define INCLUDE_xTaskGetIdleTaskHandle          0 // create an idle task handle.
#define INCLUDE_xTaskGetCurrentTaskHandle       0
#define INCLUDE_uxTaskGetStackHighWaterMark     1
```

How to use vTaskDelete() API with Arduino?

This is prototype of task deletion routine. It does not return any value and input argument is reference hander of the task which you want to delete.

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

As you know that while creating a task with xTaskCreate() API function, we define handle of the task also. For example, we created a task like this:

```
xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );
```

Here the last argument is a handle of the task “vTask2”.

Now if you want to delete the above-created task any time during the program execution, you can simply call **vTaskDelete() API** by passing handler to the task like this:

```
vTaskDelete( xTask2Handle );
```

Note: By using handler task can be deleted from any other task. Also, A task can delete itself by passing “Null” to vTaskDelete” inside its own task function definition.

If you are still confused about FreeRTOS task deletion API, you will understand when we explain with programming example.

FreeRTOS Task Deletion Example with Arduino

In order to demonstrate the use of **FreeRTOS task delete API function**, we create a very simple example with **Arduino**. We create two tasks such as Task1(LED1) and Task2(LED2).

Example with Arduino

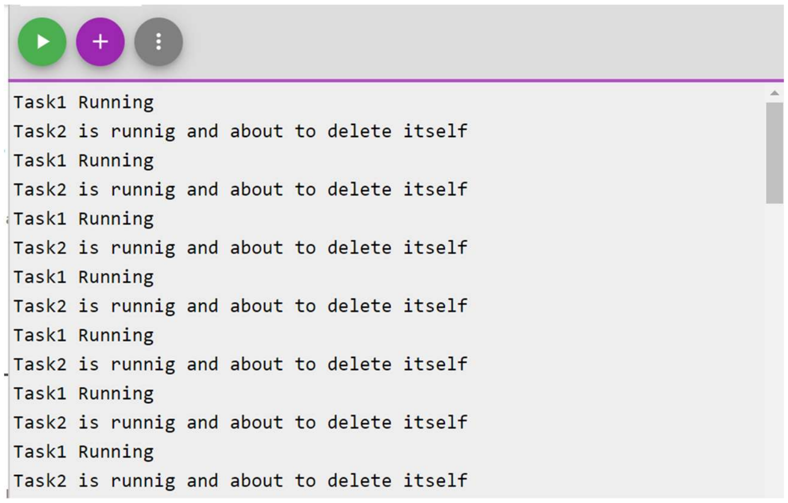
- 1. Task1 is inside setup function with priority 1. When it runs, it creates Task2 at priority 2. Task 2 is now the highest priority task, so it starts to execute immediately.
- Task2 itself does not do anything except delete itself. It could delete itself by passing NULL to vTaskDelete() but instead, for demonstration purposes, it uses its own task handle.
- When Task 2 has been deleted, Task1 will become again the highest priority task. But this time, it calls vTaskDelay() API and enters the block state for the specified time.
- The Idle task executes while Task1 is in the blocked state and frees the memory that was allocated to the now-deleted Task2.
- When Task1 leaves the blocked state it again becomes the highest priority Ready state task and so pre-empts the Idle task. When it enters the Running state it creates Task 2 again, and so it goes on.

```
#include <Arduino_FreeRTOS.h>

#include "task.h"


TaskHandle_t TaskHandle_2; // handler for Task2


void setup()

{

 Serial.begin(9600); // Enable serial communication library.

 pinMode(4, OUTPUT);  // define LED1 pin as a digital output

 pinMode(5, OUTPUT);  // define LED2 pin as a digital output


 //Create the first task at priority 1

 // Name of task is "LED1"

 // Stack size is set to 100

 // we do not pass any value to Task1. Hence, third agument is NULL

 // Set the priority of task to one

 // Task1 handler is not used. Therefore set to Null

 xTaskCreate(Task1, "LED1", 100, NULL, 1, NULL);


 // Start FreeRTOS scheduler in Preemptive timing silicing mode

 vTaskStartScheduler();

