Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

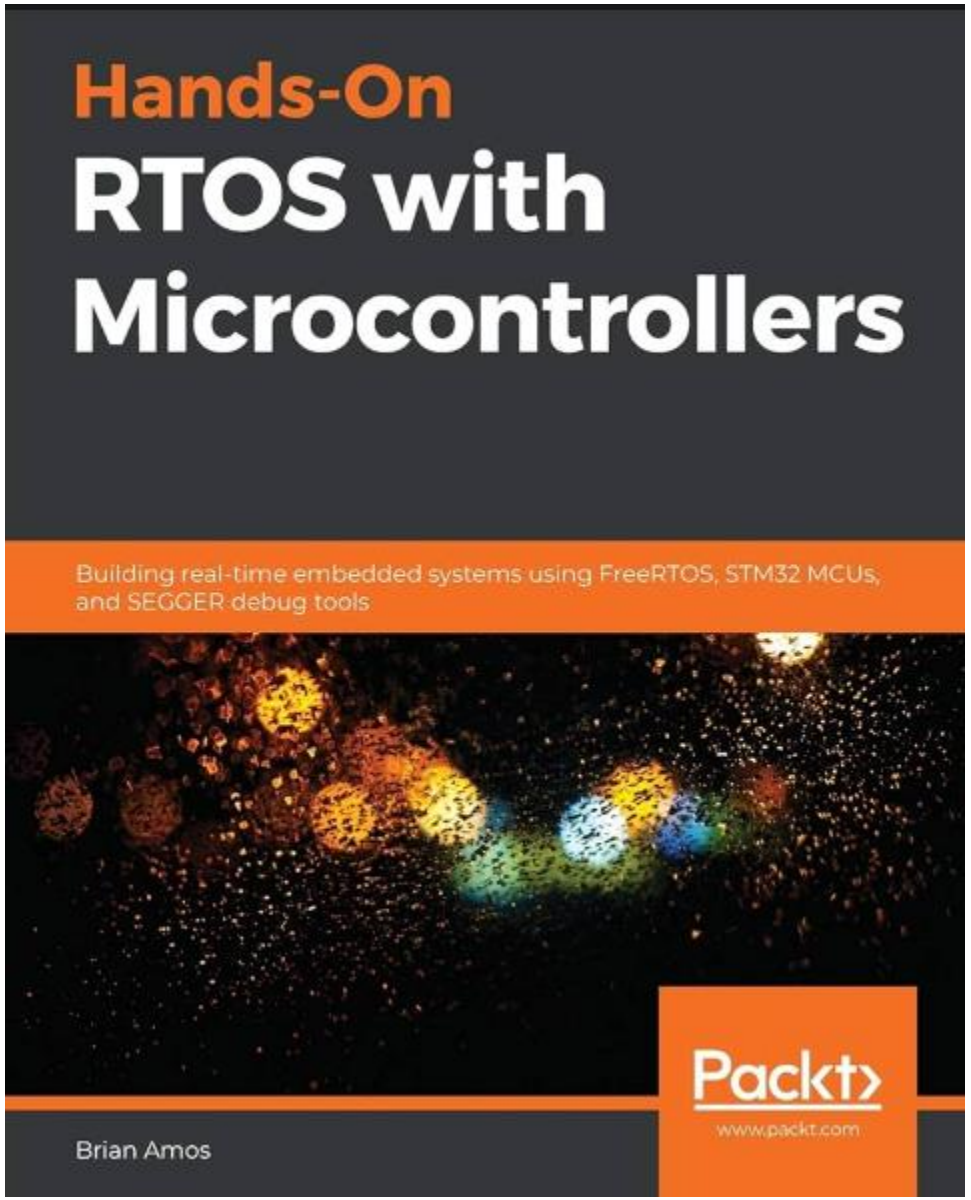
School of Electrical Engineering
and Computer Science

CEG4166/CSI4141/SEG4145 Real Time System Design

Winter 2024

Professor: Mohamed Ali Ibrahim, Ph.D., P. Eng.

