Real Time Operating Systems (RTOS)

CPET-461

Shared Resources

Exercise #9, #10, #11, & #12

Week #5 Lesson Plan *(February 13th – February 17th)*

Monday – Class

- Shared Resources - Contention

Wednesday – Class

- Overview of Ex-Set #4 - Shared Resources Ex #9 - #12

Thursday / Friday – Lab

- Ex-Set #4 - Shared Resources Ex #9 - #12

Next Week….

- Monday: Reading
  - Book #1 : Chapter #5 (5.1 – 5.2)
- Tuesday: Lab "Report"
  - Ex-Set #4 Report : myCourses Quiz – Due 2/21 @ 11:59 PM
- Wednesday: Reading
  - Book #2 : Chapter #4 (4.1 – 4.4)

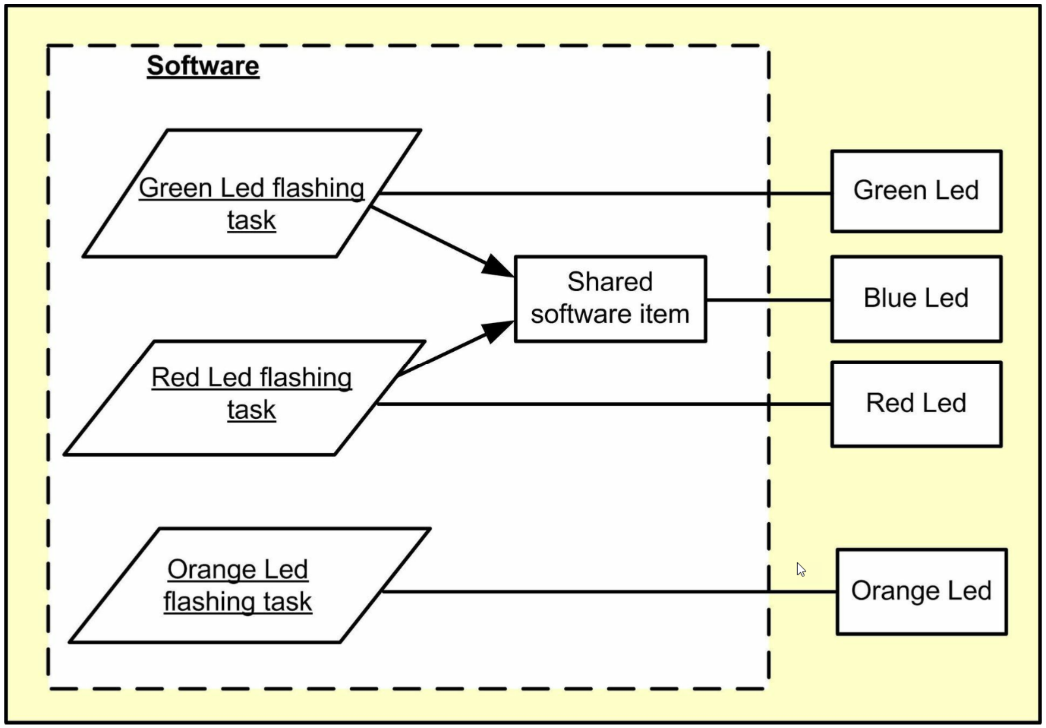
This Week's Exercises…

- **Exercise #9 – Use a MUTEX to Protect Critical Code Sections**
- **Exercise #10 – Use Encapsulation to Improve System Safety & Security**
- **Exercise #11 – Demonstrate the Effects of Priority Inversion**
- **Exercise #12 – Eliminate Priority Inversion by Priority Inheritance**
- Note : This presentation is only an overview of each of the exercises and is intended to clarify any questions that might arise. Please complete the exercises by following the detailed instructions in the textbook.
- Please pay particular attention to the ***Exercise Review*** section at the end of each section. These reviews are excellent at summarizing the exercise's learning objectives.

Fundamental purpose of the exercise is to demonstrate how to eliminate resource contention in a selective manner — using a MUTEX to protect the critical code section.

EXERCISE #9 – USE A MUTEX TO PROTECT CRITICAL CODE SECTIONS

Exercise #9 – Use a MUTEX to Protect Critical Code Sections



Green LED Flashing Task: Normal Priority

Loop:
- *Turn the **Green** LED on*
- *Access the shared data*
- *Delay for 200 mSecs (osDelay)*
- *Turn the **Green** LED off*
- *Delay for 200 mSecs (osDelay)*

End loop.

Red LED Flashing Task: Normal Priority

Loop:
- *Turn the **Red** LED on*
- *Access the shared data*
- *Delay for 550 mSecs (osDelay)*
- *Turn the **Red** LED off*
- *Delay for 550 mSecs (osDelay)*

End loop.

Orange LED Flashing Task: **Above** Normal Priority

Loop:
- *Toggle **Orange***
- *Delay for 50 mSecs (osDelay)*

End loop.

Exercise #9 – Use a MUTEX to Protect Critical Code Sections

Exercise #8 → Exercise #9

```
osMutexWait(CriticalResourceMutexHandle, osWaitForever);   // MUTEX LOCK (i.e. wait)
Access_Function();
osMutexRelease (CriticalResourceMutexHandle);              // MUTEX UNLOCK (i.e. signal)
```

- Note : The above changes should be made to the **Green** & **Red** LED Tasks ONLY. DO NOT MAKE ANY CHANGES TO THE **Orange** LED Task.

Exercise #9 – Use a MUTEX to Protect Critical Code Sections

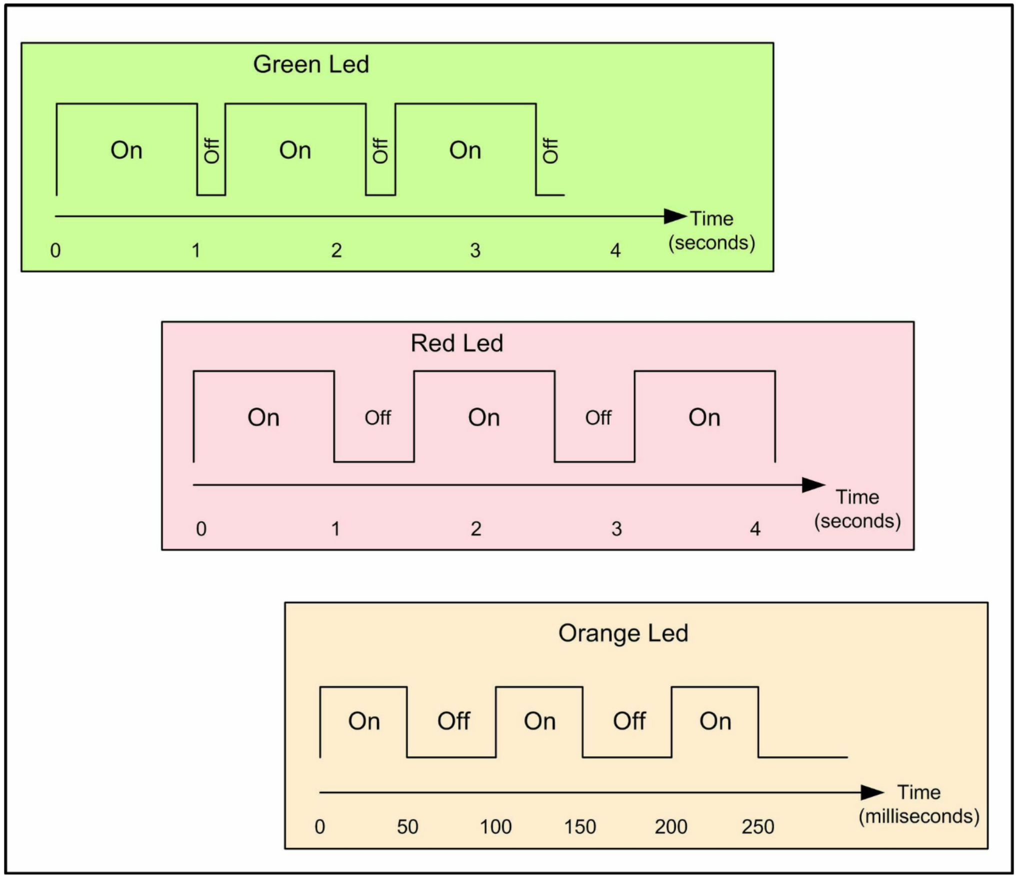
If everything is working properly, again it will, the expected output for the three LEDs is…

What you should see….

1) The three LEDs will appear to be flashing at their correct rates, and they are.
2) The **Blue** LED never turns on, thus indicating that NO resource contention has occurred.

