

**Name: Vaibhav Gorakh Lonkar**  
**PRN: 21410027**  
**Batch: EN2**

## **RTOS Experiment No.9**

**Title:** Program to illustrate need of semaphore.

### **Objective-**

1. Understand the need of semaphore and its functionality.
2. Understand the concept of priority inversion.

### **What is semaphore?**

A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:

- a) Control the access of the shared resources (mutual exclusion);
- b) Allow two tasks to synchronize the activities.
- c) Signal the occurrence of an event.

A semaphore is a key that your code acquires in order to continue execution. If semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.

### **There are two types of semaphores:**

- a) Binary Semaphore: The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
- b) Counting Semaphore: These are integer value semaphores and can be any non-negative integer.

### **1.Code without Semaphore:**

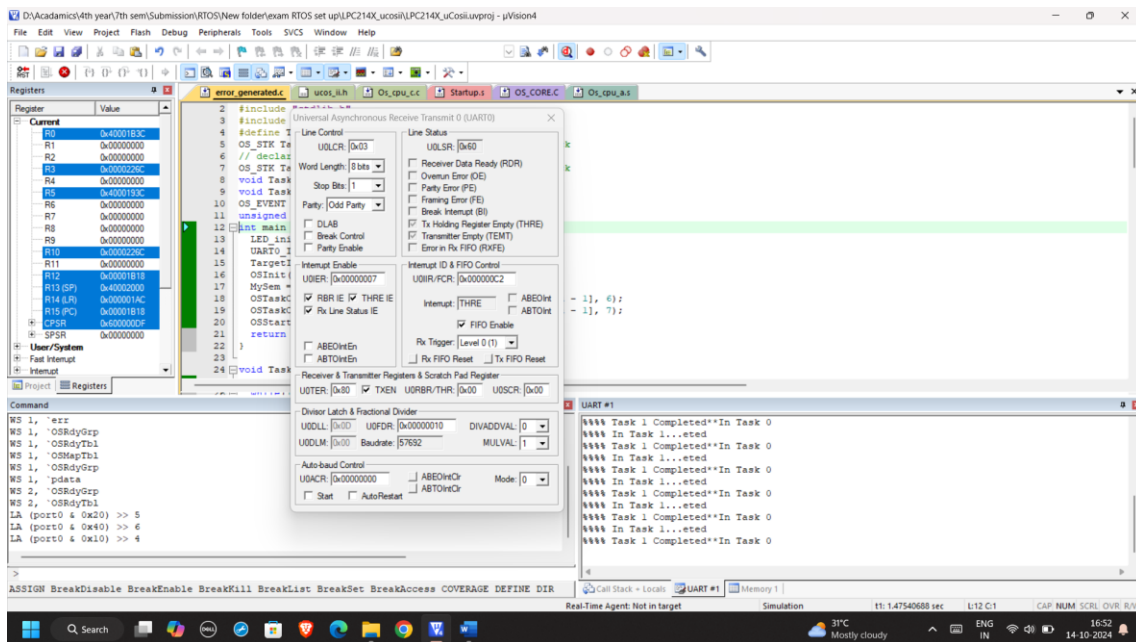
```
#include "config.h"
#include "stdlib.h"
#include <stdio.h>
#define TaskStkLengh 64 //Define Task stack length
OS_STK TaskStk0 [TaskStkLengh]; //Define the Task0 stack
// declare the tasks
OS_STK TaskStk1 [TaskStkLengh]; //Define the Task1 stack
void Task0(void *pdata);
void Task1(void *pdata);
OS_EVENT *MySem;
unsigned char err;
int main (void){
    LED_init();
    UART0_Init();
    TargetInit();
    OSInit();
```

```

MySem = OSSemCreate(1);
OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
OSTaskCreate (Task1,(void *)0, &TaskStk1[TaskStkLengh - 1], 7);
OSStart();
return 0;
}
void Task0(void *pdata){
    pdata = pdata;
    while(1){
        //Uncomment pend and post to see semaphores in action
        OSSemPend(MySem, 0, &err); UART0_SendData ("**In Task 0\n");
        OSTimeDly(4);
        UART0_SendData ("**** Task 0 Completed\r");
        OSSemPost(MySem);
    }
}
void Task1(void *pdata){
    pdata = pdata;
    while (1){
        //Uncomment pend and post to see semaphores in action
        OSSemPend(MySem, 0, &err);
        OSTimeDly(4);
        UART0_SendData ("% % % % In Task 1...\n");
        OSTimeDly(4);
        UART0_SendData ("% % % % Task 1 Completed");
        OSSemPost(MySem);
    }
}
}