Source:


Hands-On
RTOS with
Microcontrollers
Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools
Brian Amos
Packt
www.packt.com

Chapitre 14: Pilotes et ISRs

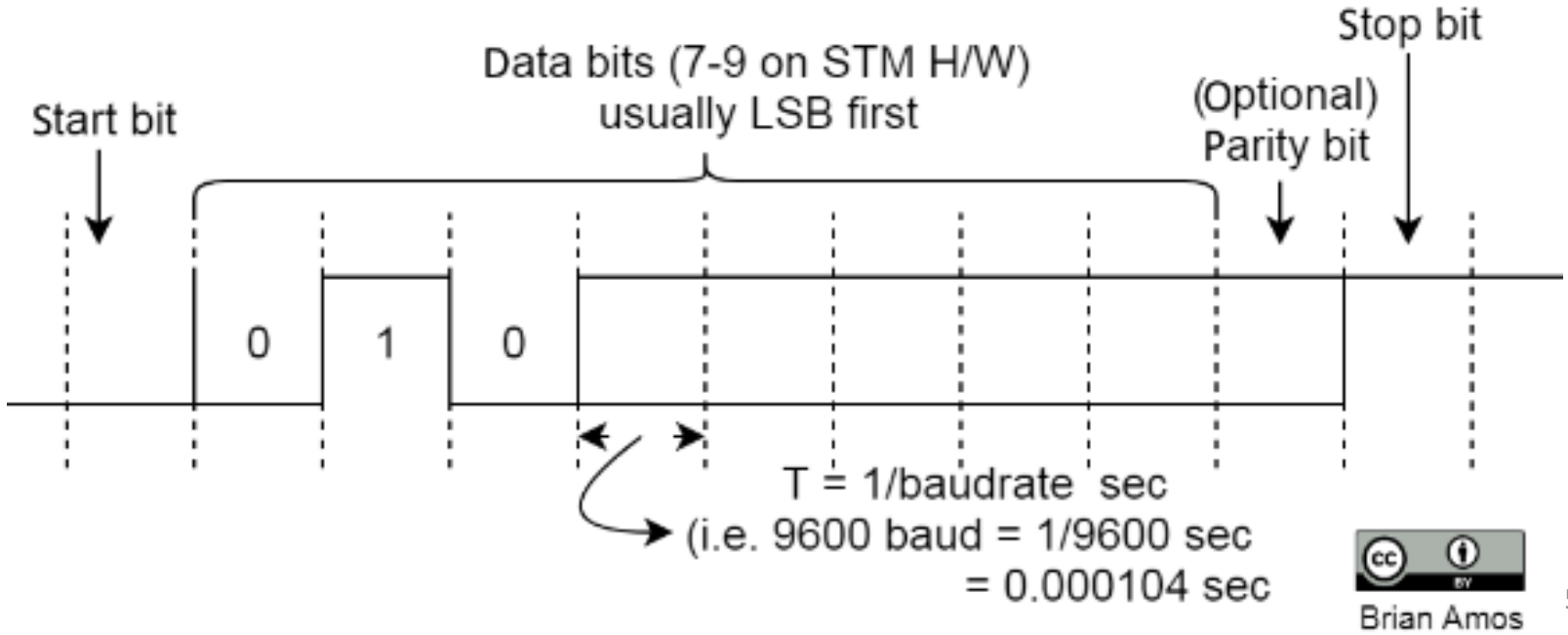
Plan

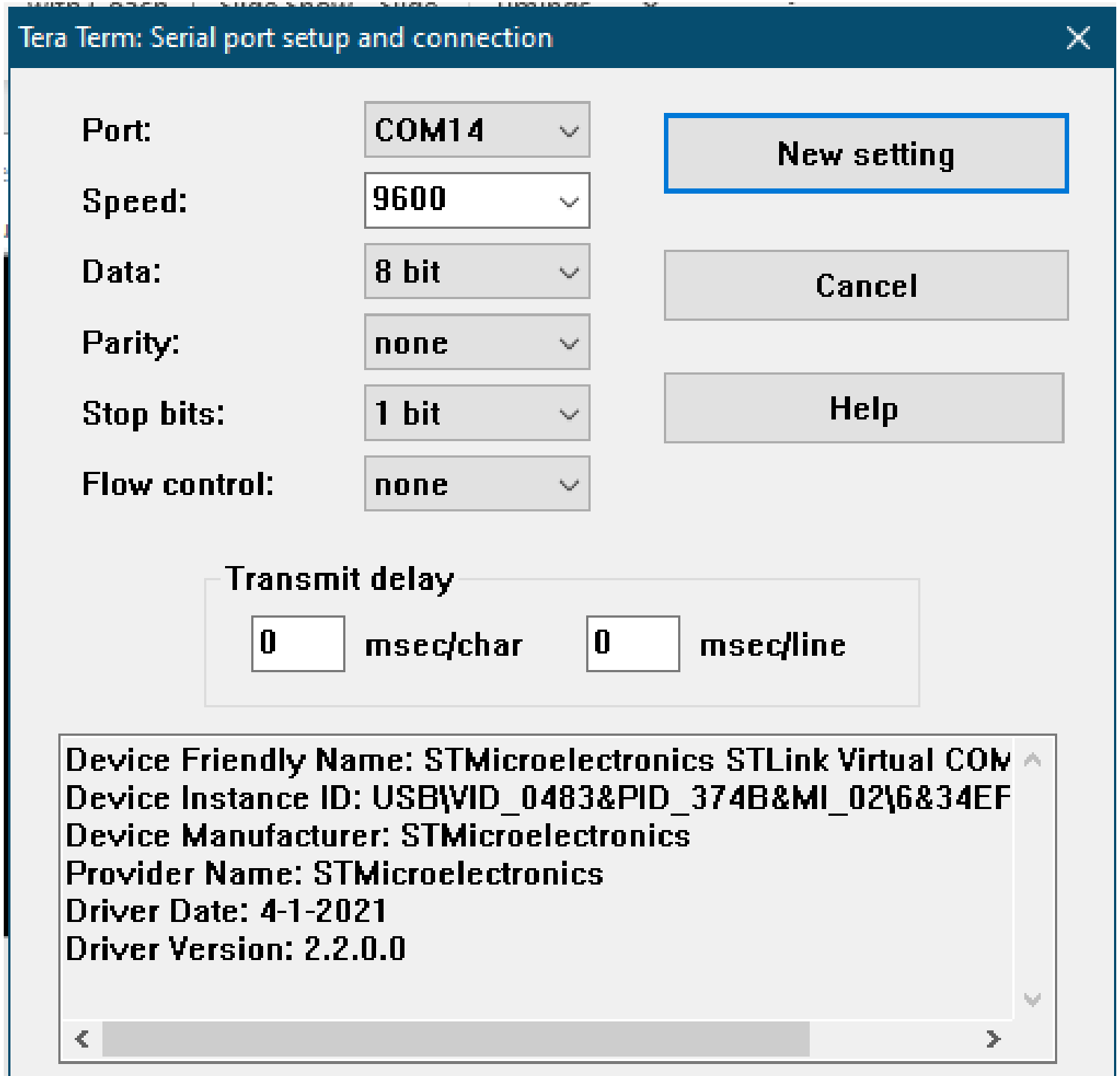
- Présentation de l'UART
- Création d'un pilote UART à interrogation
- Différencier les tâches et les ISR
- Créer des pilotes basés sur des ISR
- Créer des pilotes basés sur DMA
- Les tampons de flux FreeRTOS
- Choisir un modèle de pilote
- Utilisation de bibliothèques tierces (STM HAL)

Présentation de l'UART

- Comme nous l'avons brièvement évoqué au chapitre 4, Choix du bon MCU, l'acronyme UART signifie Universal Asynchronous Receiver/Transmitter (récepteur/émetteur asynchrone universel).
- Le matériel UART prend des octets de données et les transmet sur un câble en modulant la tension d'une ligne de signal à un taux prédéterminé..