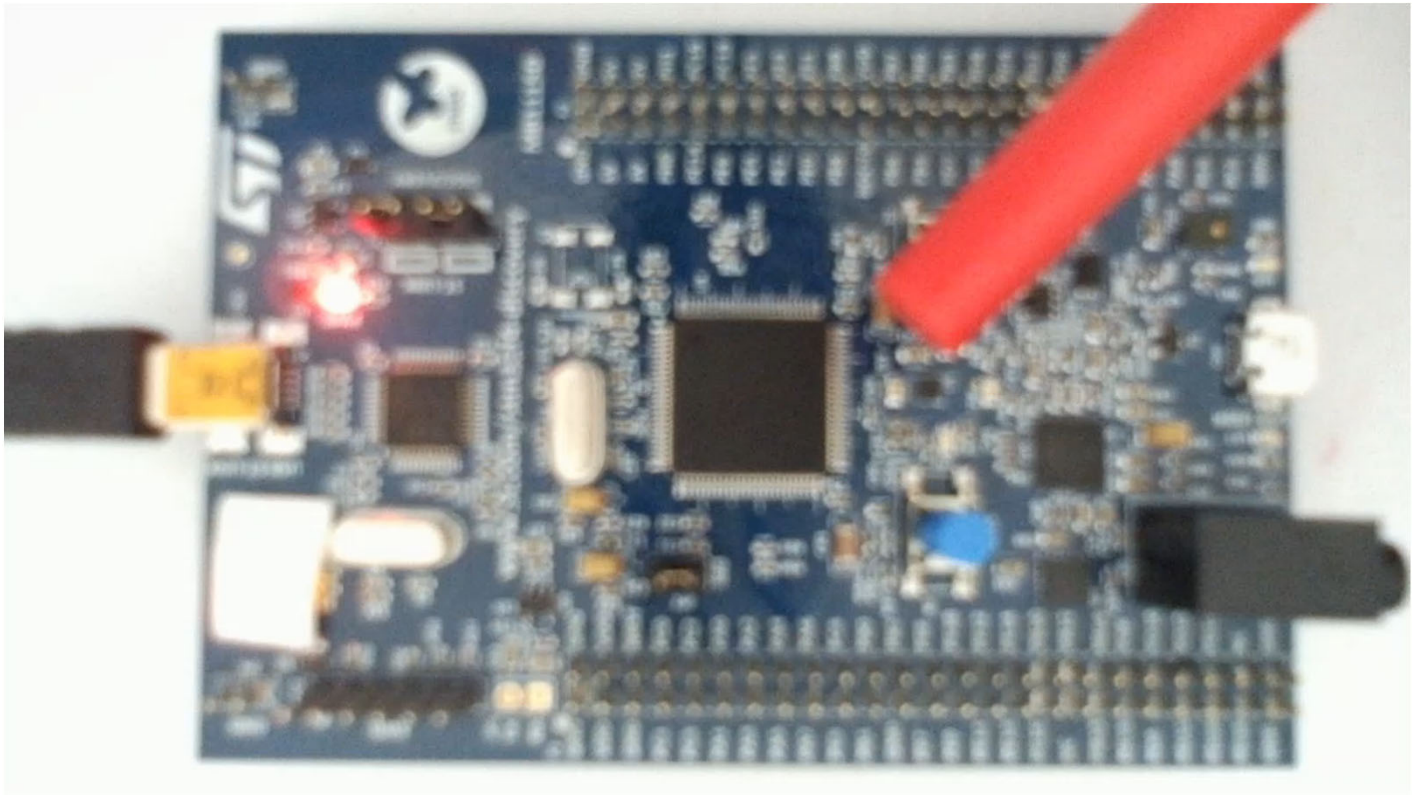
❖ Note : The results for the Semaphore (Ex #8) and MUTEX (#9) implementations are identical.

❖ Note: Unlike a Semaphore, only the task that locks a MUTEX can unlock it. This exercise doesn't show this MUTEX-specific property.



Exercise #9 – Use a MUTEX to Protect Critical Code Sections

Results…

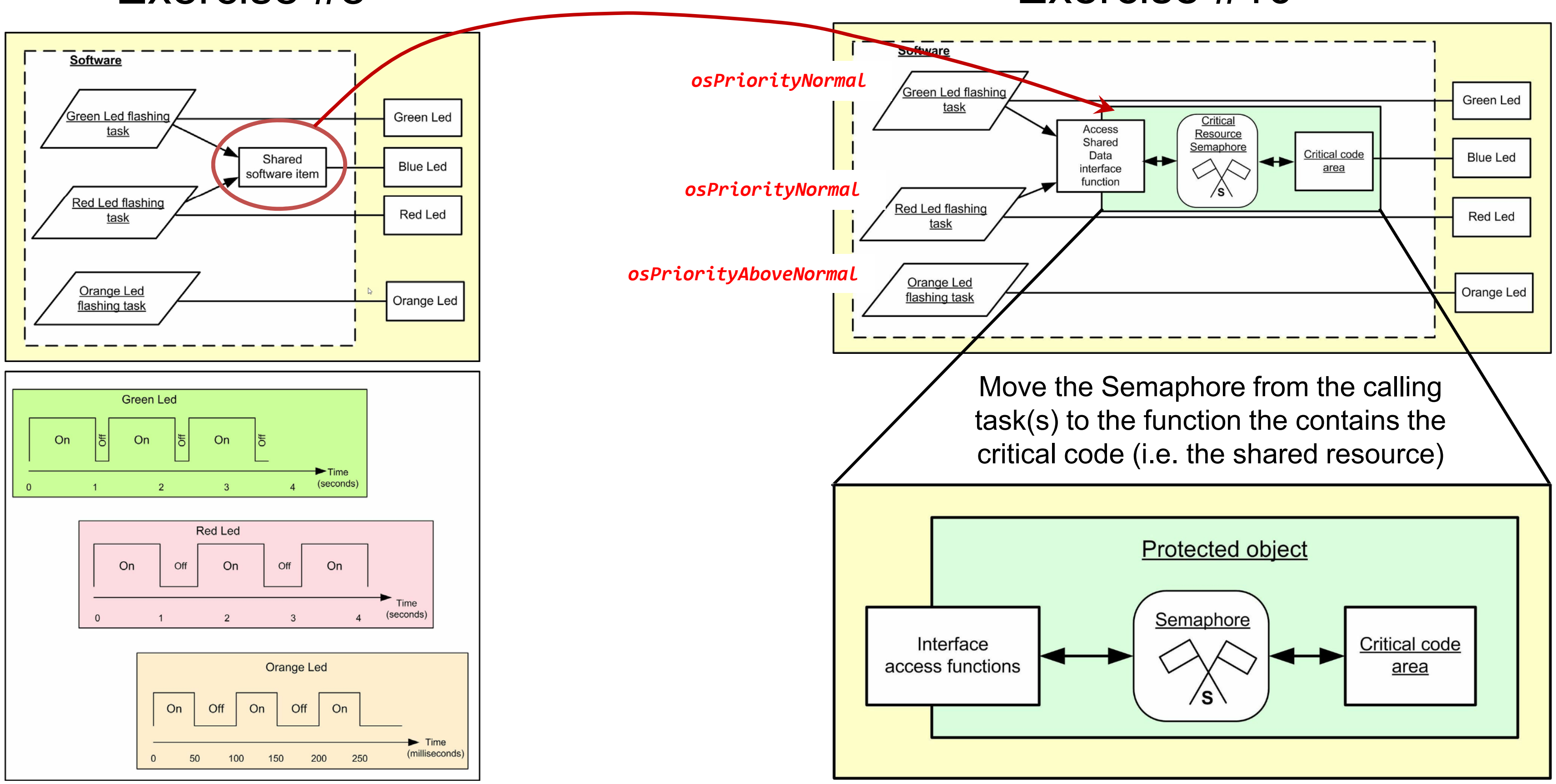


Fundamental purpose of the exercise is to demonstrate that encapsulating an item with its protecting semaphore improves the safety and security of software.