}
```

```
void loop()

{

// Do nothing as schduler will allocated CPU to Task1 and Task2 automatically

}

/* Task1 with priority 1 */

void Task1(void* pvParameters)

{


 while(1)

 {

  Serial.println("Task1 Running"); // print "Task1 Running" on Arduino Serial Monitor

  digitalWrite(4, HIGH); // sets the digital pin 4 on

  digitalWrite(5, LOW); // sets the digital pin 5 off

  xTaskCreate(Task2, "LED2", 100, NULL, 2, &TaskHandle_2); // create task2 with priority 2

  vTaskDelay( 100 / portTICK_PERIOD_MS ); // wait for one second

 }

}


/* Task2 with priority 2 */

void Task2(void* pvParameters)

{

  //digitalWrite(5, HIGH); // sets the digital pin 5 high

  //digitalWrite(4, LOW); // sets the digital pin 4 low

  Serial.println("Task2 is runnig and about to delete itself");

  vTaskDelete(TaskHandle_2);    //Delete own task by passing NULL(TaskHandle_2 can also be used)

}
```

Arduino Serial Monitor Output

This diagram shows the output of Arduino Serial monitor when program executed



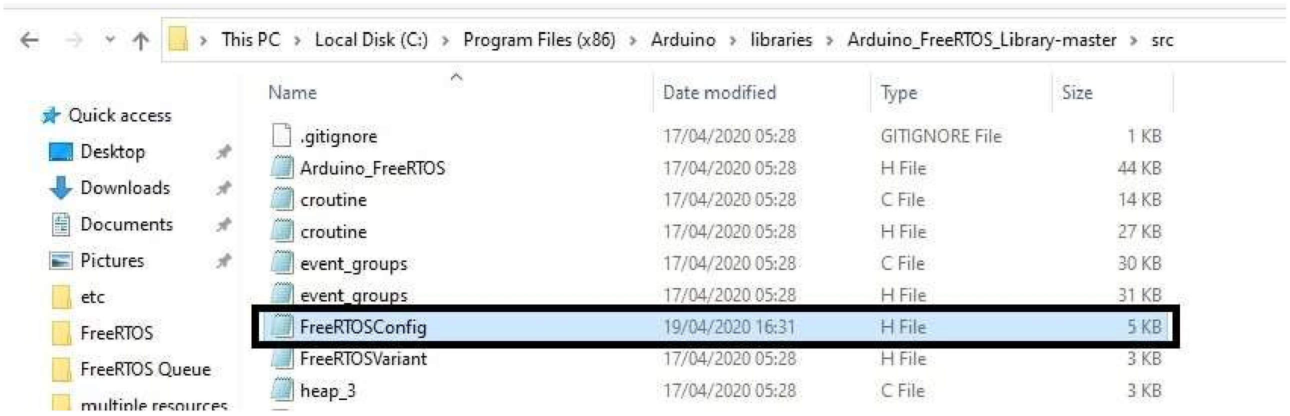
```
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
Task1 Running
Task2 is runnig and about to delete itself
```

- At T1, task1 being a highest priority task starts to execute. It turns on LED1(D4) and makes LED2 low. Moreover, it also prints “Task1 running” on Arduino Serial monitor. After that, it create Task2 with priority 2 which is higher than task2. Hence, Task2 becomes the highest priority and blocked Task2.
- At T2, Task2 enters the running state and start its execution. It does nothing except deleting itself and displaying “Task2 is runnig and about to delete itself” on Serial monitor of Arduino. It also inverses the state of GPIO pins D4 and D5.
- At T3, Task1 calls vTaskDelay(), allowing the idle task to run until the delay time expires, and the whole sequence repeats.

FreeRTOS Arduino: Changing the Priority of a Task

vTaskPrioritySet() API Function

vTaskPrioritySet() is used to set task priority after the scheduler has been started.



This PC › Local Disk (C:) › Program Files (x86) › Arduino › libraries › Arduino_FreeRTOS_Library-master › src

Name	Date modified	Type	Size
.gitignore	17/04/2020 05:28	GITIGNORE File	1 KB
Arduino_FreeRTOS	17/04/2020 05:28	H File	44 KB
croutine	17/04/2020 05:28	C File	14 KB
croutine	17/04/2020 05:28	H File	27 KB
event_groups	17/04/2020 05:28	C File	30 KB
event_groups	17/04/2020 05:28	H File	31 KB
FreeRTOSConfig	19/04/2020 16:31	H File	5 KB
FreeRTOSVariant	17/04/2020 05:28	H File	3 KB
heap_3	17/04/2020 05:28	C File	3 KB

This is a prototype of task priority changing function:

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

The first argument is a handle to the task which priority we want to change. While creating a task with xTaskCreate() API function, priority is also defined. Alternatively, a task can change its own priority by passing NULL in place of a valid task handle.

The second argument defines the task's priority level. The maximum priority a task can attain is set with (configMAX_PRIORITIES – 1. Here configMAX_PRIORITIES is constant that can be set using FreeRTOSConfig.h header file.

Acquiring Task Priority

uxTaskPriorityGet() API function can be used to get the level task priority.

Before using this API, make sure that INCLUDE_uxTaskPriorityGet is set to 1 in FreeRTOSConfig.h header file.

This is a prototype of a task priority get function. Similar to vTaskPrioritySet(), we pass handler as a input to this function to which priority we want to know.

UBaseType_t uxTaskPriorityGet(TaskHandle_t pxTask);

Tasks Priority Changing Example Arduino

In this example, we use two tasks in order to demonstrate the use of vTaskPrioritySet() API function. We create two task one with higher priority and one with low priority. We change their priorities using their reference handlers.

First, these lines adds libraries of FreeRTOS and tasks. We also define prototypes for two tasks such as Task1 and Task2.