Trame UART

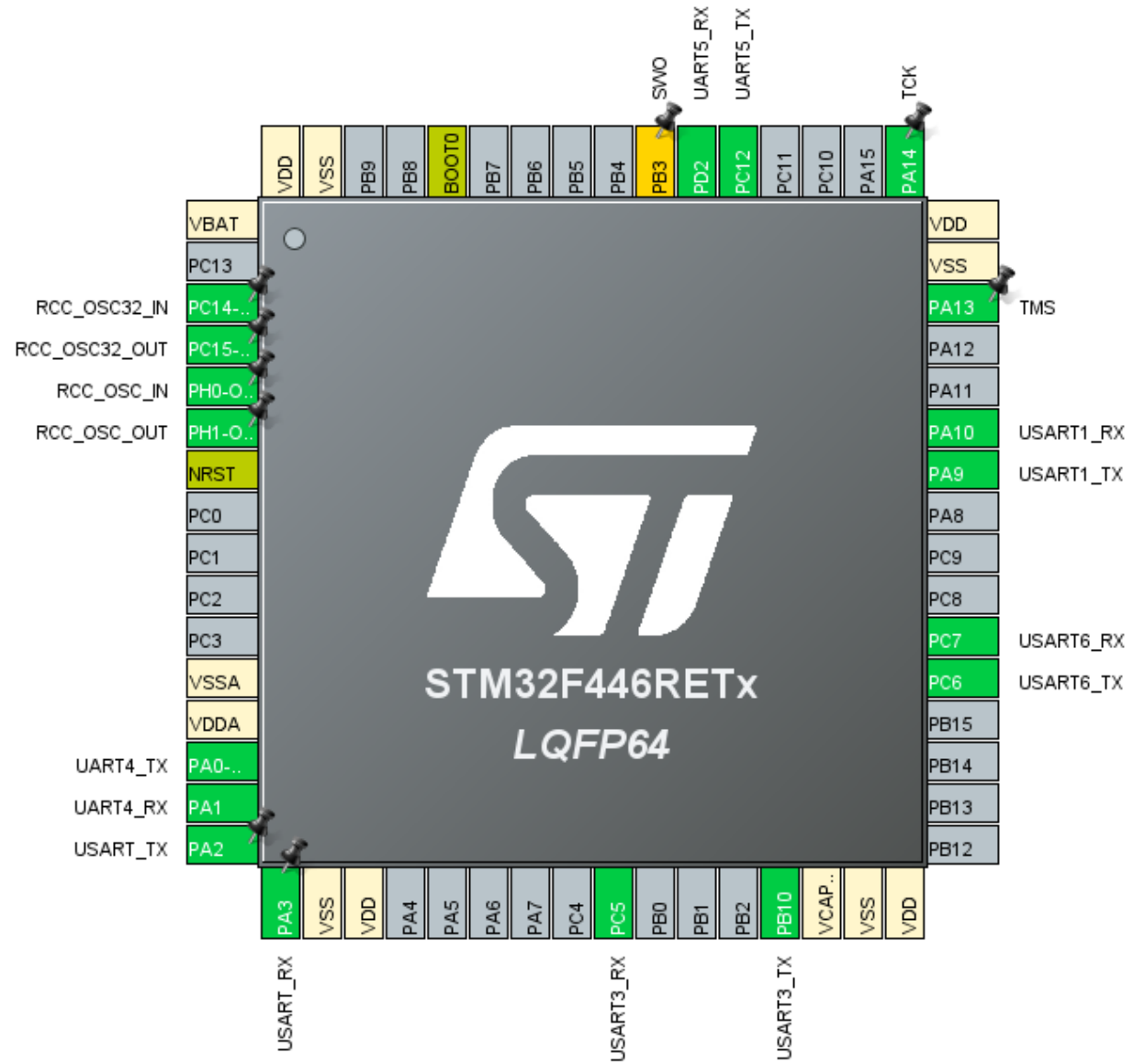



Tera Term: Serial port setup and connection
Port:
COM14
Speed:
9600
Data:
8 bit
Parity:
none
Stop bits:
1 bit
Flow control:
none
New setting
Cancel
Help
Transmit delay
0
msec/char
0
msec/line
Device Friendly Name: STMicroelectronics STLink Virtual COM
Device Instance ID: USB\VID_0483&PID_374B&MI_02\6&34EF
Device Manufacturer: STMicroelectronics
Provider Name: STMicroelectronics
Driver Date: 4-1-2021
Driver Version: 2.2.0.0

UART

- La nature asynchrone d'un UART signifie qu'aucune ligne d'horloge supplémentaire n'est nécessaire pour surveiller les transitions des bits individuels.
- Pour ce faire, le matériel est configuré pour assurer la transition de chaque bit à une fréquence spécifique (débit en bauds).
- Le matériel UART ajoute également un cadrage supplémentaire au début et à la fin de chaque paquet qu'il transmet.
- Les bits de départ et d'arrêt signalent le début et la fin d'un paquet.
- Ces bits (ainsi qu'un bit de parité facultatif) sont utilisés par le matériel pour garantir la validité des paquets (qui ont généralement une longueur de 8 bits).
- Une forme plus générale de matériel UART est le récepteur-émetteur universel synchrone/asynchrone (universal synchronous/asynchronous receiver transmitter (USART)).
- Les USART sont capables de transférer des données soit de manière synchrone (avec l'ajout d'un signal d'horloge), soit de manière asynchrone (sans signal d'horloge).

