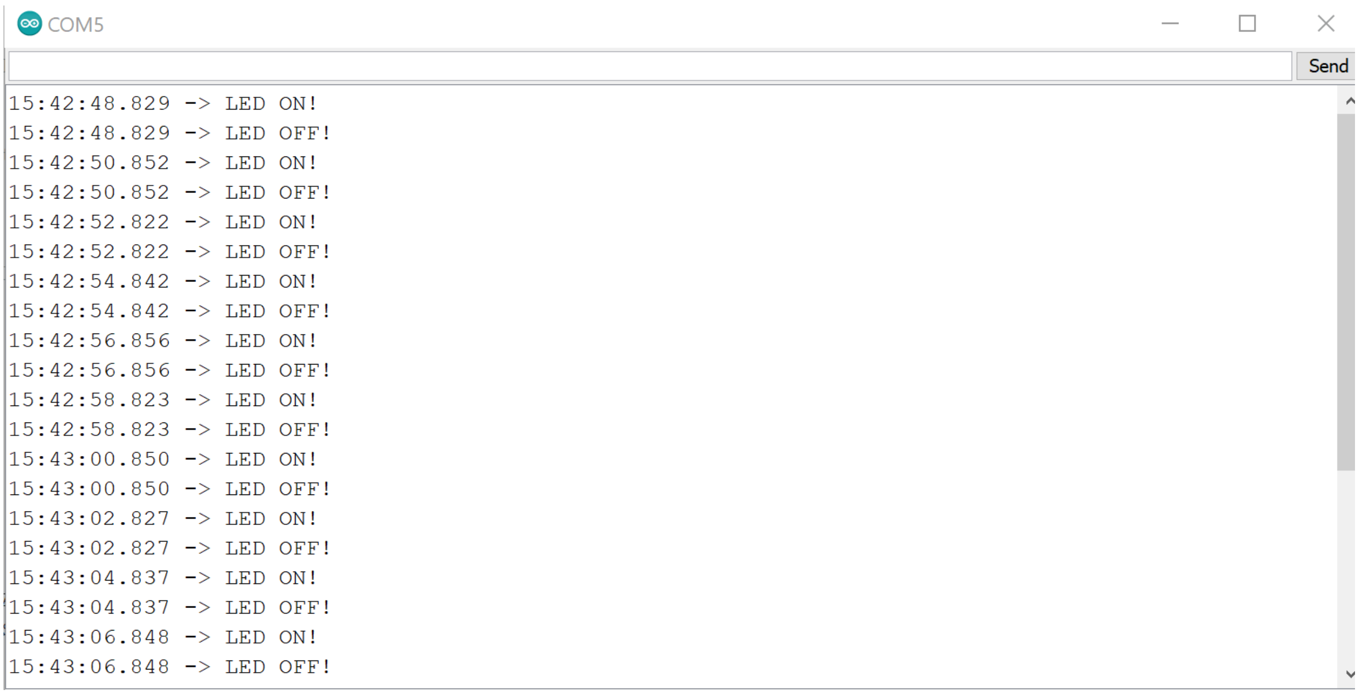
Exercise 2:

2.1

Upon observation, the LED fails to turn on completely with glitches of light. Furthermore, it is observed that the terminal display both 'LED ON!' and 'LED OFF!' messages simultaneously as shown in Figure (1). This is because both led_ON and led_OFF tasks are assigned with the same priority, same delay period without using synchronisation mechanisms. In FreeRTOS, tasks with the same priority with be scheduled in a round-robin fashion. In this case, both tasks are assigned with the same priority and both tasks will be in the ready state at the same time. Without synchronisation mechanisms, both tasks will execute at the same time, which causes erratic behaviour.