EXERCISE #10 – USE ENCAPSULATION TO IMPROVE SYSTEM SAFETY & SECURITY

Exercise #10 – Use Encapsulation to Improve System Safety & Security

Exercise #8 / Exercise #10



Exercise #10 – Use Encapsulation to Improve System Safety & Security

Exercise #8 → Exercise #10

```
void Start_RED_LED(void const * argument)
{
  for(;;)
  {
      HAL_GPIO_WritePin(GPIOD, RED_LED, GPIO_PIN_SET);

      osSemaphoreWait(CriticalResourceSemaphoreHandle, osWaitForever);  // semaphore P (i.e. wait)
      Access_Function();
      osSemaphoreRelease (CriticalResourceSemaphoreHandle);            // semaphore V (i.e. signal)

      osDelay(550);
      HAL_GPIO_WritePin(GPIOD, RED_LED, GPIO_PIN_RESET);
      osDelay(550);
  }
}
```

```
void Start_GREEN_LED(void const * argument)
{
  for(;;)
  {
      HAL_GPIO_WritePin(GPIOD, GREEN_LED, GPIO_PIN_SET);

      osSemaphoreWait(CriticalResourceSemaphoreHandle, osWaitForever);  // semaphore P (i.e. wait)
      Access_Function();
      osSemaphoreRelease (CriticalResourceSemaphoreHandle);            // semaphore V (i.e. signal)

      osDelay(200);
      HAL_GPIO_WritePin(GPIOD, GREEN_LED, GPIO_PIN_RESET);
      osDelay(200);
  }
}
```

```
osSemaphoreWait(CriticalResourceSemaphoreHandle, osWaitForever);
```

```
void Access_Function()
{
    if (Start_Flag == 1)
    {
        Start_Flag = 0;
    }
    else
    {
        HAL_GPIO_TogglePin(GPIOD, BLUE_LED);   // Toggles Blue LED @ every resource contention
    }
    for (int i = 0; i<2000000; i++);           // ~ 0.5 second delay
    Start_Flag = 1;
    return;
}
```

```
osSemaphoreRelease (CriticalResourceSemaphoreHandle);
```

Exercise #10 – Use Encapsulation to Improve System Safety & Security

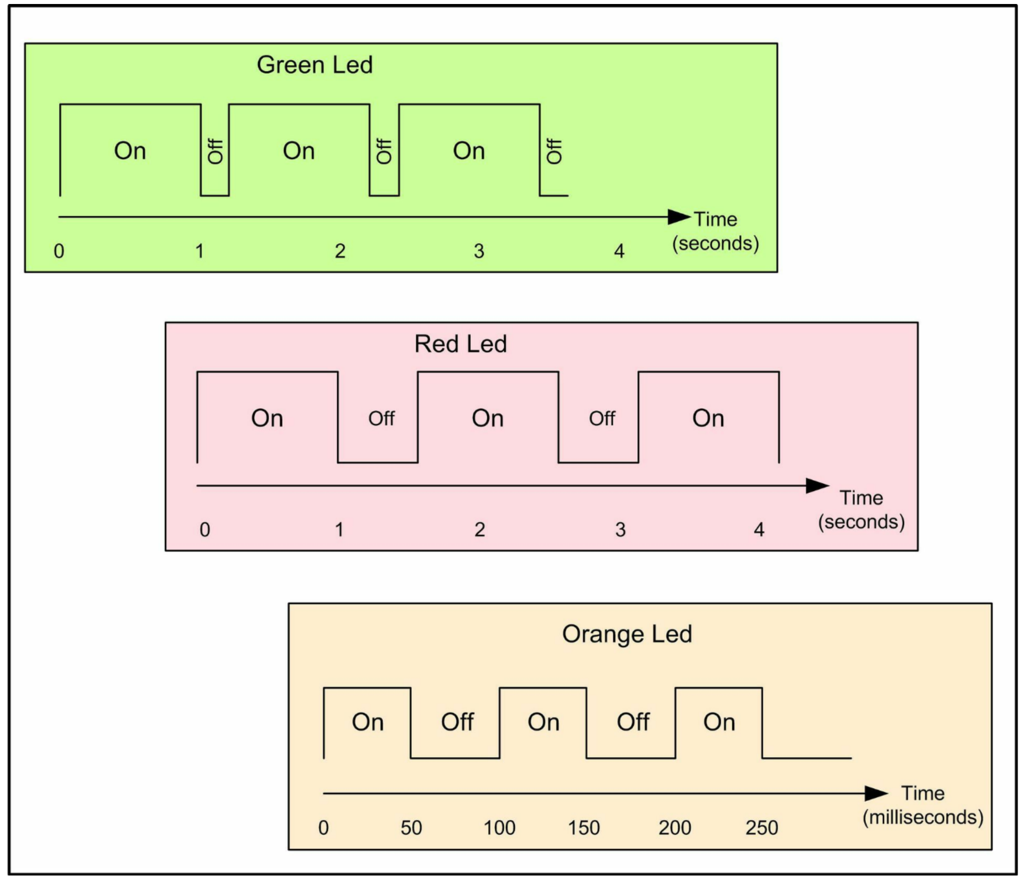
If everything is working properly, the results should be the same as seen in Exercise #8.

The expected output for the three LEDs is…

What you should see….

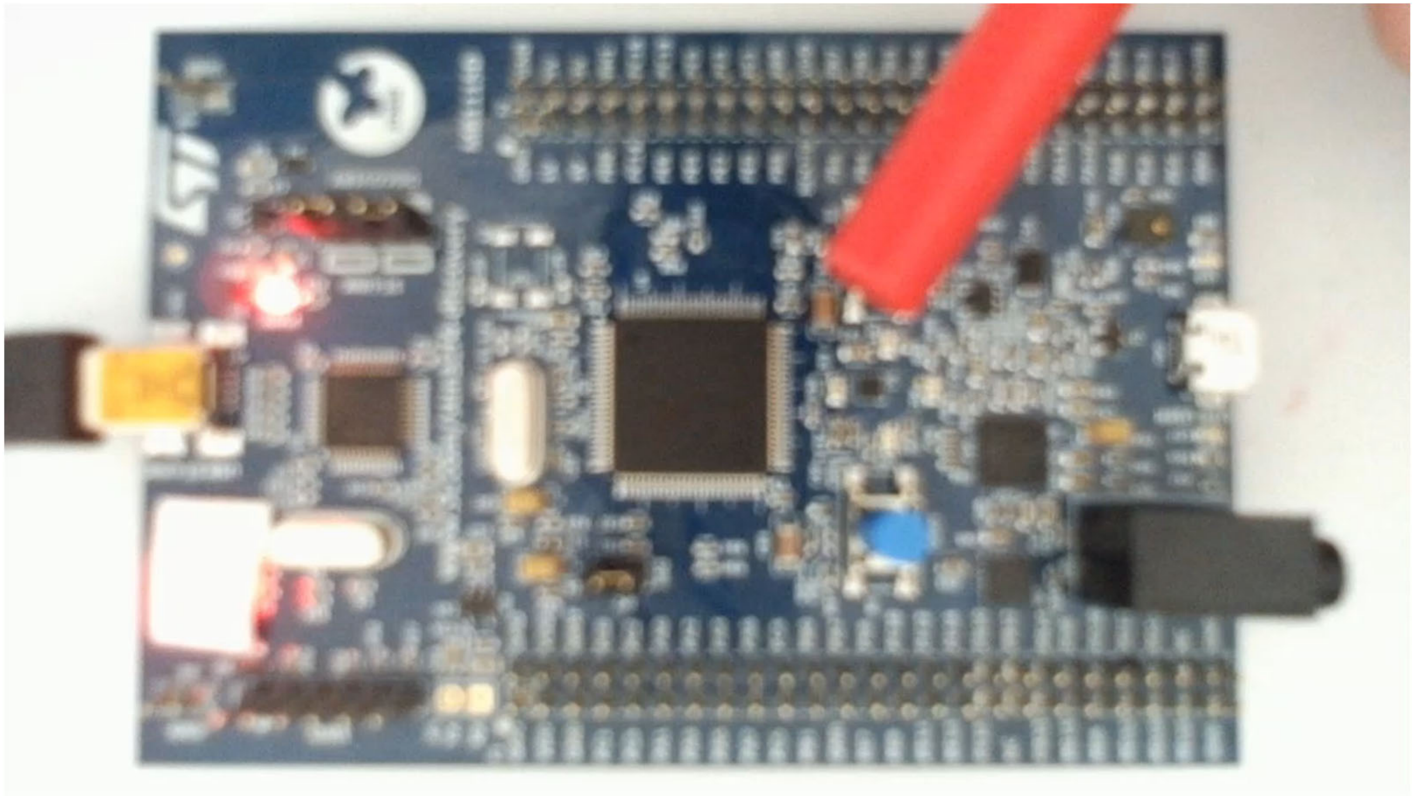
1) The three LEDs will appear to be flashing at their correct rates, and they are.
2) The **Blue** LED never turns on, thus indicating that the Semaphore in protecting the shared resource.

The advantage of encapsulating the semaphore in the **Access_Function()** is the (P) & (V) operations are bound together. Thus preventing one task from performing a (P) and another performing a (V).



