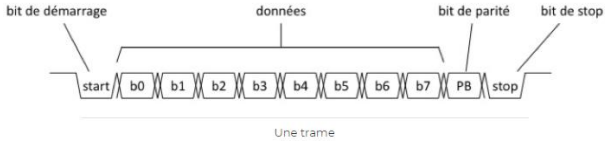
UART sur stm32 Universal asynchronous receiver transmitter (UART)

Trame :



Une trame

Le bit de parité peut être utilisé pour détecter des erreurs.
La parité est générée par l'émetteur et vérifiée par le receveur.
Si l'erreur est faible et n'entraîne pas de conséquence grave ne pas utiliser de parité est à privilégier.
Liaison Série
Pas d'horloge communes, pour identifier les différents bits d'une trame, le récepteur se base sur la durée de ses bits.
Cette durée doit être fixée et identique pour l'émetteur et le récepteur.
La durée d'un bit est le temps entre chaque bit.
Ainsi :
L'émetteur émet un bit, attend la durée d'un bit puis émet un nouveau bit.
Le receveur attend jusqu'à ce qu'il détecte un bit de démarrage puis attend la moitié d'une durée d'un bit pour se caler sur la lecture des bits qu'il reçoit.

La durée d'un bit décrit en termes de baud rate (unité bps), le nombre de bits par seconde qui est transmis par la communication série

$$\text{baud rate} = \frac{1}{\text{durée d'un bit}}$$

La bande passante (bandwith) d'une communication quantifie l'information transmise sans prendre en considération les surcoûts induits par les bits de démarrage, de parité et de stop soit :

$$\text{Bande passante} = \frac{\text{nombre de bits de la donnée par trame}}{\text{nombre de bits total de la trame}} \text{baud rate}$$

Pour transmettre 8 bits de données (soit 1 octet), il faut envoyer 10 bits sur le bus (les 8 bits de données plus les bits de start et stop). La durée de transmission d'un bit est de 1/ 9600, soit environ 0,1 ms. Donc pour envoyer les 10 bits, il faut 1 ms.

Une autre caractéristique du système est full-duplex, half-duplex (ici un bus doit assurer la retransmission en cas de collision)
Une communication UART est asynchrone il ne faut pas non plus que les horloges de dérives pas trop car une synchronisation à lieu à chaque bit de démarrage, par contre cela limite la bande passante.

Pour USART les bauds rate sont élevé il y a une horloge commune permet de réduire les surcouts dus au bit de démarrage et de stop, car les données peuvent être plus longues.

STM32 dispose d'un périphérique USART full-duplex permettant un baud rate jusqu'à 4.5Mbps.
Avec une broche RX (recevoir) et une TX (transmettre).
Quand l'émetteur est désactivé la broche prend l'état imposée par la configuration du port alors que si l'émetteur est activé et que rien n'est à transmettre la broche TX est dans l'état haut.

Configuration d'un USART en émetteur avec 1 bit de démarrage, 8 bits données, 1 bit stop (pas de parité) et un baud à 9600bps.

- Activer l'USART en mettant à 1 le bit UE du registre CR1,
- Choisir la taille des données (8 ou 9 bits) à l'aide du bit M du registre CR1,
- Indiquer le nombre de bits stop dans le champ de bits STOP du registre CR2,
- Sélectionner le baud rate à l'aide du registre BRR,
- Mettre le bit TE de CR1 à 1 pour envoyer la première trame d'attente.

Pour régler le baud rate la documentation, nous montre qu'il faut diviser la fréquence d'entrée du périphérique 80MHZ par une valeur représentée par un nombre à virgule fixe.
A saisir dans le registre BRR avec les 4 premiers bits pour la partie fractionner et les 12 suivants la partie entière.