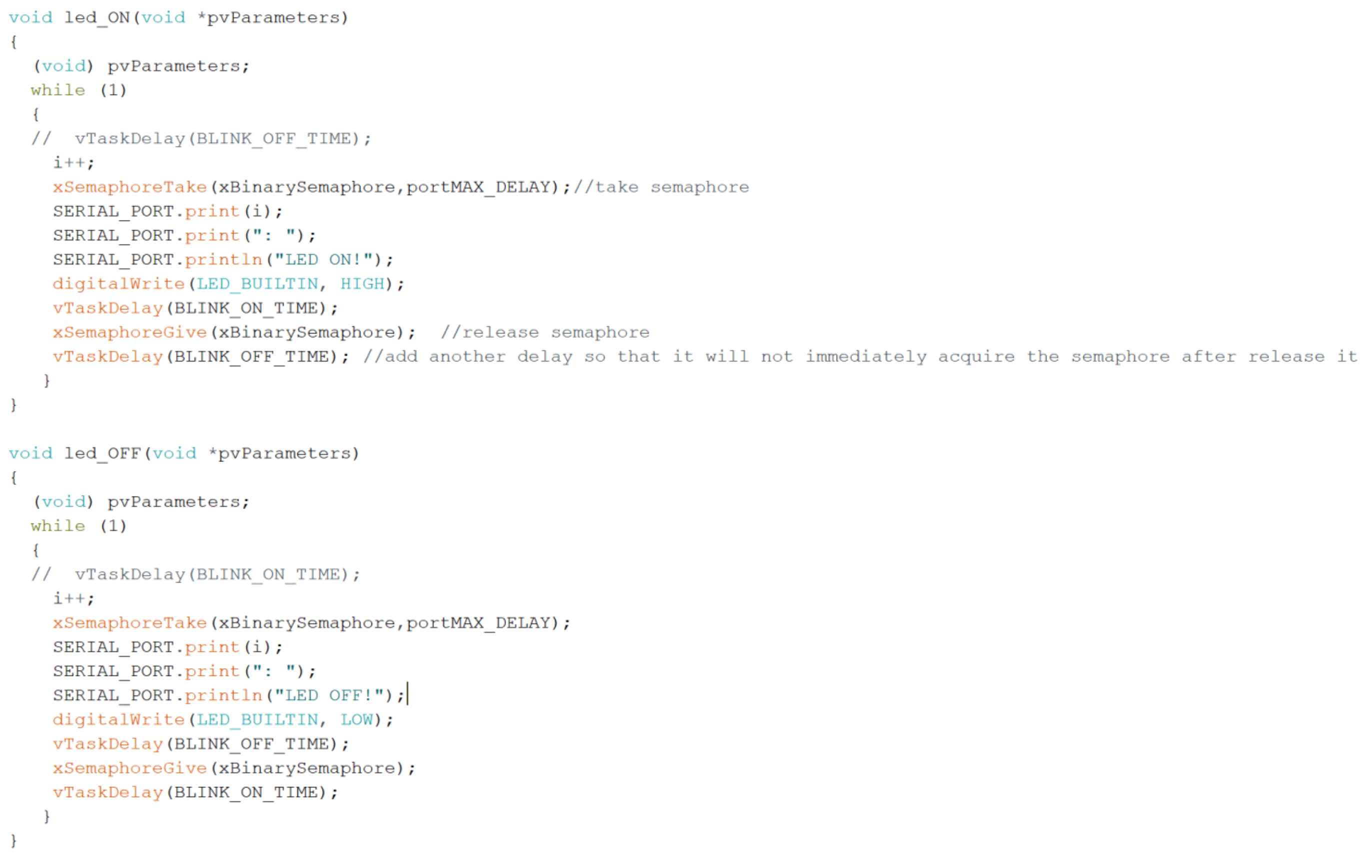


```
15:42:48.829 -> LED ON!
15:42:48.829 -> LED OFF!
15:42:50.852 -> LED ON!
15:42:50.852 -> LED OFF!
15:42:52.822 -> LED ON!
15:42:52.822 -> LED OFF!
15:42:54.842 -> LED ON!
15:42:54.842 -> LED OFF!
15:42:56.856 -> LED ON!
15:42:56.856 -> LED OFF!
15:42:58.823 -> LED ON!
15:42:58.823 -> LED OFF!
15:43:00.850 -> LED ON!
15:43:00.850 -> LED OFF!
15:43:02.827 -> LED ON!
15:43:02.827 -> LED OFF!
15:43:04.837 -> LED ON!
15:43:04.837 -> LED OFF!
15:43:06.848 -> LED ON!
15:43:06.848 -> LED OFF!
```