```

## OUTPUT:



## 2.Code with Semaphore:

```
#include "config.h"
#include "stdlib.h"

#define TaskStkLengh 64    //Define the Task0 stack length
OS_STK    TaskStk0 [TaskStkLengh];    //Define the Task0 stack
OS_STK    TaskStk1 [TaskStkLengh];    //Define the Task0 stack
void    Task0(void *pdata);    // Task0
void    Task1(void *pdata);    // Task1
OS_EVENT *MySem;
unsigned char err;

int main(void){
    LED_init();
    lcd_init();
    UART0_Init();
    UART0_SendData("\n\r**\n\r");
    UART0_SendData ("Program for semaphore demo\n\r");
    UART0_SendData ("\n\r");
    TargetInit();
    OSInit ();
    MySem = OSSemCreate(1);
    OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
    OSTaskCreate (Task1,(void *)0, &TaskStk1[TaskStkLengh - 1], 7);
    OSStart();
    return 0;
}

void Task0(void *pdata){
    int i;
    pdata = pdata;    /* Dummy data */
    while(1){
        UART0_SendData("\n\rTask 0 waiting for Semaphore");
        lcd_command(0x80);
        LCD_SendData("    ");
        lcd_command(0x80);
        LCD_SendData("Task 0 waiting");
        OSSemPend(MySem, 0, &err);
        UART0_SendData("\n\rTask 0 got the Semaphore");
        UART0_SendData("\n\rTask 0 now flashing LED-0 for 15 times");
        lcd_command(0x80);
        LCD_SendData("    ");
        lcd_command(0x80);
        LCD_SendData("Task 0 got sem");
        for(i=0; i<15;i++){
            LED_on(0);
            OSTimeDly(10);
            LED_off(0);
            OSTimeDly(10);
        }
    }
}
```

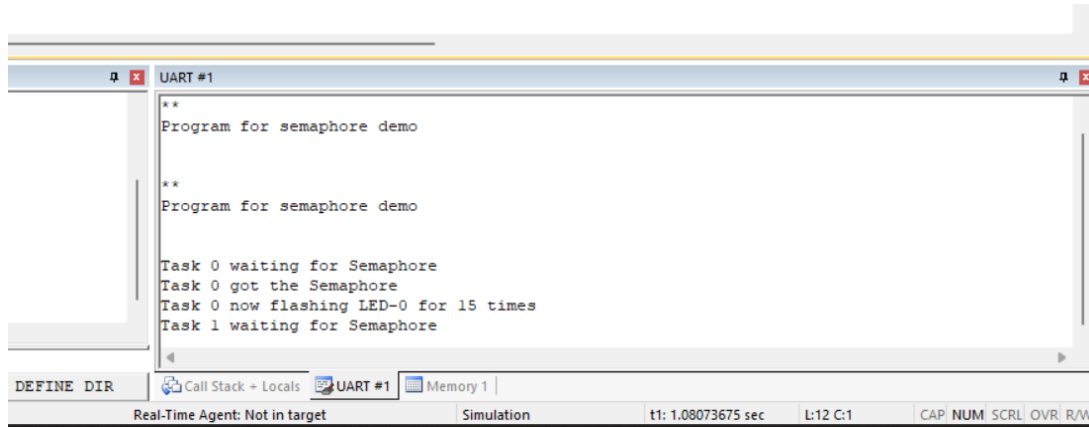
```

        UART0_SendData("\n\rTask 0 released Semaphore \n\r");
        lcd_command(0x80);
        LCD_SendData("    ");
        lcd_command(0x80);
        LCD_SendData("Task 0 released");
        OSSemPost(MySem);
        //OSTimeDly(150);
    }
}

void Task1(void *pdata){
    int i;
    pdata = pdata;    /* Dummy data */
    while(1){
        UART0_SendData("\n\rTask 1 waiting for Semaphore");
        lcd_command(0xC0);
        LCD_SendData("    ");
        lcd_command(0xC0);
        LCD_SendData("Task 1 waiting");
        OSSemPend(MySem, 0, &err);
        UART0_SendData("\n\rTask 1 got the Semaphore");
        UART0_SendData("\n\rTask 1 now flashing LED-1 for 10 times");
        lcd_command(0xC0);
        LCD_SendData("    ");
        lcd_command(0xC0);
        LCD_SendData("Task 1 got sem");
        for(i=0; i<15;i++){
            LED_on(1);
            OSTimeDly(10);
            LED_off(1);
            OSTimeDly(10);
        }
        UART0_SendData("\n\rTask 1 released Semaphore\n\r");
        lcd_command(0xC0);
        LCD_SendData("    ");
        lcd_command(0xC0);
        LCD_SendData("Task 1 released");
        OSSemPost(MySem);
    }
}