Envoyer
1. Programmez les bits M dans USART_CR1 pour définir la longueur du mot.
2. Sélectionnez le débit en bauds souhaité à l'aide du registre USART_BRR.
3. Programmez le nombre de bits d'arrêt dans USART_CR2.
4. Activez l'USART en écrivant le bit UE dans le registre USART_CR1 à 1.
5. Sélectionnez l'activation DMA (DMAT) dans USART_CR3 si la communication multi-tampon doit prendre endroit.
   Configurez le registre DMA comme expliqué dans la communication multi-tampon.
6. Réglez le bit TE dans USART_CR1 pour envoyer une trame inactive comme première transmission.
7. Ecrire les données à envoyer dans le registre USART_TDR (ceci efface le bit TXE).
   Répétez ceci pour chaque donnée à transmettre en cas de mémoire tampon unique.
8. Après avoir écrit les dernières données dans le registre USART_TDR, attendez que TC=1.
   Cela indique que la transmission de la dernière trame est terminée.
   Ceci est requis par exemple lorsque l'USART est désactivé ou passe en mode Halt pour éviter de corrompre la derniere Transmission.

Procédure de réception de trame
1. Programmez les bits M dans USART_CR1 pour définir la longueur du mot.
2. Sélectionnez le débit en bauds souhaité à l'aide du registre de débit en bauds USART_BRR
3. Programmez le nombre de bits d'arrêt dans USART_CR2.
4. Activez l'USART en écrivant le bit UE dans le registre USART_CR1 à 1.
5. Sélectionnez l'activation DMA (DMAR) dans USART_CR3 si la communication multi-tampon doit prendre endroit.
   Configurez le registre DMA comme expliqué dans la communication multi-tampon.
6. Activez le bit RE USART_CR1. Cela permet au récepteur qui commence à rechercher un peu de démarrage.

Lorsqu'une trame est reçu :
- Le bit RXNE est défini pour indiquer que le contenu du registre à décalage est transféré au RDR. En d'autres termes, les données ont été reçues et peuvent être lues (ainsi que leurs indicateurs d'erreur associés).
- Une interruption est générée si le bit RXNEIE est défini.
- Les drapeaux d'erreur peuvent être définis si une erreur de trame, du bruit ou une erreur de dépassement a été détecté lors de la réception. Le drapeau PE peut également être défini avec RXNE.
- En multi-tampon, RXNE est défini après chaque octet reçu et est effacé par la lecture DMA de le Registre des données de réception.
- En mode tampon unique, l'effacement du bit RXNE est effectué par un logiciel lu sur le Registre USART_RDR. Le drapeau RXNE peut également être effacé en écrivant 1 dans le RXFRQ dans le registre USART_RQR. Le bit RXNE doit être effacé

Pratique :

Comme l'exemple d'openclassRoom

42	42	E3	E3	E3	68	D10	D10	101	101	D11	PA9	I/O	FT_lu	-	TIM1_CH2, USART1_TX, LCD_COM1, TIM15_BKIN, EVENTOUT	OTG_FS_VBUS
43	43	D2	D2	D2	69	C12	C12	102	102	C12	PA10	I/O	FT_lu	-	TIM1_CH3, USART1_RX, OTG_FS_ID, LCD_COM2, TIM17_BKIN, EVENTOUT	-

3 USART, 2 UART, 1 Low-power UART

40.8.1 USART control register 1 (USART_CR1)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

40.8.4 USART baud rate register (USART_BRR)

This register can only be written when the USART is disabled (UE=0). It may be automatically updated by hardware in auto baud rate detection mode.

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:4 **BRR[15:4]**
BRR[15:4] = USARTDIV[15:4]

Bits 3:0 **BRR[3:0]**
When OVER8 = 0, BRR[3:0] = USARTDIV[3:0].
When OVER8 = 1:
BRR[2:0] = USARTDIV[3:0] shifted 1 bit to the right.
BRR[3] must be kept cleared.

Calcul du baud rate
Si USART->CR1 => OVER 8 à 0 pour nous (on peut le mettre à 1 mais le calcul change)

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times \text{USARTDIV}} \quad \text{if OVER8 = 1}$$

Peripheral Clock

Divide factor to generate different baud rates

$$\text{Tx/Rx baud} = \frac{f_{CK}}{16 \times \text{USARTDIV}} \quad \text{if OVER8 = 0}$$

Figure 1. Formule de calcul du débit en bauds.