Figure 1: Terminal Output of exercise 2 when synchronisation is not implemented

2.2

To resolve this issue, synchronisation mechanisms such as semaphores or mutexes can be implemented. In this case, semaphores are implemented to ensure the execution of one task after the completion of the other task, thereby ensuring that the LED state changes in the predefined manner. Semaphore is a counter that is used to control access to share resources such as the LED and serial output. This ensures that both led_ON and led_OFF tasks will work in a synchronised manner such that the led_OFF task will only execute after the completion of the led_ON task. When a task tries to execute, it will call the xSemaphoreTake function. If the semaphore's value is greater than zero, the task will proceed and decrement the semaphore. If the semaphore value is 0, it will wait until the semaphore value becomes greater than zero. When a task finish execution, it will call the xSemaphoreGive function and increment the semaphore value, allowing other tasks to execute. The semphr.h library is used to implement semaphore. A delay introduced at the end of each task to prevent the same task from immediately reacquiring it. As shown in Figure(3), the LED blink on and off every two seconds, which matches with exercise 1.

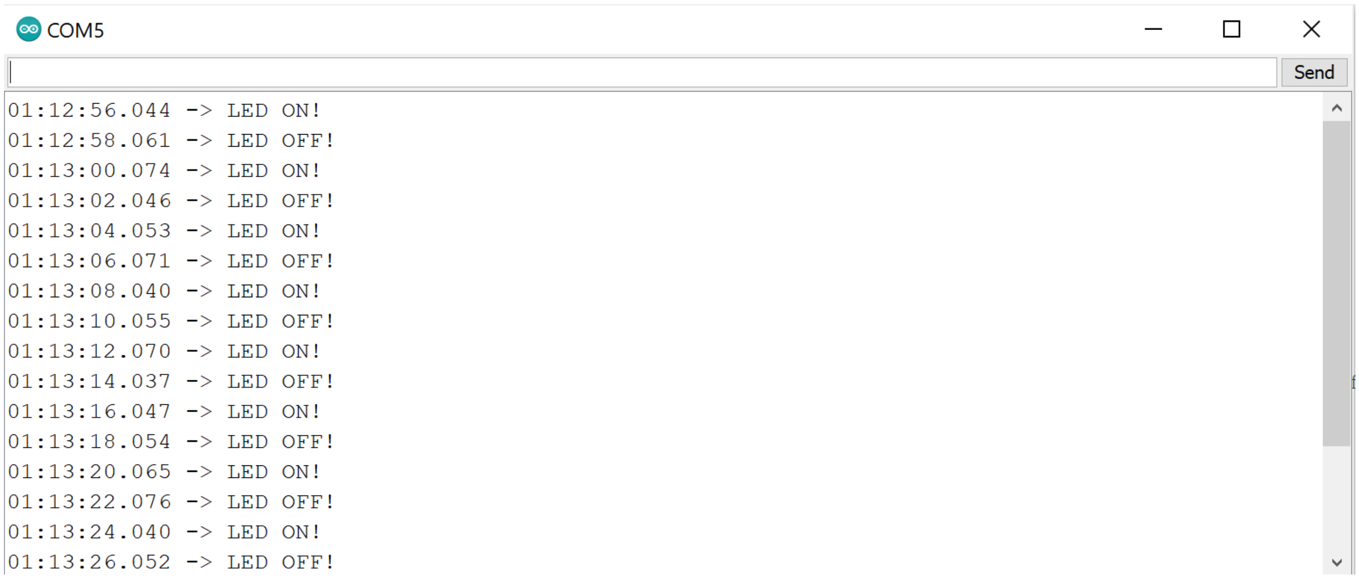


```
void led_ON(void *pvParameters)
{
  (void) pvParameters;
  while (1)
  {
  //  vTaskDelay(BLINK_OFF_TIME);
    i++;
    xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);//take semaphore
    SERIAL_PORT.print(i);
    SERIAL_PORT.print(": ");
    SERIAL_PORT.println("LED ON!");
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay(BLINK_ON_TIME);
    xSemaphoreGive(xBinarySemaphore);  //release semaphore
    vTaskDelay(BLINK_OFF_TIME); //add another delay so that it will not immediately acquire the semaphore after release it
   }
}

void led_OFF(void *pvParameters)
{
  (void) pvParameters;
  while (1)
  {
  //  vTaskDelay(BLINK_ON_TIME);
    i++;
    xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);
    SERIAL_PORT.print(i);
    SERIAL_PORT.print(": ");
    SERIAL_PORT.println("LED OFF!");
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay(BLINK_OFF_TIME);
    xSemaphoreGive(xBinarySemaphore);
    vTaskDelay(BLINK_ON_TIME);
   }
}
```