Exercise #10 – Use Encapsulation to Improve System Safety & Security

Results…



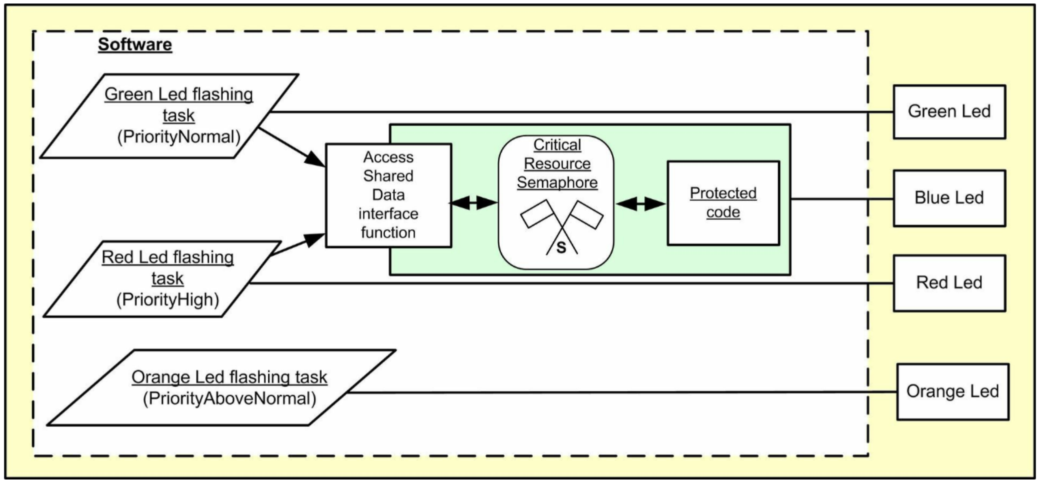
Fundamental purpose of the exercise is to demonstrate the effects of priority inversion in a multitasking design.

EXERCISE #11 – DEMONSTRATE THE EFFECTS OF PRIORITY INVERSION

Exercise #11 – Demonstrate the Effects of Priority Inversion

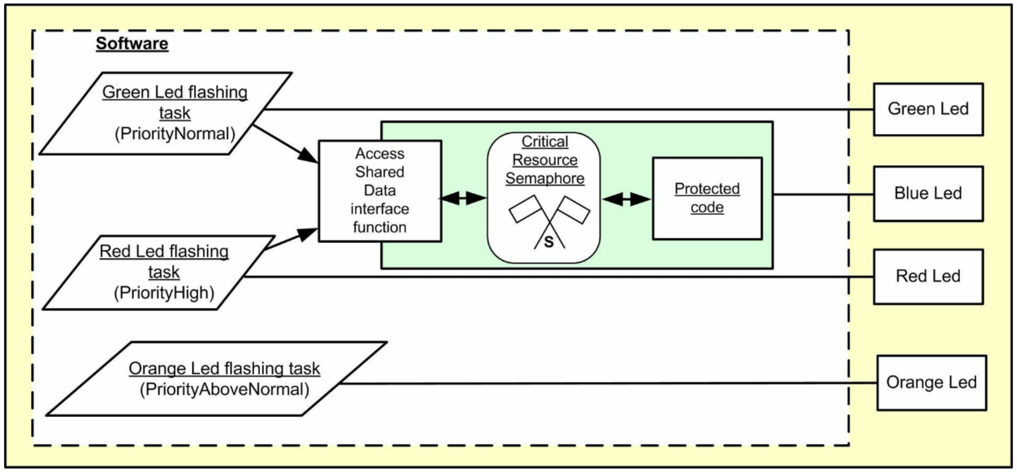
This exercise has ~~four~~ three step:

- #11.1 - Demonstration of the run-time behavior when all tasks execute in an unimpeded fashion: no resource contention and no delays introduced by task interactions.
- #11.2 - Demonstration of mutual exclusion to prevent resource contention (essentially a recap operation but provides reference timing data for the later priority inversion demo).
- #11.3 - Demonstration of the classical priority inversion problem.
- ~~#11.4 - Repeat of #11.3, but extra visual information relating to task execution.~~



Exercise #11.1 – Demonstrate the Effects of Priority Inversion

#11.1 - Demonstration of the run-time behavior when all tasks execute in an unimpeded fashion: no resource contention and no delays introduced by task interactions.



Red LED Task: High Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Red** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Orange LED Task: Above Normal Priority

*Flash **Orange** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Green LED Task: Normal Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Green** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Access_Function()

Take semaphore (P operation)
*Flash **BLUE** LED for (2) seconds @ 10 Hz*
Release semaphore (V operation)

Exercise #11.1 – Demonstrate the Effects of Priority Inversion

Sample Code to Flash LED

40 Cycle = 4 seconds
(20 → 2 seconds)

```
for (int i = 0; i<=40; i++)   // 40 x 100 mSec = 4 Sec
{
  // F= 10 Hz -> T= 100 mSec -> T-High = 50 mSec & T-Low = 50 mSec
  HAL_GPIO_WritePin(GPIOD, [illegible] GPIO_PIN_SET);
  for (int i = 0; i<75000; i++);  // ~ 50 mSeconds
  HAL_GPIO_WritePin(GPIOD, [illegible] GPIO_PIN_RESET);
  for (int i = 0; i<75000; i++);  // ~ 50 mSeconds
}
```

1 Cycle @ 10 Hz

Insert LED Here

Red LED Task: High Priority

- *Access the shared data { i.e. call **Access_Function()** }*
- *Flash **Red** LED for (4) seconds @ 10 Hz*
- *Suspend Task { **vTaskSuspend(NULL)** }*

Orange LED Task: Above Normal Priority

- *Flash **Orange** LED for (4) seconds @ 10 Hz*
- *Suspend Task { **vTaskSuspend(NULL)** }*

Green LED Task: Normal Priority

- *Access the shared data { i.e. call **Access_Function()** }*
- *Flash **Green** LED for (4) seconds @ 10 Hz*
- *Suspend Task { **vTaskSuspend(NULL)** }*

Access_Function()

- *Take semaphore (P operation)*
- *Flash **BLUE** LED for (2) seconds @ 10 Hz*
- *Release semaphore (V operation)*

1) **Red** get scheduled
   - calls *Access_Function()*
2) *Access_Function()*
   - P Operation
   - **Blue** Flashes for 2 Sec
   - V Operation
3) **Red**
   - Flashes for 4 Sec
   - Suspends
4) **Orange** get scheduled
   - **Orange** Flashes for 4 Sec
   - Suspends
5) **Green** get scheduled
   - calls *Access_Function()*
6) *Access_Function()*
   - P Operation
   - **Blue** Flashes for 2 Sec
   - V Operation
7) **Green**
   - Flashes for 4 Sec
   - Suspends

What you should see….

Red LED Task: High Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Red** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Orange LED Task: Above Normal Priority

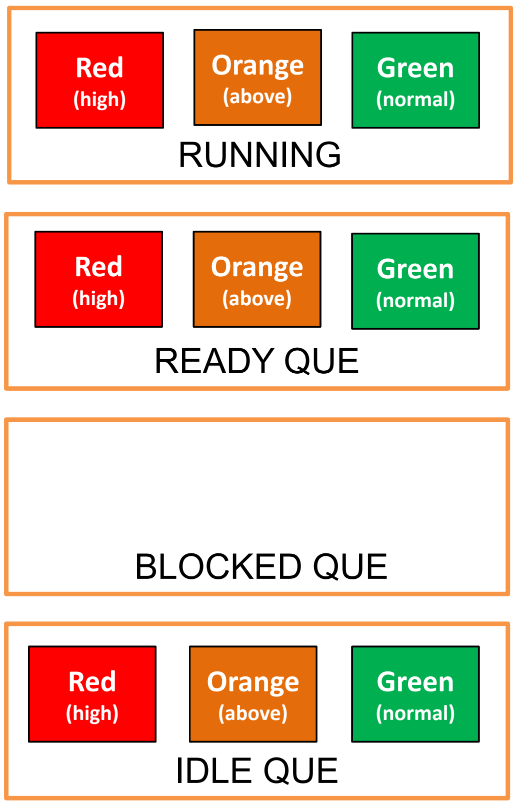
*Flash **Orange** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Green LED Task: Normal Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Green** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

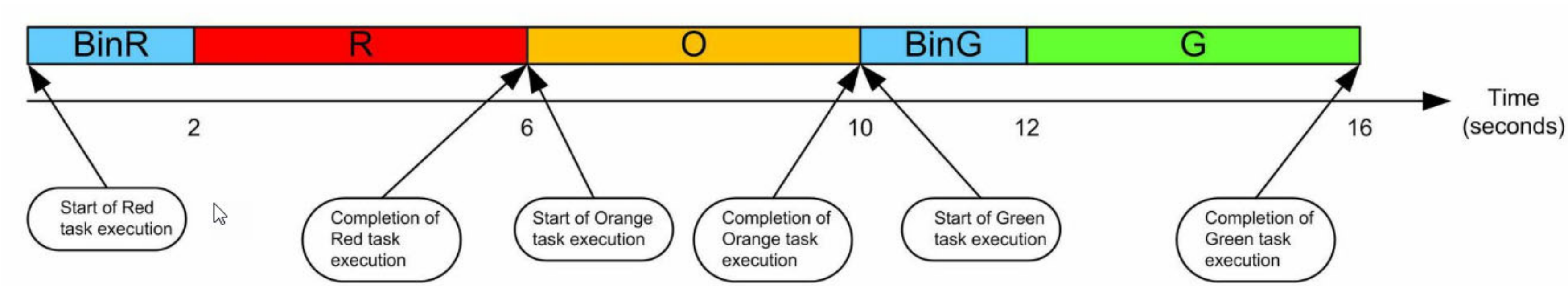
Access_Function()