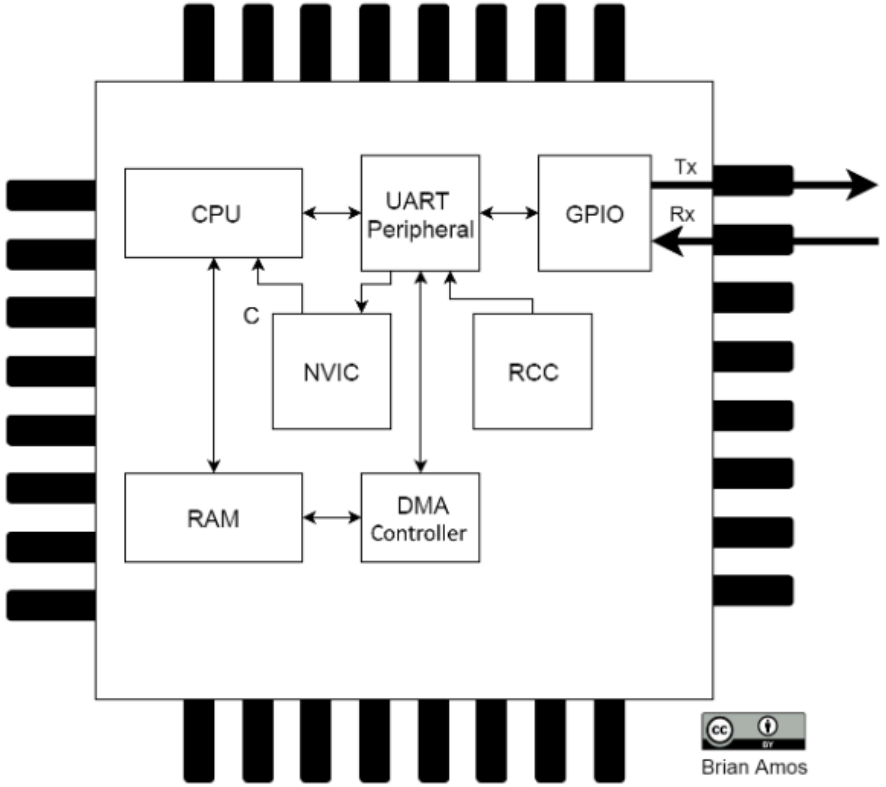
La carte nucleo-f446RE comprend 6 USARTs



Configuration de l'UART

- Comme le montre le schéma fonctionnel simplifié ci-dessous, la configuration d'un UART pour la communication implique quelques composants.
- L'UART doit être correctement configuré pour transmettre à la vitesse de transmission correcte, aux paramètres de parité, au contrôle de flux et aux bits d'arrêt.
- Les autres matériels qui interagissent avec l'UART doivent également être configurés correctement :

UARTs dans le STM32



Création d'un pilote UART à interrogation

- Lorsque vous écrivez des pilotes de bas niveau, il est indispensable de lire la fiche (datasheet) technique afin de comprendre le fonctionnement du périphérique.

- Même si vous n'écrivez pas un pilote de bas niveau à partir de zéro, il est toujours bon de se familiariser avec le matériel avec lequel vous allez travailler.

- Plus vous serez familiarisé, plus il vous sera facile de diagnostiquer les comportements inattendus et de créer des solutions efficaces.

Création d'un pilote UART à interrogation

- Notre premier pilote adoptera une approche extrêmement simple pour obtenir des données de l'UART et dans une file d'attente qui peut être facilement surveillée et utilisée par n'importe quelle tâche dans le système.

- En surveillant le bit de réception non vide (receive not empty, RXNE) du registre d'état d'interruption (interrupt status register, ISR) du périphérique UART, le pilote peut déterminer quand un nouvel octet est prêt à être transféré du registre de données de réception (receive data register, RDR)) de l'UART dans la file d'attente.