Figure 2: Implementation of led_ON and led_OFF functions

Table 1: Semaphore Function Description

Function name	Usage
xSemaphoreCreateBinary	To create a binary semaphore which has two states (0 and 1)
xSemaphoreTake	To take or wait for a semaphore. Task will take the semaphore and proceed with its execution if the semaphore is available. If the semaphore has already been taken, the task will wait until the semaphore is released and become available.
xSemaphoreGive	To release semaphore after finish task execution



```
01:12:56.044 -> LED ON!
01:12:58.061 -> LED OFF!
01:13:00.074 -> LED ON!
01:13:02.046 -> LED OFF!
01:13:04.053 -> LED ON!
01:13:06.071 -> LED OFF!
01:13:08.040 -> LED ON!
01:13:10.055 -> LED OFF!
01:13:12.070 -> LED ON!
01:13:14.037 -> LED OFF!
01:13:16.047 -> LED ON!
01:13:18.054 -> LED OFF!
01:13:20.065 -> LED ON!
01:13:22.076 -> LED OFF!
01:13:24.040 -> LED ON!
01:13:26.052 -> LED OFF!
```

Figure 3: Terminal Output of Exercise 2 after implementation of semaphore

Exercise 3:

```
if(lValueToSend == 200){
 vTaskDelay(pdMS_TO_TICKS(1000));
}
```

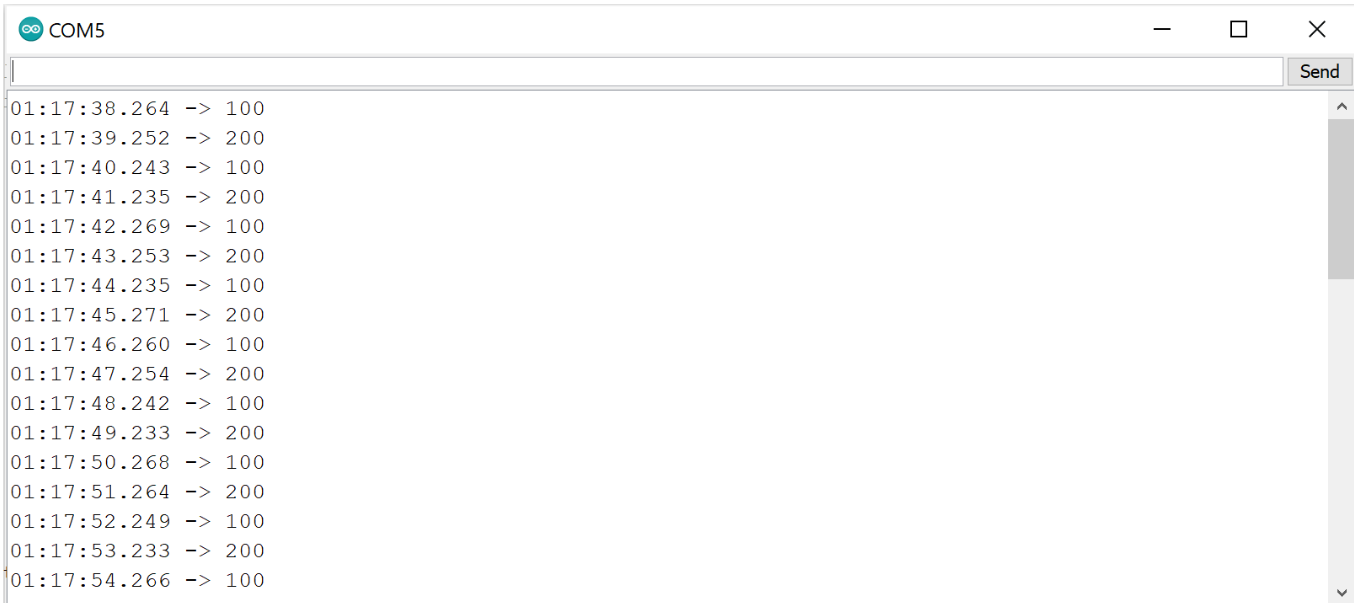
```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 2000 ) );
```

The vSenderTask is modified such that a delay of 1second is added if the value to send is 200. This prevents the value of 100 and 200 from sending simultaneously. As a result, this ensures that the value 100 will be sent to the queue first before sending value 200. In addition, the second argument of vTaskDelayUntil function is modified to 2000, setting the task period to 2 seconds.

```
if(lReceivedValue == 100){
 digitalWrite(LED_BUILTIN,HIGH);
}
else if(lReceivedValue == 200){
 digitalWrite(LED_BUILTIN,LOW);
}
```

The vReceiverTask is modified such that when the received value from queue is 100, the LED will turn on, whereas the LED will turn off when the received value from queue is 200.

