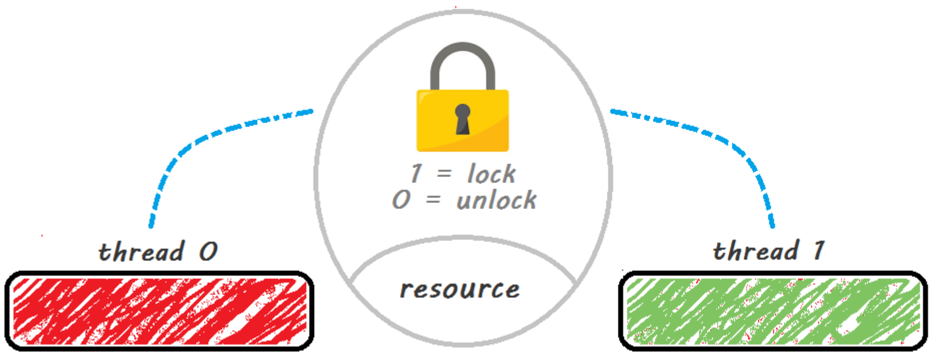
Binary Semaphore Introduction

It is a kernel object that a task can acquire for the purpose of tasks synchronization. Semaphore is a like a token which allows a task to carry out execution if the task can acquire semaphore. Otherwise, the task remains in the block state and can not execute unless it acquires binary semaphore. But as soon as, task acquires the intended token, it begins to carry-out its execution.

The value of binary semaphore can be either one (available) or zero (Empty or not available). If a binary semaphore is already acquired by task one and task 2 also tries to access binary semaphore, but it is already taken. Therefore, task 2 will enter the blocking state and remain in the blocking state until task 1 releases the binary semaphore.



Multiple tasks can use binary semaphore within a single application. But only one task can acquire it at a time. Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it.

TASK-1	(BS)	KEY(SEMAPHORE)	CPU
TASK-2	(Blocked State)		
TASK-3	(ACQUIRE SEMPHORE -→ Ready to Running State)		
TASK-4	(BS)		

0 -→ key(semaphore) is not available

1 → Key (semaphore) is available

Where to use Binary Semaphores?

Binary semaphores can be used for different things. But most often, they are used for task synchronization. Especially, we can use a binary semaphore to synchronize an interrupt service routine in a task. Therefore, in this tutorial, what we're going to discuss is how to use binary semaphores for synchronization of tasks

Synchronzation of FreeRTOS Tasks

But we mostly use binary semaphores to achieve deferred interrupt processing. We can use binary semaphores to synchronize tasks. It can synchronize multiple tasks. We can place semaphores at specific portions of different parts of the code and make them all meet at the same point and execute them at the same time.

FreeRTOS Binary Semaphore API

First, let's talk about FreeRTOS API functions that are used to take and give semaphores either through normal tasks or through interrupt service routines. We'll quickly go through the FreeRTOS semaphore APIs and after that see examples with Arduino.

FreeRTOS Create Binary Semaphore

Let's start with his very first one. xSemaphoreCreateBinary() is used to create a Binary semaphore.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

This function takes no argument. However, it returns a semaphore handler. We use this semaphore reference handler to check whether it can be created or not.

If it returns a null value, this means there isn't enough heap memory available for freeRTOS to allocate the semaphore data structures. If it returns a non-null value, this means Binary Semaphore is successfully created and it is available to use. Therefore, we must create this token before using it.

FreeRTOS xSemaphoreTake() API

The next semaphore relatedFreeRTOS API is the xSemaphoreTake(). xSemaphoreTake() API Function is basically used for taking/acquiring a binary semaphore. To take a semaphore means to obtain or receive a semaphore. Now you know that it can be taken only if it is available or not currently acquired by any other task.

If in any application, we have two tasks that use a single binary semaphore. If one task takes the semaphores. There is no semaphore available for other tasks to take and other tasks will remain in the blocking state. Therefore, a semaphore can only be taken if it's available.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait
);
```

```
Void add (int a, float b)
```

This function takes two arguments. The first argument is the handle of the binary semaphores, the handle of the semaphore we wish to take and the second argument is the XTickstowait(), and this argument basically specifies the maximum amount of time the task should remain in the block state to wait for the binary semaphore.

FreeRTOS xSemaphoreGive() API

The last important FreeRTOS API is xSemaphoreGive(). It is used to release binary/counting semaphores. We can also give/release a semaphore. Once a particular task no longer needs it, we can give it back by using xSemaphoreGive() function.

```
BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

This function takes just one argument and this argument is the handle of the semaphore that is being given back. For instance, one task takes a binary semaphore. The semaphore is no longer available but after that task gives it back and it becomes available for another task to take.

But if you are releasing binary semaphore from an interrupt service routine, you should also interrupt a safe version of this function such as xsemaphoresgivegivefromISR().