- Pour faciliter au maximum cette opération, la boucle while est placée dans une tâche (polledUartReceiver), ce qui permet d'exécuter d'autres tâches plus prioritaires.

La fonction main()

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  xTaskCreate(polledUARTReceiver, "pollling UART", STACK_SIZE, NULL,
              tskIDLE_PRIORITY+2, NULL);
  xTaskCreate(HandlerTask, "UART print task", STACK_SIZE,
              NULL, tskIDLE_PRIORITY+3, NULL);
  uart2_BytesReceived = xQueueCreate(10,sizeof(char));
  vTaskStartScheduler();
  while (1)
  {
      printf("pollling UART“);
  }
}
```

Fonction polledUARTReceiver

```
void polledUARTReceiver(void *pvParameters)
{
    uint8_t nexByte;
    USART2_UART_RX_Init();
    while(1)
    {
        nexByte = uart2_read();
        xQueueSend(uart2_BytesReceived,&nexByte,0);
    }
}

char rcByte;
void HandlerTask(void *pvParameters)
{
    while(1)
    {
        xQueueReceive(uart2_BytesReceived, &rcByte, portMAX_DELAY);
    }
}
```

```
char uart2_read(void)
{
    while(!(USART2->SR & USART_SR_RXNE));
    return USART2->DR;
}
```

Avantages et inconvénients d'un pilote à interrogation

- Voici quelques-uns des avantages liés à l'utilisation d’un pilote interrogé:
  - Il est facile à programmer.
  - Toute tâche a un accès immédiat aux données de la file d'attente.
- En même temps, cette approche pose de nombreux problèmes :
  - Il doit s'agir de l'une des tâches les plus prioritaires du système.
  - Le risque de perte de données est élevé si elle n'est pas exécutée en priorité.
  - Elle gaspille énormément de cycles de l'unité centrale.

Différenciation entre les tâches et les ISRs

- Avant de nous lancer dans le codage d'un pilote de périphérique qui utilise les interruptions, regardons rapidement comment les interruptions se comparent aux tâches de FreeRTOS.
- Il existe de nombreuses similitudes entre les tâches et les ISRs:
  - Elles permettent toutes deux d'exécuter du code en parallèle.
  - Elles ne s'exécutent que lorsque c'est nécessaire.
  - Elles peuvent toutes deux être écrites en C/C++ (les ISRs ne doivent généralement plus être écrites en code assembleur).

Différenciation entre les tâches et les ISRs

- Mais il existe également de nombreuses différences entre les tâches et les ISRs :
  - Les ISR sont mis en contexte par le matériel ; les tâches sont mises en contexte par le noyau RTOS :
    - Les tâches sont toujours mises en contexte par le noyau FreeRTOS. .
  - Les interruptions, en revanche, sont générées par le matériel de l'unité centrale.
  - Les ISR doivent sortir le plus rapidement possible ; les tâches sont plus indulgentes :
    - Les tâches FreeRTOS sont souvent configurées pour s'exécuter de la même manière qu'une boucle while infinie
      - ils seront synchronisés avec le système à l'aide de primitives (telles que les files d'attente et les sémaphores) et mis en contexte en fonction de leur priorité.

Différenciation entre les tâches et les ISRs

- Les fonctions de l’ISR ne prennent pas de paramètres d'entrée :
  - Contrairement aux tâches, les ISRs ne peuvent jamais avoir de paramètres d'entrée.
- Étant donné qu'une interruption est déclenchée en raison d'un état matériel, la tâche la plus importante de l'ISR est de lire l'état matériel (par le biais de registres mappés en mémoire) et de prendre la ou les mesures appropriées.
- Par exemple, une interruption peut être générée lorsqu'un UART reçoit un octet de données.
- Dans ce cas, l'ISR lira un registre d'état, lira (et stockera) l'octet reçu dans une variable statique et effacera l'interruption.

Utiliser l'API FreeRTOS à partir des interruptions

- La plupart des primitives FreeRTOS couvertes jusqu'à présent ont des versions ISR-safe de leurs APIs.

- Par exemple, xQueueSend() a une version ISR-safe équivalente, xQueueSendFromISR().

Utiliser l'API FreeRTOS à partir des interruptions

- Il existe quelques différences entre la version sécurisée par l'ISR et l'appel standard
  - Les variantes FromISR ne bloquent pas. Par exemple, si xQueueSendFromISR rencontre une file d'attente pleine, il retournera immédiatement.
  - Les variantes FromISR nécessitent un paramètre supplémentaire, BaseType_t *pxHigherPriorityTaskWoken, qui indique si une tâche de priorité supérieure doit ou non être placée dans le contexte immédiatement après l'interruption.
  - Seules les interruptions qui ont une priorité logiquement inférieure à celle définie par configMAX_API_CALL_INTERRUPT_PRIORITY dans FreeRTOSConfig.h sont autorisées à appeler les fonctions de l'API FreeRTOS.

Création de pilotes basés sur des ISR

- Dans la première itération du pilote UART, une tâche interrogeait les registres périphériques UART pour déterminer si un nouvel octet avait été reçu.
- L'interrogation constante est à l'origine de l’utilisation par la tâche de plus de 95% des cycles de l'unité centrale.
- Le travail le plus significatif effectué par ce pilote basé sur une tâche était le transfert d'octets de données hors du périphérique UART et dans la file d'attente.
- Dans cette itération du pilote, au lieu d'utiliser une tâche pour interroger continuellement les registres UART, nous allons configurer le périphérique UART2 et le NVIC pour fournir une interruption lorsqu'un nouvel octet est reçu.

Pilote basé sur une file d'attente

- Tout d'abord, examinons une implémentation plus efficace que le pilote interrogé (précédemment implémenté en interrogeant les registres UART au sein d'une tâche).
- Dans cette implémentation, au lieu d'utiliser une tâche pour interroger de manière répétée les registres UART, une fonction sera utilisée pour configurer le périphérique afin d'utiliser les interruptions et de lancer le transfert.
- Cet exemple comporte trois composants principaux :
  - HandlerTask()
  - start_rx_interrupt()
  - USART2_IRQHandler()

La fonction main()