Take semaphore (P operation)
*Flash **BLUE** LED for (2) seconds @ 10 Hz*
Release semaphore (V operation)



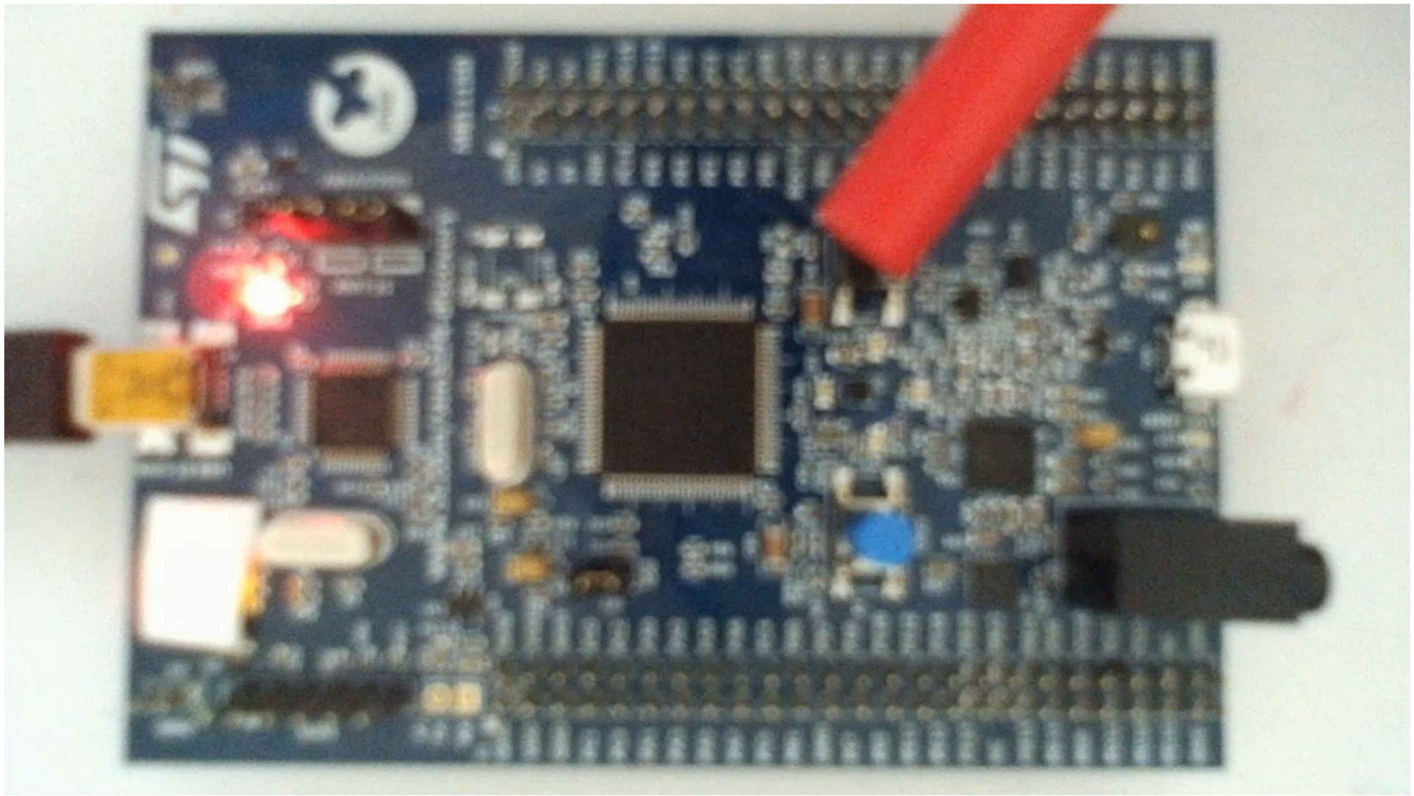
Exercise #11.1 – Demonstrate the Effects of Priority Inversion

- Summary of 11.1
  - All tasks run correctly
  - No resource contention or deadlock occurred
  - Correct prioritization was followed (1st Red, 2nd Orange, 3rd Green)



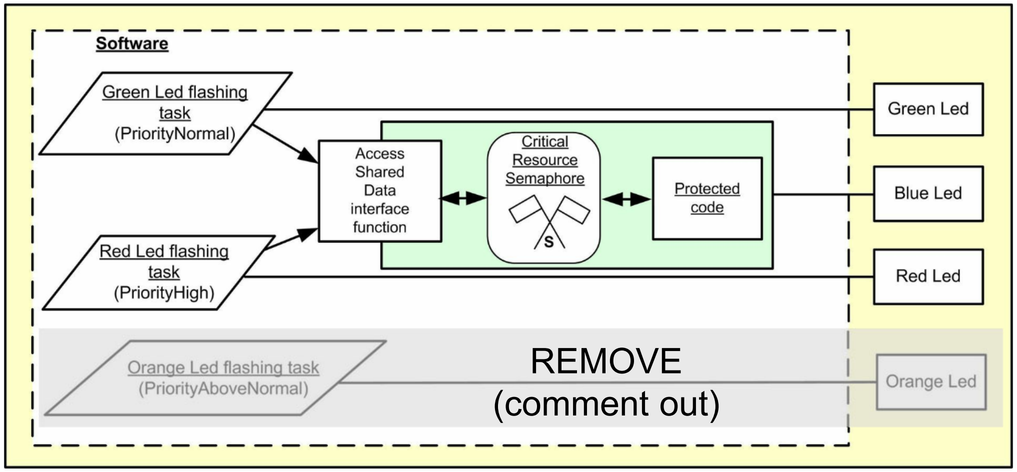
Exercise #11.1 – Demonstrate the Effects of Priority Inversion

Results…



Exercise #11.2 – Demonstrate the Effects of Priority Inversion

#11.2 - Demonstration of mutual exclusion to prevent resource contention (essentially a recap operation but provides reference timing data for the later priority inversion demo).



Red LED Task: High Priority

*Delay (1) Second **{ osDelay(1000) }***
*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Red** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Orange LED Task: Above Normal Priority

*Flash **Orange** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

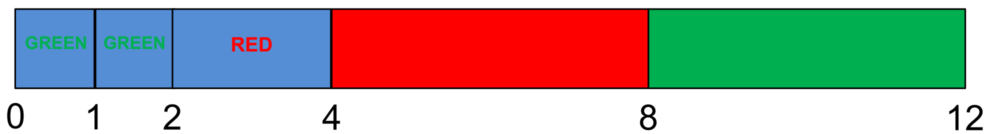
Green LED Task: Normal Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Green** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Access_Function()

Take semaphore (P operation)
*Flash **BLUE** LED for (2) seconds @ 10 Hz*
Release semaphore (V operation)

What you should see….



1) **Red** get scheduled
   - osDelay (Red Blocked)
2) **Green** get scheduled
   - calls *Access_Function()*
3) *Access_Function()*
   - P Operation
   - **Blue** Flashes for 1 Sec
   - **Red** Un-Blocked
   - **Blue** Flashes for 1 Sec
   - V Operation
4) **Green** get preempted
5) **Red** get re-scheduled
   - calls *Access_Function()*
6) *Access_Function()*
   - P Operation
   - **Blue** Flashes for 2 Sec
   - V Operation
7) **Red**
   - Flashes for 4 Sec
   - Suspends
8) **Green** get scheduled
   - Flashes for 4 Sec
   - Suspends

Red LED Task: High Priority

*Delay (1) Second { **osDelay(1000)** }*
*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Red** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Orange LED Task: Above Normal Priority

*Flash **Orange** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Green LED Task: Normal Priority

*Access the shared data { i.e. call **Access_Function()** }*
*Flash **Green** LED for (4) seconds @ 10 Hz*
*Suspend Task { **vTaskSuspend(NULL)** }*

Access_Function()

Take semaphore (P operation)
*Flash **BLUE** LED for (2) seconds @ 10 Hz*
Release semaphore (V operation)