```
#include <Arduino_FreeRTOS.h>#include <task.h>void Task1( void *pvParameters );void Task2( void
*pvParameters );
```

We should also define reference handlers to these two tasks. Because we use these TaskHandle_t type variables to change task priority.

```
TaskHandle_t TaskHandle_1; // handler for Task1

TaskHandle_t TaskHandle_2; // handler for Task2
```

This code creates two tasks with different priorities. Task1 has the highest and Task2 has the lowest priority. Hence, Task1 will be the first one in the running state selected by the FreeRTOS scheduler.

```
Serial.begin(9600); // Enable serial communication library.

xTaskCreate(Task1, "LED1", 100, NULL, 2, &TaskHandle_1); // Create Task1 with priority=2

xTaskCreate(Task2, "LED2", 100, NULL, 1, &TaskHandle_2); // Create Task2 with priority=1

vTaskStartScheduler(); // start FreeRTOS scheduler
```

This routine defines Task1. Firstly, we get the priority of the task using uxTaskPriorityGet() and store it in uxPriority variable of type UBaseType_t. After that inside the while(1) loop, it prints a string "Task1 is running and about to raise Task2 Priority". In the end, we call vTaskPrioritySet() to raise the priority of Task2 by one higher than Task1.

Therefore, Task2 priority will become higher than the Task1. In response, Task1 enters the Ready state and Task2 enters the running state.

```
//definition of Task1void Task1(void* pvParameters){

    UBaseType_t uxPriority = uxTaskPriorityGet( NULL );

   while(1)

   {

   Serial.println("Task1 is running and about to raise Task2 Priority");

   vTaskPrioritySet( TaskHandle_2, ( uxPriority + 1 ) );