```
int main(void)
{
	HAL_Init();
	SystemClock_Config();
	MX_GPIO_Init();

	xTaskCreate(HandlerTask, "UART print task", STACK_SIZE,
			NULL, tskIDLE_PRIORITY+3, NULL);

	uart2_BytesReceived = xQueueCreate(10,sizeof(char));
	vTaskStartScheduler();
	while (1)
	{
	}
}
```

```
void UARTReceiver(void *pvParameters)
{
   uint8_t nexByte;
   USART2_UART_RX_Init();
   start_rx_interrupt();
   while(1)
   {
      xQueueSend(uart2_BytesReceived,&nexByte,0);
   }


}
```

```
void start_rx_interrupt(void)
{
   rxInProgress = 1;
   USART2->CR1 |= USART_CR1_RXNEIE;
   NVIC_SetPriority(USART2_IRQn,6);
   NVIC_EnableIRQ(USART2_IRQn);
}
```

```
void USART2_IRQHandler(void)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(USART2->SR & USART_SR_RXNE)
   {
      uint8_t tempVal = (uint8_t)USART2->DR;
      if(rxInProgress)
      {
       xQueueSendFromISR(uart2_BytesReceived,&tempVal,&xHigherPriorityTaskWoken);
      }
   }
   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```
void USART2_IRQHandler(void)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(USART2->SR & USART_SR_RXNE)
   {
      uint8_t tempVal = (uint8_t)USART2->DR;
      if(rxInProgress)
      {
       xQueueSendFromISR(uart2_BytesReceived,&tempVal,&xHigherPriorityTaskWoken);
      }
   }
   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

La fonction main()

Réception d'un caractère à la fois

```
int main(void)
{
	HAL_Init();
	SystemClock_Config();
	MX_GPIO_Init();

	xTaskCreate(HandlerTask, "UART print task", STACK_SIZE,
			NULL, tskIDLE_PRIORITY+3, NULL);

	uart2_BytesReceived = xQueueCreate(10,sizeof(char));
	vTaskStartScheduler();
	while (1)
	{
	}
}
```

Réception d'un caractère à la fois

```
void start_rx_interrupt(void)
{
   rxInProgress = 1;
   USART2->CR1 |= USART_CR1_RXNEIE;
   NVIC_SetPriority(USART2_IRQn,6);
   NVIC_EnableIRQ(USART2_IRQn);
}
```