Red (high) | Green (normal)
RUNNING

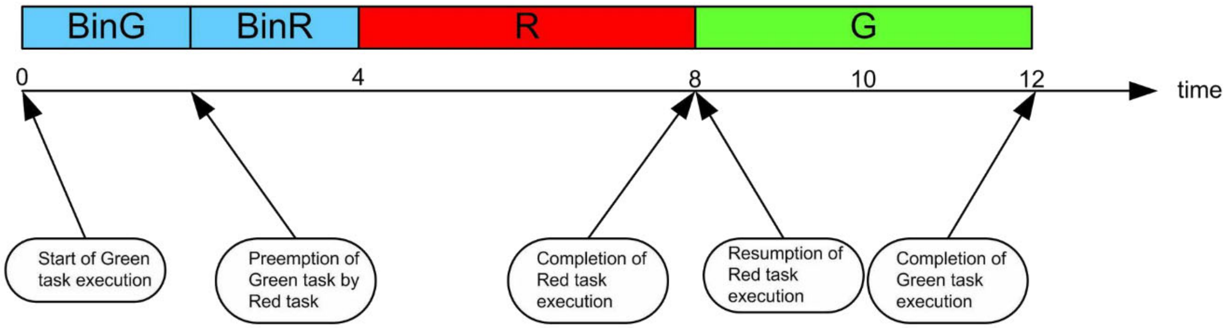
Red (high) | Green (normal)
READY QUE

Red (high)
BLOCKED QUE

Red (high) | Green (normal)
IDLE QUE

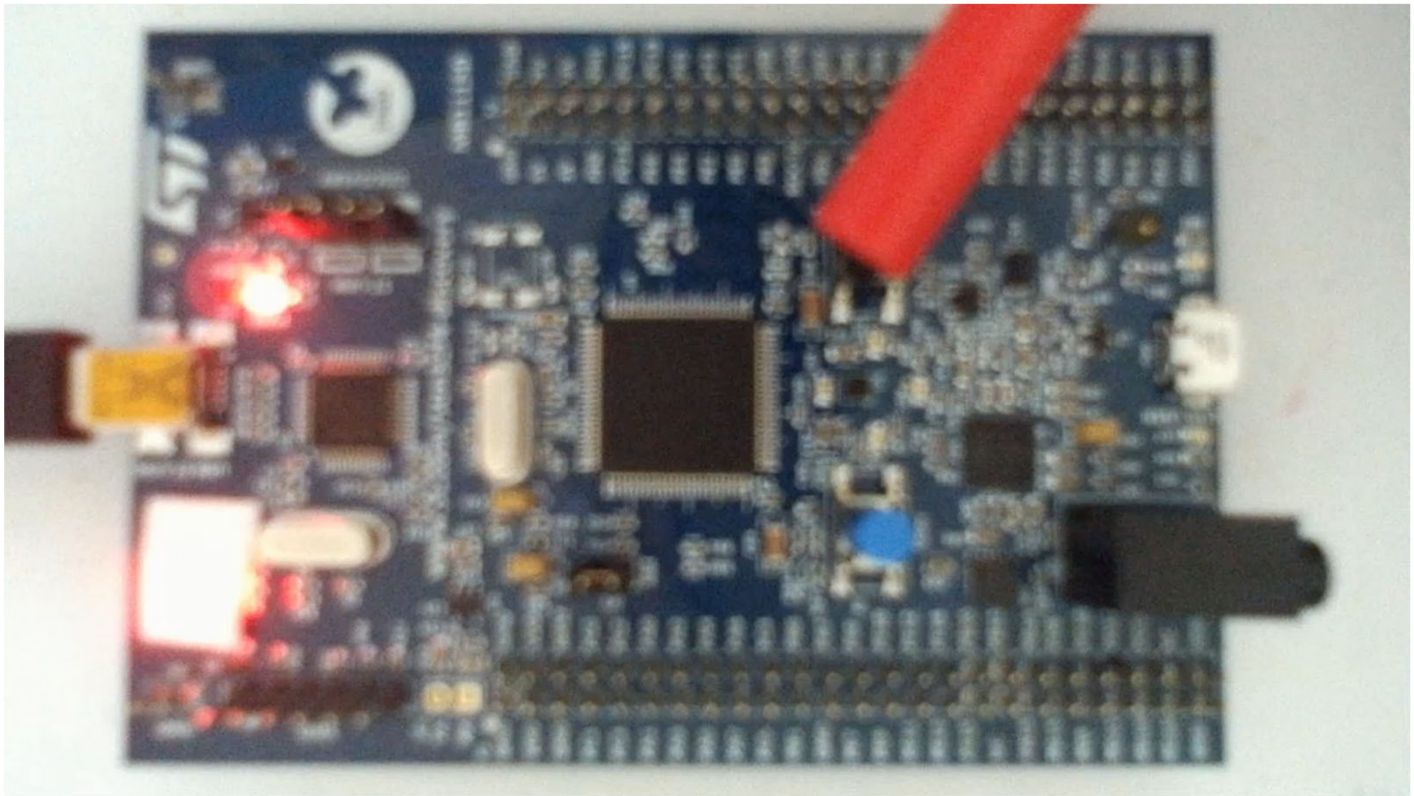
Exercise #11.2 – Demonstrate the Effects of Priority Inversion

- Summary of 11.2
  - All tasks run correctly
  - No resource contention or deadlock occurred
  - Correct prioritization was following (1st Red, 2rd Green)
  - In 11.1 the Red task completed @ t=6. In 11.2 it completed @ t=8. So, the 1-second delay in the Red task caused it's completion to be delayed 2-seconds.
  - Note, with Green running first it appears we have priority inversion, but we don't because Red self-suspended.



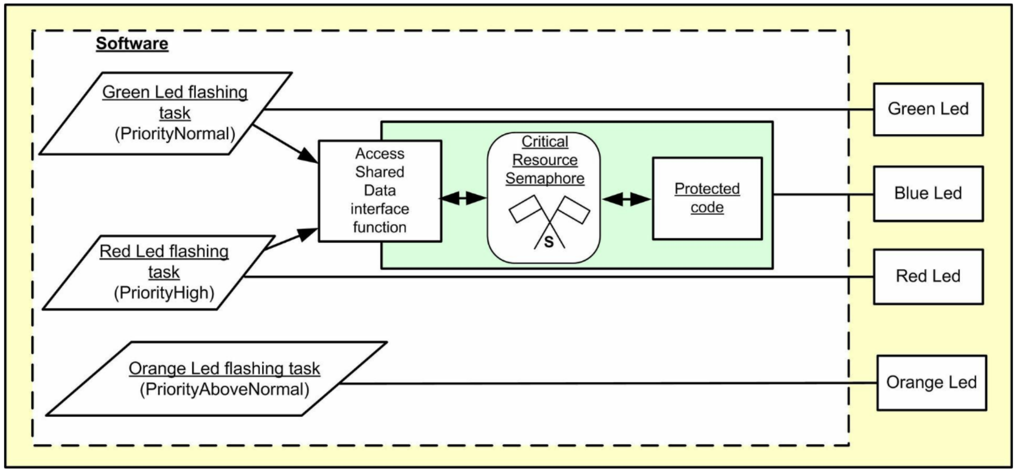
Exercise #11.2 – Demonstrate the Effects of Priority Inversion

Results…



Exercise #11.3 – Demonstrate the Effects of Priority Inversion

#11.3 - Demonstration of the classical priority inversion problem.



Red LED Task: High Priority

Delay (1) Second ***{ osDelay(1000) }***
Access the shared data { i.e. call ***Access_Function()*** *}*
Flash ***Red*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Orange LED Task: Above Normal Priority

Delay (1) Second ***{ osDelay(1000) }***
Flash ***Orange*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Green LED Task: Normal Priority