   }}
```

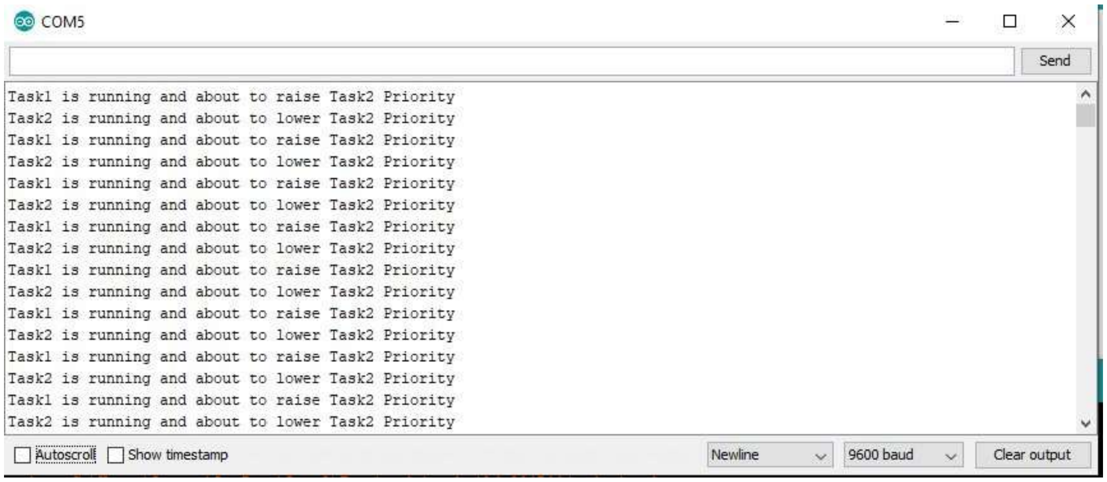
This routine defines Task2. Firstly, we get the priority of the task2 using uxTaskPriorityGet() and store it in uxPriority variable of type UBaseType_t. After that inside the while(1) loop, it prints a string "Task2 is running and about to lower Task2 Priority". In the end, we call vTaskPrioritySet() to lower the priority of Task2 by one.

Therefore, Task1 priority will become higher than the Task2 again. In response, Task2 enters the Ready state, and Task2 enters the running state.

```
void Task2(void* pvParameters){

 UBaseType_t   uxPriority = uxTaskPriorityGet( NULL );

    while(1)

   {

   Serial.println("Task2 is running and about to lower Task2 Priority");

   vTaskPrioritySet( TaskHandle_2, ( uxPriority - 2 ) );



   }

 }
```

```
#include <Arduino_FreeRTOS.h>
#include <task.h>
void Task1( void *pvParameters );
void Task2( void *pvParameters );

TaskHandle_t TaskHandle_1; // handler for Task1
TaskHandle_t TaskHandle_2; // handler for Task2

void setup()
{
 Serial.begin(9600); // Enable serial communication library.

  xTaskCreate(Task1, "LED1", 100, NULL, 3, &TaskHandle_1);
  xTaskCreate(Task2, "LED2", 100, NULL, 2, &TaskHandle_2);
  vTaskStartScheduler();
}

void loop()
{
 // put your main code here, to run repeatedly:

}

//definition of Task1
void Task1(void* pvParameters)
{
    UBaseType_t uxPriority = uxTaskPriorityGet( NULL );
   while(1)
   {
   Serial.println("Task1 is running and about to raise Task2 Priority");
   vTaskPrioritySet( TaskHandle_2, ( uxPriority + 1 ) );

   }
}
```

```
void Task2(void* pvParameters)

{

 UBaseType_t   uxPriority = uxTaskPriorityGet( NULL );

    while(1)

   {

   Serial.println("Task2 is running and about to lower Task2 Priority");

   vTaskPrioritySet( TaskHandle_2, ( uxPriority - 2 ) );


   }


}
```



```
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
Task1 is running and about to raise Task2 Priority
Task2 is running and about to lower Task2 Priority
```

How Priority Changing Code works?

- At the start, Task1 being the highest priority run first and enters the running state. After that, it prints out strings before raising the priority of Task2 to above its own priority. Because now Task2 runs and Task1 waits in the ready state.
- Task2 enters the Running state as soon as it has the highest relative priority. Only one task can be in the Running state at any one time, so when Task 2 is in the Running state, Task1 is in the Ready state.
- Task2 prints out “Task2 is running and about to lower Task2 Priority” before setting its own priority back down to below that of Task1.
- Task2 setting its priority back down means Task 1 is once again the highest priority task, so Task1 re-enters the Running state, forcing Task 2 back into the Ready state.