Mode usart nous parlons de UART asynchrone
Over8=1 cela signifie nous utilisons un suréchantillonnage par 8, et le bloc récepteur du périphérique prend 8 échantillons pour comprendre un peu.
Over=8=0 on utilise un suréchantillons par 16.
USARTDIV est le facteur de division pour générer différents débits en bauds (la valeur minimale est 1)

Formule générique :

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

OVER8 =1 , if Oversampling by 8 is used
OVER8=0, if Oversampling by 16 is used

Figure 2. Formule générique pour le calcul du débit en bauds.

Générer un baud de 9600 bits par seconde
Si horloge générique est de 16MHZ et le suréchantillonnage par 16 est utilisé :
= 16000000/(8*2*9600) = 104.16 arrondi à 17 dans le BRR ce registre est divisé en 2 sections [15 :4] et [3 :0]

Figure 3. Sections de USART_BRR.

$$1.3254 = \underbrace{13254}_{\text{mantisse}} \times \overbrace{10^{-4}}^{\text{exposant}}$$

$$signe \times mantisse \times \text{base}^{exposant}$$

Pour exemple 104.1978 (fractionner 1875 et 104 est la mantisse) : 0000 0000 0000 0000 mantisse de 11 bits

30.6.3 Baud rate register (USART_BRR)

Note: The baud counters stop counting if the TE or RE bits are disabled respectively.

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value

Bits 15:4 **DIV_Mantissa[11:0]**: mantissa of USARTDIV
These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV_Fraction[3:0]**: fraction of USARTDIV
These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV_Fraction3 bit is not considered and must be kept cleared.

Figure 4. Registre de débit en bauds.

Donc 0,1876 mutipliez par 16 (suréchantillonnage) = 3 partie fractionnée
Donc 104 en hexa 0x68
Donc USARTDIV sera 0x683 pour 9600bps

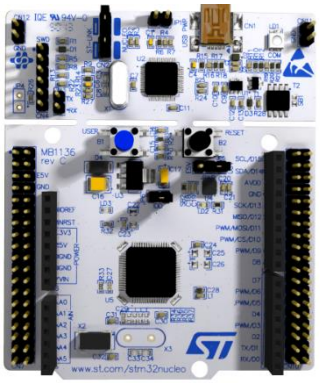
Pour nous 72Mhz en 9600bps over8=0
= 72000000/(8*2*9600) = 468.75 on retrouve bien le bon résultat
Pour nous on est à 80MHZ : = 80000000/(8*2*9600)=520,83

40.8.11 USART transmit data register (USART_TDR)

Address offset: 0x28

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TDR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw



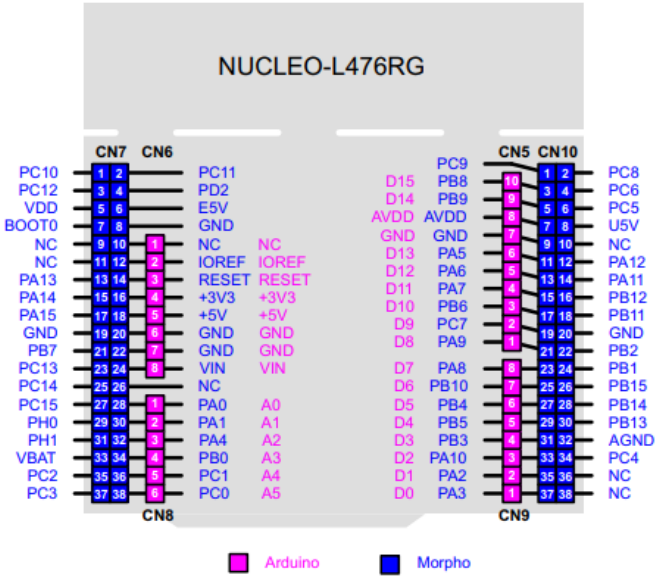


Figure 3. Top layout

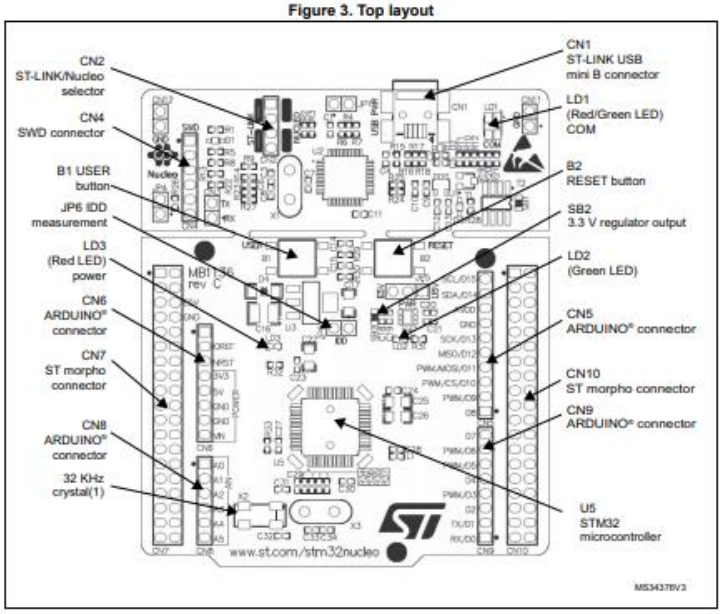


Table 23. ARDUINO® connectors on NUCLEO-L476RG

Connector	Pin	Pin name	STM32 pin	Function
Left connectors				
CN6 power	1	NC	-	-
	2	IOREF	-	3.3V Ref
	3	RESET	NRST	RESET
	4	+3.3V	-	3.3V input/output
	5	+5V	-	5V output
	6	GND	-	ground
	7	GND	-	ground
	8	VIN	-	Power input
CN8 analog	1	A0	PA0	ADC12_IN5
	2	A1	PA1	ADC12_IN6
	3	A2	PA4	ADC12_IN9
	4	A3	PB0	ADC12_IN15
	5	A4	PC1 or PB9(1)	ADC123_IN2 (PC1) or I2C1_SDA (PB9)
	6	A5	PC0 or PB8(1)	ADC123_IN1 (PC0) or I2C1_SCL (PB8)
Right connectors				
CN5 digital	10	D15	PB8	I2C1_SCL
	9	D14	PB9	I2C1_SDA
	8	AREF	-	AVDD
	7	GND	-	ground
	6	D13	PA5	SPI1_SCK
	5	D12	PA6	SPI1_MISO
	4	D11	PA7	TIM17_CH1 or SPI1_MOSI
	3	D10	PB6	TIM4_CH1 or SPI1_CS
CN5 digital	2	D9	PC7	TIM3_CH2
	1	D8	PA9	-
CN9 digital	8	D7	PA8	-
	7	D6	PB10	TIM2_CH3
	6	D5	PB4	TIM3_CH1
	5	D4	PB5	-
	4	D3	PB3	TIM2_CH2
	3	D2	PA10	-
	2	D1	PA2	USART2_TX
	1	D0	PA3	USART2_RX

1. Refer to *Table 10: Solder bridges* for details.

PA9 USART1_TX même pas dans la doc Nucleo mais dans [datasheet]

Table 16. STM32L476xx pin definitions (continued)

Pin Number											Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
LQFP64	LQFP64_SMPS	WLCSP72	WLCSP72_SMPS	WLCSP81	LQFP100	UFBGA132	UFBGA132_SMPS	LQFP144	LQFP144_SMPS	UFBGA144					Alternate functions	Additional functions
41	41	E2	E2	E2	67	D11	D11	100	100	D12	PA8	I/O	FT_l	-	MCO, TIM1_CH1, USART1_CK, OTG_FS_SOF, LCD_COM0, LPTIM2_OUT, EVENTOUT	-
42	42	E3	E3	E3	68	D10	D10	101	101	D11	PA9	I/O	FT_lu	-	TIM1_CH2, USART1_TX, LCD_COM1, TIM15_BKIN, EVENTOUT	OTG_FS_VBUS
43	43	D2	D2	D2	69	C12	C12	102	102	C12	PA10	I/O	FT_lu	-	TIM1_CH3, USART1_RX, OTG_FS_ID, LCD_COM2, TIM17_BKIN, EVENTOUT	-
44	44	D1	D1	D1	70	B12	B12	103	103	B12	PA11	I/O	FT_u	-	TIM1_CH4, TIM1_BKIN2, USART1_CTS, CAN1_RX, OTG_FS_DM, TIM1_BKIN2_COMP1, EVENTOUT	-
45	45	C1	C1	C1	71	A12	A12	104	104	B11	PA12	I/O	FT_u	-	TIM1_ETR, USART1_RTS_DE, CAN1_TX, OTG_FS_DP, EVENTOUT	-