```

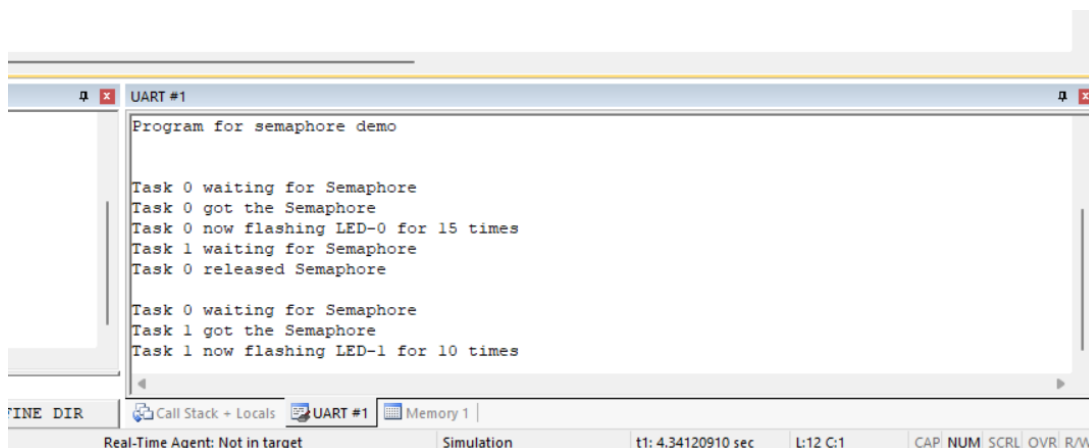
## OUTPUT:



```
UART #1
**
Program for semaphore demo
**
Program for semaphore demo
Task 0 waiting for Semaphore
Task 0 got the Semaphore
Task 0 now flashing LED-0 for 15 times
Task 1 waiting for Semaphore
```

DEFINE DIR | Call Stack + Locals | UART #1 | Memory 1 |

Real-Time Agent: Not in target | Simulation | t1: 1.08073675 sec | L:12 C:1 | CAP NUM SCRL OVR R/W



```
UART #1
Program for semaphore demo
Task 0 waiting for Semaphore
Task 0 got the Semaphore
Task 0 now flashing LED-0 for 15 times
Task 1 waiting for Semaphore
Task 0 released Semaphore
Task 0 waiting for Semaphore
Task 1 got the Semaphore
Task 1 now flashing LED-1 for 10 times
```

DEFINE DIR | Call Stack + Locals | UART #1 | Memory 1 |

Real-Time Agent: Not in target | Simulation | t1: 4.34120910 sec | L:12 C:1 | CAP NUM SCRL OVR R/W

### Priority Inversion:

Priority Inversion is an operating system scenario in which a higher priority process is preempted by a lower priority process. This implies the inversion of priorities of the two tasks.

It is not possible to totally avoid it. But its effect can be minimized by:

- a. Priority ceiling
- b. Disabling interrupts
- c. Priority inheritance
- d. No blocking
- e. Random boosting

### Conclusion:

- i. When we don't use semaphore in the code the output will be not in an order. That means any other function will start running before the completion of current function.
- ii. When we use semaphore, it allows tasks to synchronise their activity.