Réception d'un caractère à la fois

```
void USART2_IRQHandler(void)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(USART2->SR & USART_SR_RXNE)
   {
      uint8_t tempVal = (uint8_t)USART2->DR;
      if(rxInProgress)
      {
       xQueueSendFromISR(uart2_BytesReceived,&tempVal,&xHigherPriorityTaskWoken);
      }
   }
   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Réception d'un caractère à la fois

```
void USART2_IRQHandler(void)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(USART2->SR & USART_SR_RXNE)
   {
      uint8_t tempVal = (uint8_t)USART2->DR;
      if(rxInProgress)
      {
       xQueueSendFromISR(uart2_BytesReceived,&tempVal,&xHigherPriorityTaskWoken);
      }
   }
   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Réception d'une chaîne de caractères

La fonction main()

```
int main(void)
{
   HAL_Init();
   SystemClock_Config();
   MX_GPIO_Init();
   rxDone = xSemaphoreCreateBinary();
   xTaskCreate(HandlerTask, "UART print task", STACK_SIZE, NULL, tskIDLE_PRIORITY+3,
               NULL);
   uart2_BytesReceived = xQueueCreate(10,sizeof(char));
   vTaskStartScheduler();
   while (1)
   {
   }
}
```

HandlerTask()

```
void HandlerTask(void *pvParameters)
{
    for (int i = 0;i < sizeof(rxData);i++)
    {
        rxData[i] = 0;
    }
    const TickType_t timeout = pdMS_TO_TICKS(8000);
    USART2_UART_RX_Init();
    while(1)
    {
        start_rx_interrupt(rxData,EXPECTED_LENGTH);
        if(xSemaphoreTake(rxDone,timeout) == pdPASS)
        {
                if(EXPECTED_LENGTH == rxItr)
                {
                        sprintf(rxCode,"received");
                }
                else
                {
                        sprintf(rxCode,"Length mismatch");
                }
        }
        else
        {
                sprintf(rxCode,"Timeout" );
        }
    }
}
```

start_rx_interrup()

```
uint32_t start_rx_interrupt(uint8_t * Buffer, uint_fast16_t len)
{
   if(!rxInProgress&& (NULL != Buffer))
   {
      rxInProgress = 1;
      rxLen = len;
      rxBuff = Buffer;
      rxItr = 0;
      USART2->CR1|=USART_CR1_RXNEIE;
      NVIC_SetPriority(USART2_IRQn,6);
      NVIC_EnableIRQ(USART2_IRQn);
      return 0;
   }
   return -1;
}
```

USART2_IRQHandler()

```
void USART2_IRQHandler(void)
{
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(USART2->SR & USART_SR_RXNE)
   {
       uint8_t tempVal = (uint8_t)USART2->DR;
      if(rxInProgress)
      {
         rxBuff[rxItr++] = tempVal;
         if(rxLen == rxItr)
         {
            rxInProgress = 0;
            xSemaphoreGiveFromISR(rxDone,&xHigherPriorityTaskWoken);
         }
      }
   }
   portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

UART | Transmit | Receive | DMA

```c
int main(void)
{

   HAL_Init();

   SystemClock_Config();

   MX_GPIO_Init();
   MX_DMA_Init();
   MX_USART2_UART_Init();
   HAL_UART_Receive_DMA(&huart2, rxdata, sizeof(rxdata));

   while (1)
   {
   }
}
```

UART | Transmit | Receive | DMA

```
HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
   /* Check that a Rx process is not already ongoing */
   if (huart->RxState == HAL_UART_STATE_READY)
   {
      if ((pData == NULL) || (Size == 0U))
      {
               return HAL_ERROR;
      }
      /* Set Reception type to Standard reception */
      huart->ReceptionType = HAL_UART_RECEPTION_STANDARD;
      return (UART_Start_Receive_DMA(huart, pData, Size));
   }
   else
   {
   return HAL_BUSY;
   }
}
```

UART | Transmit | Receive | DMA

```c
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(huart);
  /* NOTE: This function should not be modified, when the callback is needed,
  the HAL_UART_RxCpltCallback could be implemented in the user file
  */
  HAL_UART_Transmit(&huart2, rxdata, sizeof(rxdata), 100);
}
```