Liaison

RaspberryPi Nucleo

GND----------CN7(20)

TXD----------CN10(23) PA10 USART1_RX

RXD----------CN10(21) PA9 USART1_TX

Programme keil (Envoyer les données depuis stm32)

GND ----------CN7(20)

RXD <----------CN10(21) PA9 USART1_TX

```c
//main.c
/* Includes */
#include "stm3214xx.h"
#include "stm321476xx.h"
//#include "IT_TIM2.h"

void configure_usart1_9600bps(void);
void send(char data);
void configure_gpio_alternate_push_pull(void);
int  main(void){
   char data = 'A';
   int cmpt = 0;
   configure_gpio_alternate_push_pull ();
   configure_usart1_9600bps ();
   while (cmpt < 2)  {
         send(data);
         cmpt ++;
    }
    while(1) {
       /*traitement de scrutation*/
    }
}
void configure_usart1_9600bps(){
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN; // validation horloge USART1
     USART1->CR1 |= USART_CR1_TE; // Envoi de la première trame d'attente
                                  //bit d'activation de transmission (TE)
                                  //doit être défini pour activer la fonction emetteur
   USART1->CR1 |= USART_CR1_RE; // Envoi de la première trame d'attente
                                  //bit d'activation de transmission (TE)
                                  //doit être défini pour activer la fonction emetteur
   USART1->BRR |= 0x001A << 4;  // Fixe le baud rate à 9600bps mauvais calcul 520
    USART1->BRR |= 0x01; // Fixe le baud rate à 9600bps 83
   USART1->CR1 |= USART_CR1_UE; // Activation de l'USART
}
void send(char data){
    USART1->TDR |= data; // Ecriture de la donnée dans le registre DR 0x0041 = A
    while(!(USART1->ISR & USART_ISR_TC)) {} //attente fin de transmission
} //attendre le bit TC du registre SR passe à 1 (indique la dernière trame a été transmise)

//Broche USART PA.9 en mode alternat push pull
void configure_gpio_alternate_push_pull(){
  //uint32_t position = 0x20u;
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
    GPIOA->MODER &= 0xABFBFFFF; //PA9 alternate function mode 10
    //GPIOA->OTYPER = 0x0000000;
    GPIOA->AFR[1] = 0x0000070;
}
```

Après CubMX la compréhension est ça marche (code qui fonctionne ci-dessus)
MXCUBE config PA.9 USART1 (9600,8bits, 1stop)

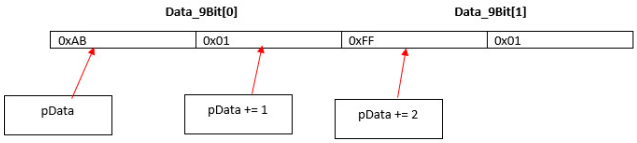
```
  /* USER CODE BEGIN 2 */
char *welcome="bonjour|\n\r";
                        //USART1, chaine de caractère de type pointeur, taille de la chaine, timout
//HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)

HAL_UART_Transmit( &huart1, (uint8_t *)welcome, strlen(welcome), HAL_MAX_DELAY);
  /* USER CODE END 2 */
```