As shown in Figure (4), the value 100 and 200 is displayed one by one in the terminal every second.



```
01:17:38.264 -> 100
01:17:39.252 -> 200
01:17:40.243 -> 100
01:17:41.235 -> 200
01:17:42.269 -> 100
01:17:43.253 -> 200
01:17:44.235 -> 100
01:17:45.271 -> 200
01:17:46.260 -> 100
01:17:47.254 -> 200
01:17:48.242 -> 100
01:17:49.233 -> 200
01:17:50.268 -> 100
01:17:51.264 -> 200
01:17:52.249 -> 100
01:17:53.233 -> 200
01:17:54.266 -> 100
```

Figure 4: Terminal Output of Exercise 3

Exercise 4:

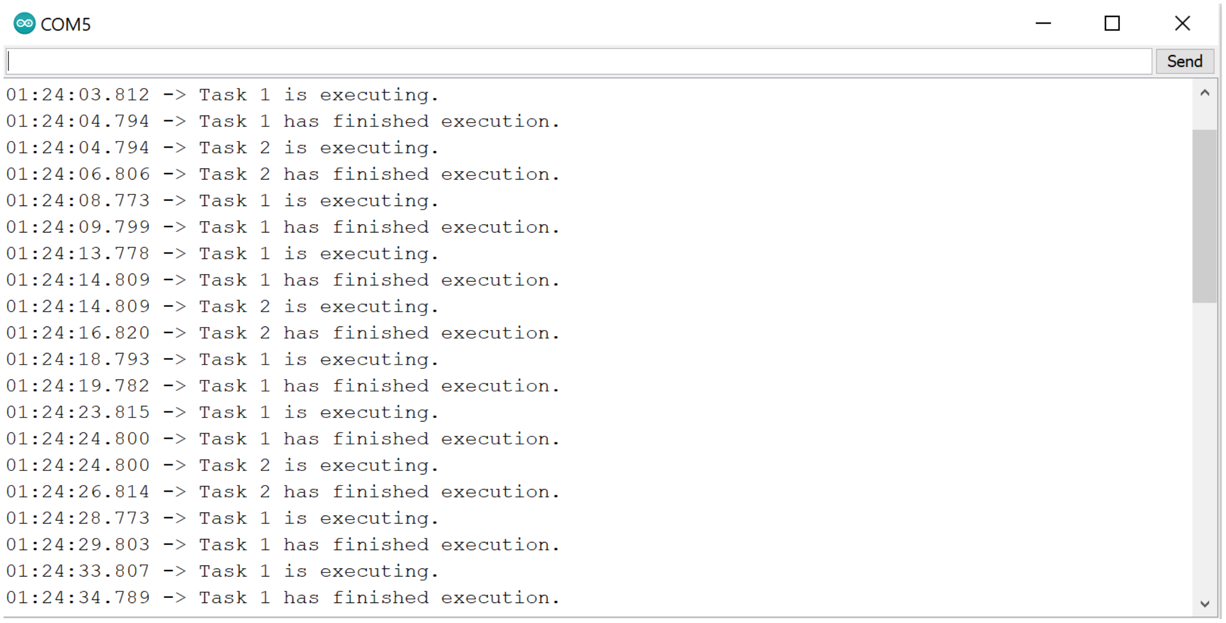
```
void vTask1(void *pvParameters) {

  TickType_t xLastWakeTime = xTaskGetTickCount();
  const TickType_t xTask1Period = pdMS_TO_TICKS(5000); // 5 seconds
  const TickType_t xTask1ExecutionTime = pdMS_TO_TICKS(1000);// execution time is 1 second

  for (;;) {

    TickType_t previousTime = xTaskGetTickCount();
    vPrintString("Task 1 is executing.");

    //light LED up for 1 second execution time
    while (xTaskGetTickCount() - previousTime < xTask1ExecutionTime){

      digitalWrite(LED_BUILTIN,HIGH);
    }

    digitalWrite(LED_BUILTIN, LOW);  // Turn off LED
    vPrintString("Task 1 has finished execution.");
    vTaskDelayUntil(&xLastWakeTime, xTask1Period); //repeat every 5 seconds

  }
}
```

```
void vTask2(void *pvParameters) {

  TickType_t xLastWakeTime = xTaskGetTickCount();
  const TickType_t xTask2Period = pdMS_TO_TICKS(10000); // 10 seconds
  const TickType_t xTask2ExecutionTime = pdMS_TO_TICKS(2000);// 2 seconds

  for (;;) {

    vTaskDelay(1000);

    TickType_t previousTime = xTaskGetTickCount();
    vPrintString("Task 2 is executing.");
    //light up LED for 2 seconds execution time
    while (xTaskGetTickCount() - previousTime < xTask2ExecutionTime){
      digitalWrite(LED_BUILTIN,HIGH);
    }

    digitalWrite(LED_BUILTIN, LOW);  // Turn off LED
    vPrintString("Task 2 has finished execution.");
    vTaskDelayUntil(&xLastWakeTime, xTask2Period);   //repeat every 10 seconds
  }

}
```