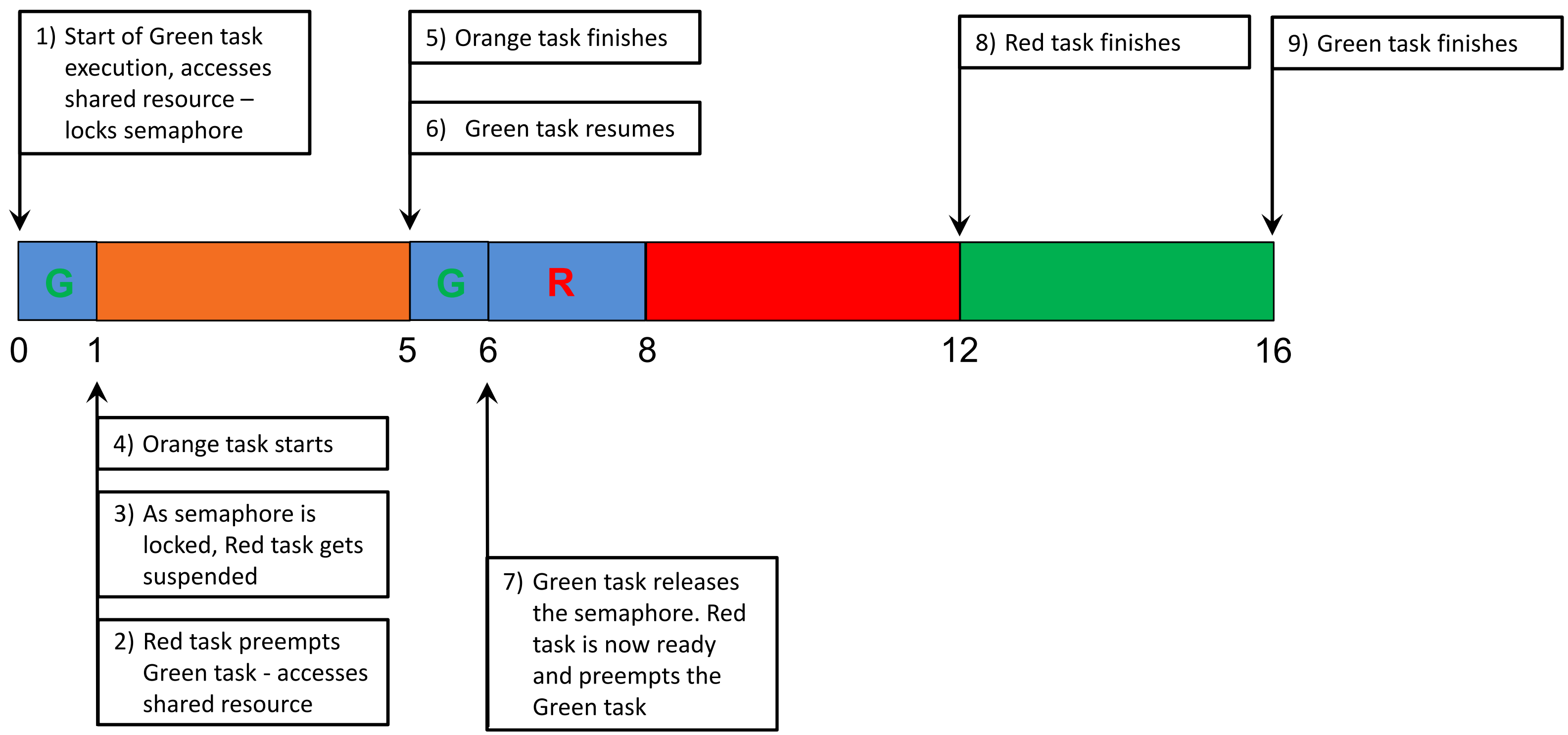
Access the shared data { i.e. call ***Access_Function()*** *}*
Flash ***Green*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Access_Function()

Take semaphore (P operation)
Flash ***BLUE*** *LED for (2) seconds @ 10 Hz*
Release semaphore (V operation)

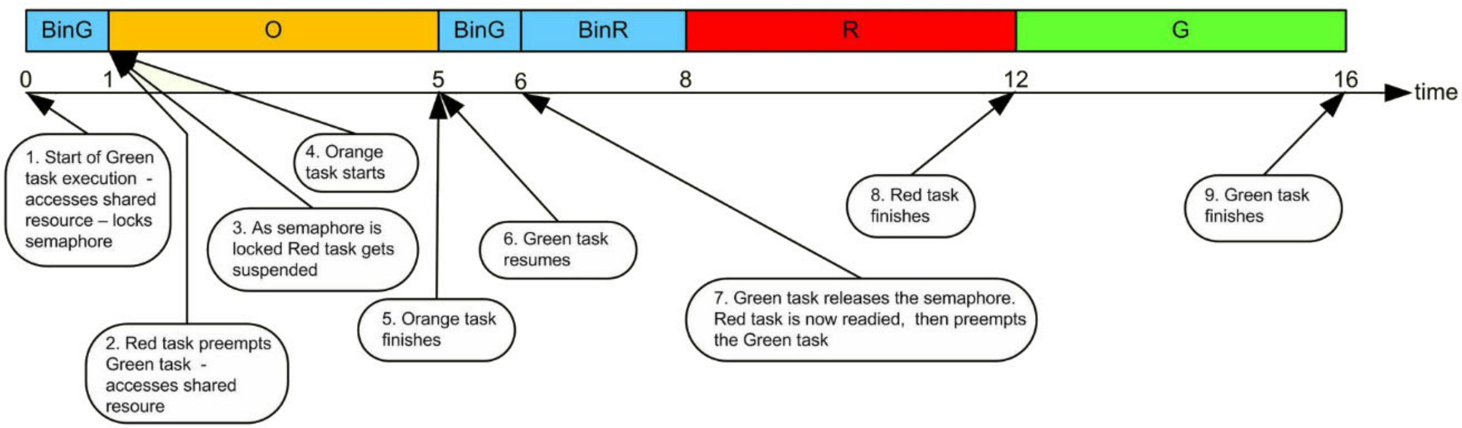
Exercise #11.3 – Demonstrate the Effects of Priority Inversion

What you should see….



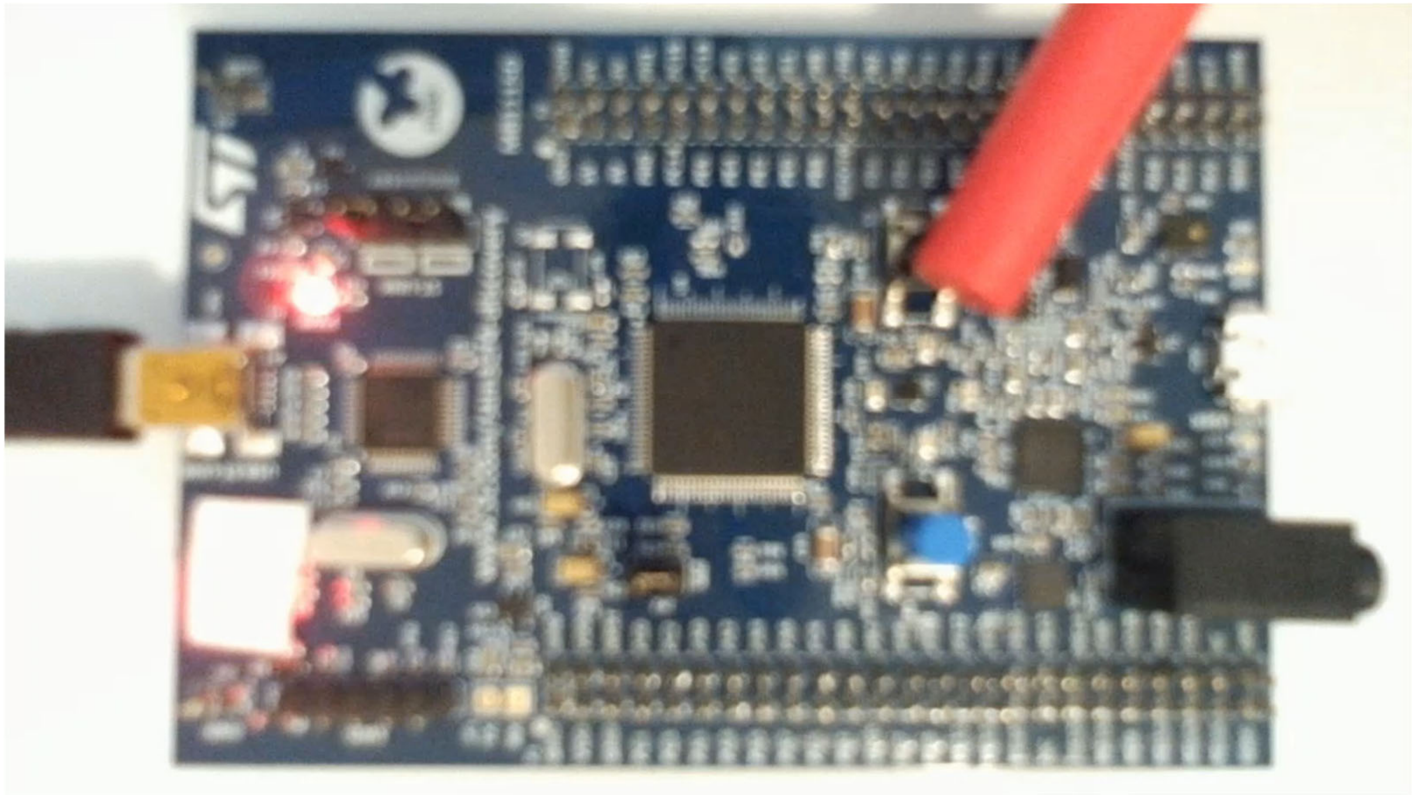
Exercise #11.3 – Demonstrate the Effects of Priority Inversion

- Summary of 11.3
  - This example demonstrates priority inversion.
  - The higher priority Red task completed AFTER the lower priority task Orange.
  - In 11.1 the Red task completed @ t=6. In 11.3 it completed @ t=12. So, the 1-second delay in the Red task, along with the priority inversion with the Orange task, caused it's completion to be delayed 6-seconds. TWICE A LONG!



