FreeRTOS application:

Let’s consider this application:

Four tasks T1, T2, T3, T4. Each task sends via UART the message “Hello from task i”, where i is the task number.

T1: Low Priority, Periodicity: 400 ms

T2: Low Priority 1, Periodicity: 300 ms

T3: Low Priority 2, Periodicity: 200 ms

T4: Low Priority 3, Periodicity: 100 ms

Configure the FreeRTOS in the (.ioc) file. We obtain the following configuration:

```
/* Definitions for Task1 */
osThreadId_t Task1Handle;
const osThreadAttr_t Task1_attributes = {
  .name = "Task1",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityLow,
};
/* Definitions for Task2 */
```

Then, we create the task: `Task1Handle = osThreadNew(Task1Function, NULL, &Task1_attributes);`

Then, we start the Kernel: `osKernelStart();`

Here is the definition of the task function:

```
/* USER CODE END Header_Task1Function */
void Task1Function(void *argument)
{
  /* USER CODE BEGIN 5 */
        uint8_t occurence1 = 0;
  /* Infinite loop */
  for(;;)
  {
          osDelay(PERIODICITY_TASK1);
          HAL_UART_Transmit(&huart3, (uint8_t*)RTOS_TaskMsg1_ac, TX_BUFFER_SIZE, TX_MAX_DELAY);
  }
  /* USER CODE END 5 */
}
```

We run the code and we obtain the following result:

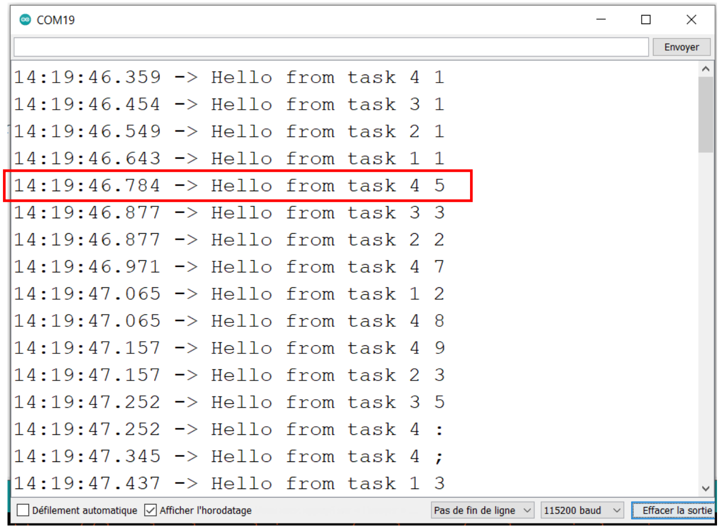

Figure 1: Result of the first essay

Interpretation:

- According to this result we can conclude that the UART itself is not facing any problems: When it begins sending, it is never preempted, and it sends all the data. (Unexpected result).
- Assuming that the data sent includes the number of times a task is executed (the number at the end), and considering the result in the red rectangle, there some messages that are not sent.
  - ⇨ Protection in the function HAL_UART_Transmit.

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)
{
  const uint8_t  *pdata8bits;
  const uint16_t *pdata16bits;
  uint32_t tickstart;

  /* Check that a Tx process is not already ongoing */
  if (huart->gState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return  HAL_ERROR;
    }

    __HAL_LOCK(huart);

    huart->ErrorCode = HAL_UART_ERROR_NONE;
    huart->gState = HAL_UART_STATE_BUSY_TX;

    /* Init tickstart for timeout management */
    tickstart = HAL_GetTick();

    huart->TxXferSize  = Size;
    huart->TxXferCount = Size;

    /* In case of 9bits/No Parity transfer, pData needs to be handled as a uint16_t pointer */
    if ((huart->Init.WordLength == UART_WORDLENGTH_9B) && (huart->Init.Parity == UART_PARITY_NONE))
    {
      pdata8bits  = NULL;
      pdata16bits = (const uint16_t *) pData;
    }
    else
    {
      pdata8bits  = pData;
      pdata16bits = NULL;
    }

    __HAL_UNLOCK(huart);

    while (huart->TxXferCount > 0U)
    {
      if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TXE, RESET, tickstart, Timeout) != HAL_OK)
      {
        return HAL_TIMEOUT;
      }
      if (pdata8bits == NULL)
      {
        huart->Instance->TDR = (uint16_t)(*pdata16bits & 0x01FFU);
        pdata16bits++;
      }
      else
      {
        huart->Instance->TDR = (uint8_t)(*pdata8bits & 0xFFU);
        pdata8bits++;
      }
      huart->TxXferCount--;
    }

    if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TC, RESET, tickstart, Timeout) != HAL_OK)
    {
      return HAL_TIMEOUT;
    }

    /* At end of Tx process, restore huart->gState to Ready */
    huart->gState = HAL_UART_STATE_READY;

    return HAL_OK;
  }
  else
  {
    return HAL_BUSY;
  }
}
```

To remove this protection, we just need to define another function called FEKI_UART_Transmit without any protection.

Let's try it!