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

This interrupt safe version takes two input arguments. The first argument is similar to the first one we saw which is the handle of the semaphores and the second argument over here is called the HigherPriorityTaskWoken. HigherPriorityTaskWoken ensures that the interrupt returns directly to the highest priority Ready state task.

For more information about this funcion, check this link.

FreeRTOS Binary Semaphore Example to Synchronize Tasks

In this example, we will create two tasks. One task turn-on LED and other task turns-off LED. But both tasks can not execute at the same time. Because both share the same binary semaphore. Both tasks have the same priority level. Therefore, the FreeRTOS scheduler will schedule both tasks in a time-sharing or round-robin fashion.

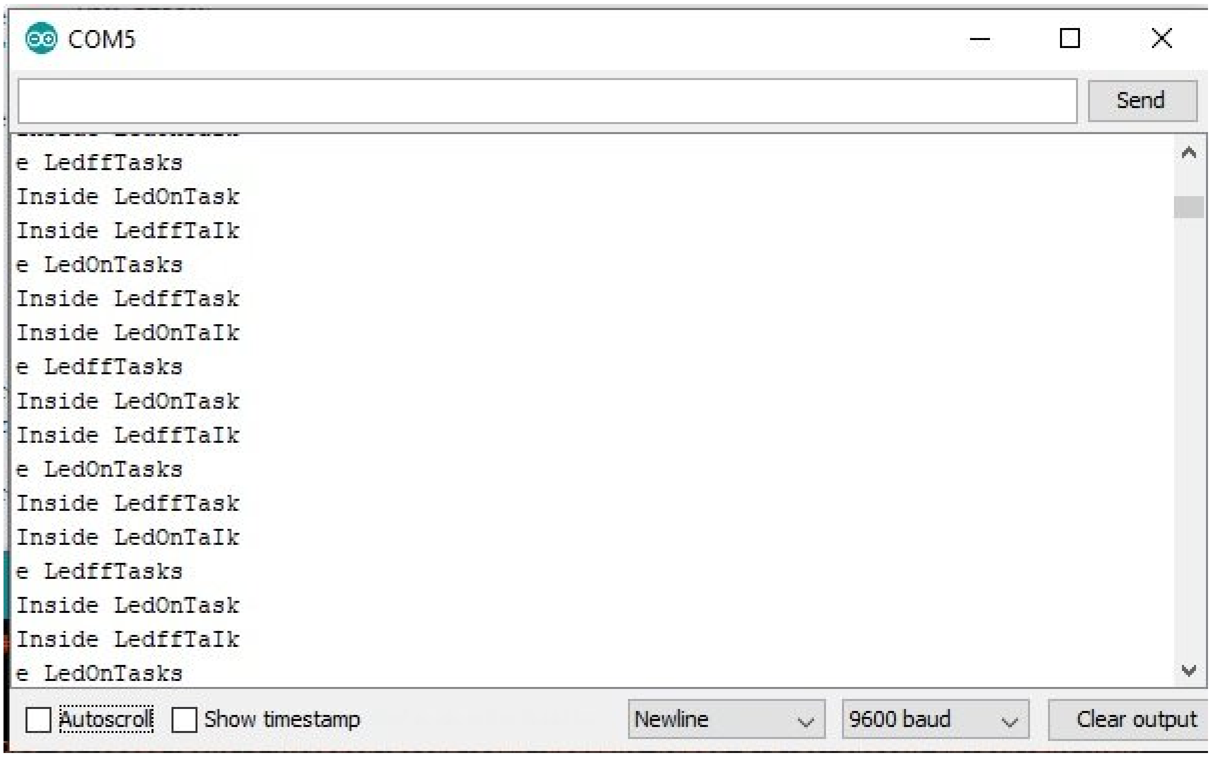
If we do not use a binary semaphore to synchronize these tasks, tasks will not execute according to the order we want and it will follow a haphazard form of round-robin scheduling. We want to execute LedOnTask and after that LedOffTaks or vice versa. To maintain this order, we have to synchronize both tasks using binary semaphore.