<u>Exercise #11.3</u> – Demonstrate the Effects of Priority Inversion

Results…



Fundamental purpose of the exercise is to demonstrate how priority inversion can be eliminated by using priority inheritance.

EXERCISE #12 – ELIMINATE PRIORITY INVERSION BY PRIORITY INHERITANCE

Exercise #12 – Eliminate Priority Inversion by Priority Inheritance

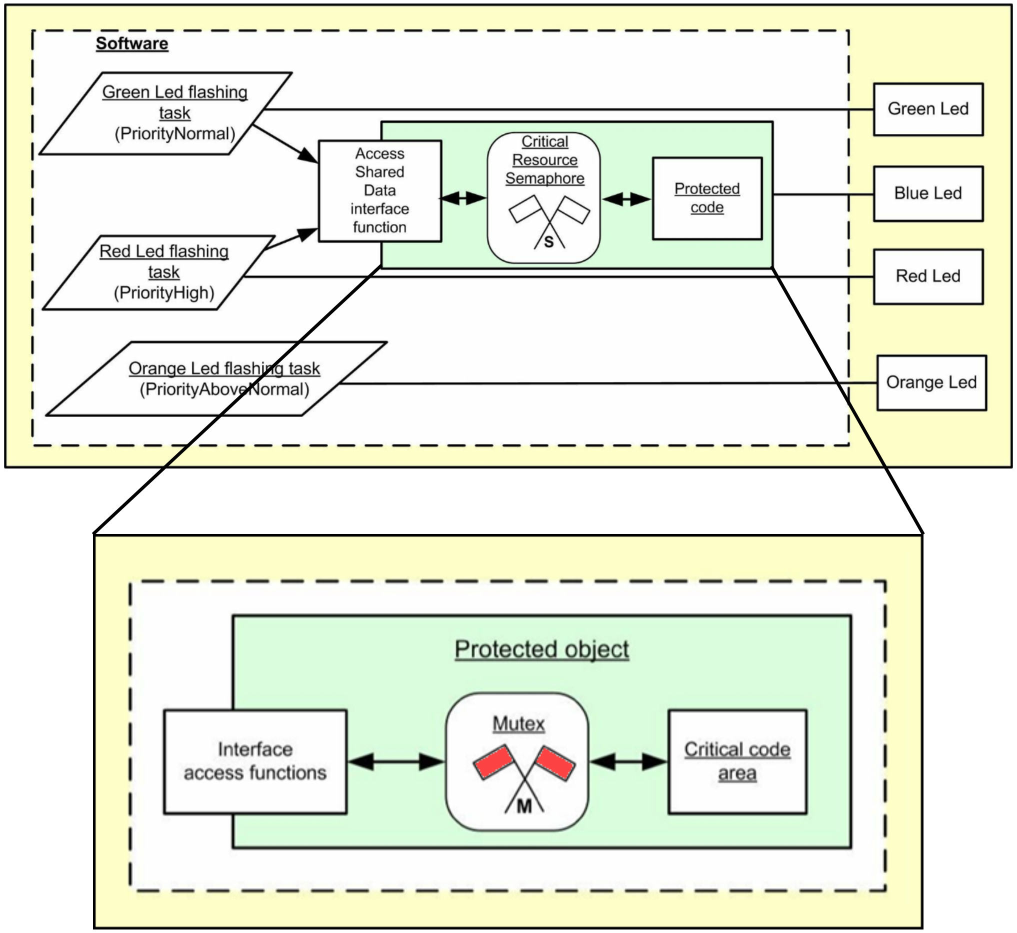
- **Priority Inheritance :** If a **lower** priority task (T_L) holds a resource by means of a MUTEX and a **higher** priority task (T_H) tries to preempt the **lower** priority task, the priority of the **lower** priority task will temporarily be raised to the same priority of the **higher** priority task. As soon as a task switch occurs, T_L will revert back to it's original priority.

  The results of this **priority inheritance** is:

  1) While the higher priority task T_H will still be temporarily blocked, no other tasks with lower priority than T_H can be scheduled. Thus priority inversion will be prevented.
  2) As soon as the lower priority task T_L releases the resource (i.e. MUTEX unlock) and a task switch occurs, the highest priority task that is ready, presumably $T_{H,}$ will get scheduled because T_L has reverted back to its original, lower, priority.
  3) Note, while the lower priority task T_L is still running before the higher priority task $T_{H,}$ it will only run until it's finished with the shared resource.

- In freeRTOS, priority inheritance is implicitly implemented with a standard MUTEX.
- There is nothing special that needs to be done, simply protect the shared resource with a MUTEX.

Exercise #12 – Eliminate Priority Inversion by Priority Inheritance



Red LED Task: High Priority

Delay (1) Second ***{ osDelay(1000) }***
Access the shared data { i.e. call ***Access_Function()*** *}*
Flash ***Red*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Orange LED Task: Above Normal Priority

Delay (1) Second ***{ osDelay(1000) }***
Flash ***Orange*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Green LED Task: Normal Priority

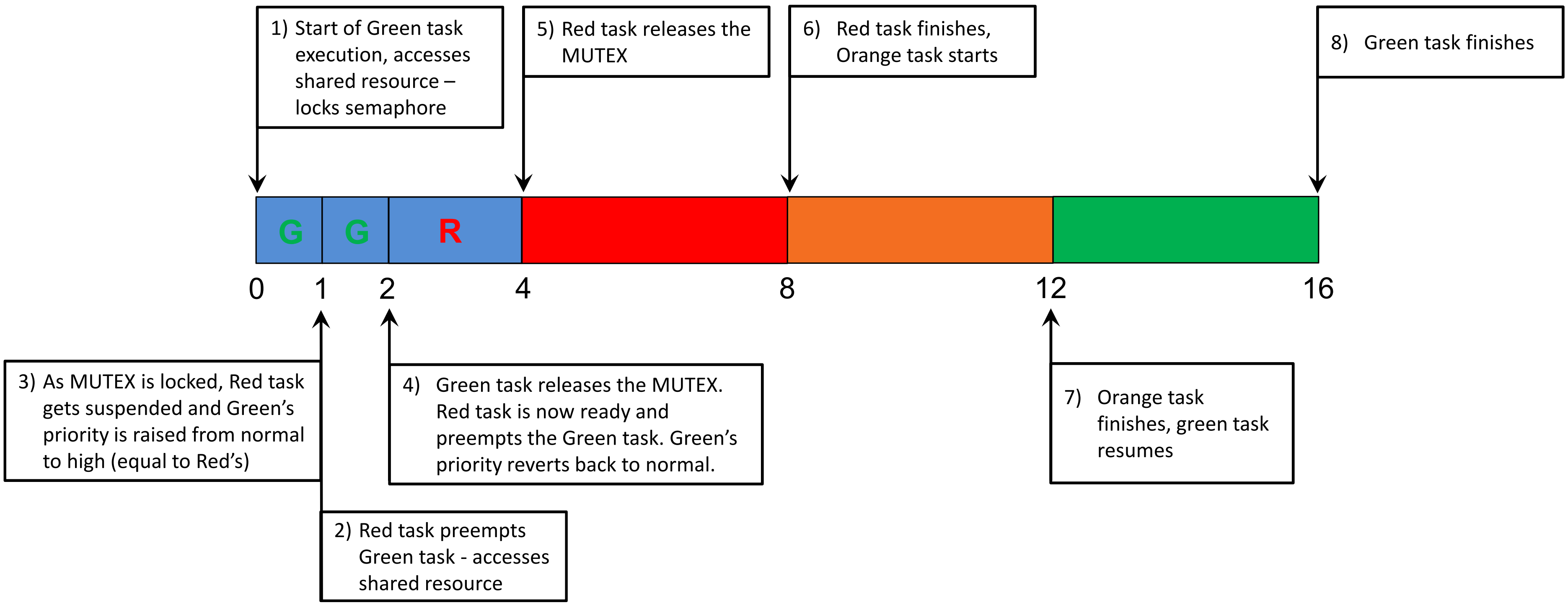
Access the shared data { i.e. call ***Access_Function()*** *}*
Flash ***Green*** *LED for (4) seconds @ 10 Hz*
Suspend Task { ***vTaskSuspend(NULL)*** *}*

Access_Function()

Take MUTEX (lock operation)
Flash ***BLUE*** *LED for (2) seconds @ 10 Hz*
Release MUTEX (unlock operation)

Exercise #12 – Eliminate Priority Inversion by Priority Inheritance

What you should see….



Exercise #12 – Eliminate Priority Inversion by Priority Inheritance

- Summary of 12
  - This example demonstrates priority inheritance
  - All tasks run correctly
  - No resource contention or deadlock occurred
  - Correct prioritization was followed (1st Red, 2nd Orange, 3rd Green)

Exercise #12 – Eliminate Priority Inversion by Priority Inheritance

Results…