```
/* USER CODE END Header_Task1Function */
void Task1Function(void *argument)
{
  /* USER CODE BEGIN 5 */
        uint8_t occurence1 = 0;
  /* Infinite loop */
  for(;;)
  {
          osDelay(PERIODICITY_TASK1);
          FEKI_UART_Transmit(&huart3, (uint8_t*)RTOS_TaskMsg1_ac, TX_BUFFER_SIZE, TX_MAX_DELAY);
  }
  /* USER CODE END 5 */
}
```

We obtain the following result!

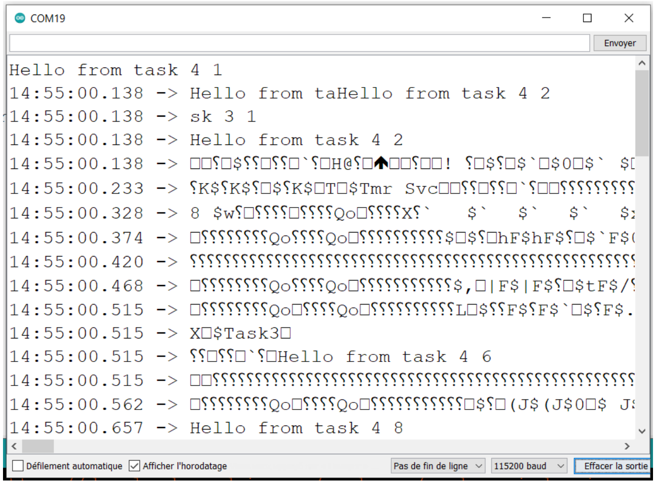

Figure 2: Result without the HAL protection

- ⇨ Logical result!
- ⇨ Need to implement protection using binary semaphore!

Use semaphore:

In our case, UART is a shared resource and the code that sends data is a critical section.

Due to the problems previously mentioned, we have to protect this critical section.

- ⇨ We use binary semaphore.

To do so, we configure and create a binary semaphore, then we use the APIs `osSemaphoreAcquire(Semaphore1Handle, portMAX_DELAY)` and `osSemaphoreRelease(Semaphore1Handle);`.

Code:

```
if (osOK == osSemaphoreAcquire(Semaphore1Handle, portMAX_DELAY))
{
        RTOS_TaskMsg4_ac[18] = (char)(occurence4 + 48);
        FEKI_UART_Transmit(&huart3, (uint8_t*)RTOS_TaskMsg4_ac, TX_BUFFER_SIZE, TX_MAX_DELAY);
        osSemaphoreRelease(Semaphore1Handle);
}
else
{
        /* do nothing */
}
```

We obtain the following result:

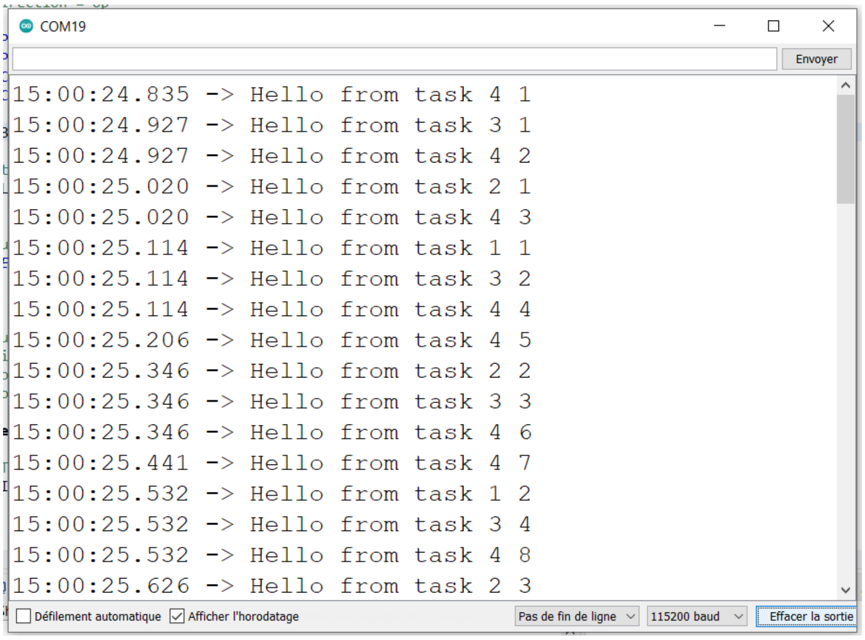

Figure 3: Result after implementing the semaphore

Interpretation:

- After using binary semaphores, we obtain a good enough result that allows us to protect the critical section and have full messages. None of the messages is preempted.