Arduino Tasks Synchronization Code

```
#include <Arduino_FreeRTOS.h>

#include "semphr.h"

#define  LED  13

SemaphoreHandle_t xBinarySemaphore;

void setup()

{

  Serial.begin(9600);

  pinMode(LED ,OUTPUT);

  xBinarySemaphore = xSemaphoreCreateBinary();

  xTaskCreate(LedOnTask, "LedON",100,NULL,1,NULL);
  xTaskCreate(LedoffTask, "LedOFF", 100,NULL,1,NULL);

  xSemaphoreGive(xBinarySemaphore);

}

void loop()

{

}

void LedOnTask(void *pvParameters)

{

  while(1)

  {

   xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);

   Serial.println("Task-1-Acquires the semaphore");

   digitalWrite(LED,HIGH);

   xSemaphoreGive(xBinarySemaphore);

Serial.println("Task-1-Releases the semaphore");

   vTaskDelay(1);

  }

}

void LedoffTask(void *pvParameters)

{

  while(1)

  {

    xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);

    Serial.println("Task-2-Acquires the semaphore");

    digitalWrite(LED,HIGH);
    xSemaphoreGive(xBinarySemaphore);

Serial.println("Task-2-Releases the semaphore");

    vTaskDelay(1);

  }

}
```

Output without Semaphore

If we uncomment the semaphore functions from the above Arduino code and upload code to Arduino, you will see this type of output on the serial monitor.

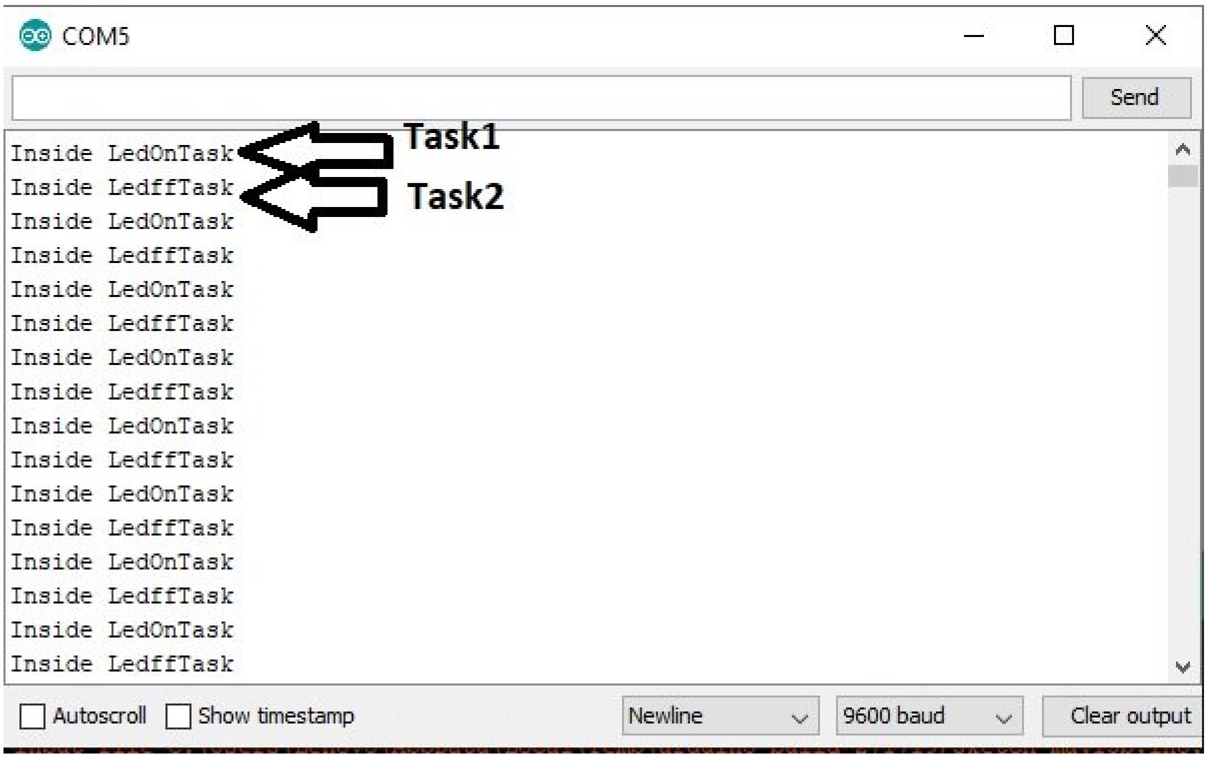


```
e LedffTasks
Inside LedOnTask
Inside LedffTaIk
e LedOnTasks
Inside LedffTask
Inside LedOnTaIk
e LedffTasks
Inside LedOnTask
Inside LedffTaIk
e LedOnTasks
Inside LedffTask
Inside LedOnTaIk
e LedffTasks
Inside LedOnTask
Inside LedffTaIk
e LedOnTasks
```

Because tasks are not synchronized and scheduler follows time-sharing scheduling. Hence, for some time, it executes “LedoffTask” and for some timer “LedOnTask”. Therefore, we get a mixed output on the Arduino Serial monitor. But, we can make it in order by using binary semaphore.

Output with Semaphore

This picture shows the output of serial monitor when the binary semaphore is used for both tasks such as “LedoffTask” and “LedOnTask”. You can see that both tasks are executing in order.



```
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
Inside LedOnTask
Inside LedffTask
```

Newline
9600 baud

Video Demo