A rate monotonic scheduler assigned priority to different tasks based on their period. Task with smaller period will be assigned higher priority. In this case, the first task has a greater priority compared to the second task as it has smaller period.

As shown in the code snippets above, two tasks are created with different period and execution time. The period and execution time of the tasks are defined using the pdMS_TO_TICKS function. The pdMS_TO_TICKS function is used to convert time in millisecond to the equivalent ticks used in FreeRTOS. During the execution time, both tasks are designed to turn the LED on. This is implemented using a while loop that check if the execution time has been reached. A delay of 1s is added at the beginning of task 2 to delay and suspend its execution, so that task 1 will be given higher priority and execute first.

As shown in Figure (5), task 1 will execute and complete its execution before task 2 starts its execution. This demonstrates that shorter period task is given higher priority and execute first. The tasks scheduling is also simulated and verified using Cheddar software as shown in Figure (6).



```
01:24:03.812 -> Task 1 is executing.
01:24:04.794 -> Task 1 has finished execution.
01:24:04.794 -> Task 2 is executing.
01:24:06.806 -> Task 2 has finished execution.
01:24:08.773 -> Task 1 is executing.
01:24:09.799 -> Task 1 has finished execution.
01:24:13.778 -> Task 1 is executing.
01:24:14.809 -> Task 1 has finished execution.
01:24:14.809 -> Task 2 is executing.
01:24:16.820 -> Task 2 has finished execution.
01:24:18.793 -> Task 1 is executing.
01:24:19.782 -> Task 1 has finished execution.
01:24:23.815 -> Task 1 is executing.
01:24:24.800 -> Task 1 has finished execution.
01:24:24.800 -> Task 2 is executing.
01:24:26.814 -> Task 2 has finished execution.
01:24:28.773 -> Task 1 is executing.
01:24:29.803 -> Task 1 has finished execution.
01:24:33.807 -> Task 1 is executing.
01:24:34.789 -> Task 1 has finished execution.
```

Figure 5: Terminal Output of Exercise 4

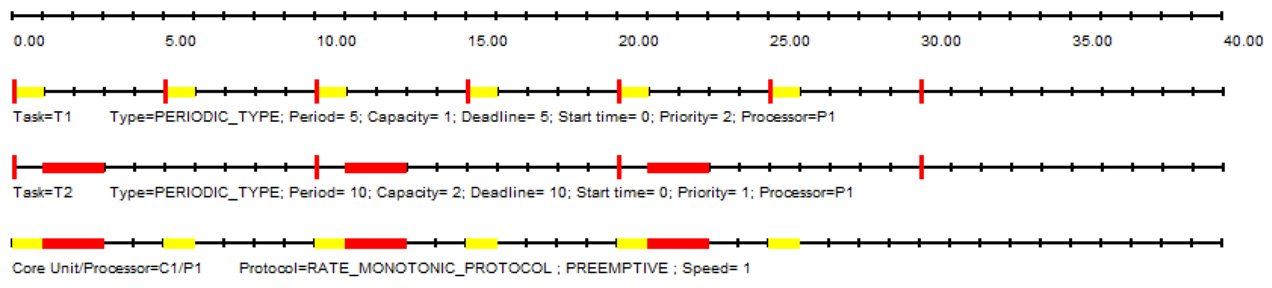


Figure 6: Simulation Output form Cheddar