AStar.4 (membre de la communauté)
Modifié le 28 mai 2021 à 03h04

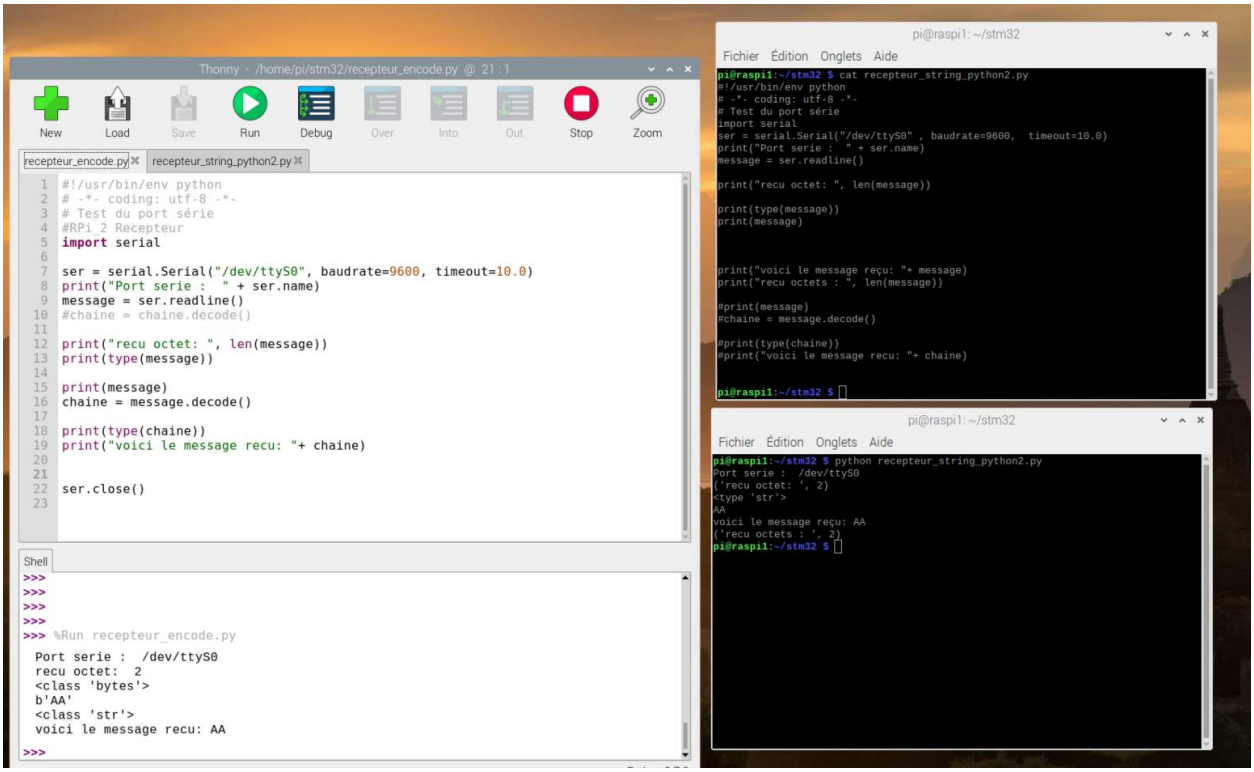
Cher @TDK (membre de la communauté) , merci pour votre réponse. Votre réponse m'a aidé à comprendre l'essence du problème.
Je voudrais revoir ma compréhension comme ci-dessous :
Si je veux envoyer des données 9 bits, je dois déclarer et transférer à la fonction des données 2 octets (uint16_t)
Étant donné que le pointeur pData déclaré dans la fonction est de 8 bits (uint8_t), il est nécessaire d'augmenter pData de 2U pour pointer correctement sur l'emplacement suivant de la zone de données
Par exemple, si je veux passer un Array : uint16_t Data_9Bit[2] = {0x1AB, 0x1FF} ;

uint16_t Data_9Bit[2] = {0x1AB, 0x1FF};



Ainsi, les données transmises seront garanties d'être correctes et exactes.

ET L'INCROYABLE et mis en évidence
Python2 msg=ser.read() le type <c_str> Python3 msg=ser.read() le type <bytes>
Pourtant j'envoie bien je pense en byte. Même constat si on discute uart avec un pc et le raspberry
Effectivement la seul fois que j'envoie un string c'est avec python2 emetteur
Mais si on exécute le script python2 avec python3 il nous dit bytes
Le code envoie un seul caractère et je n'ai pas réussit à envoyer une chaine de caractère

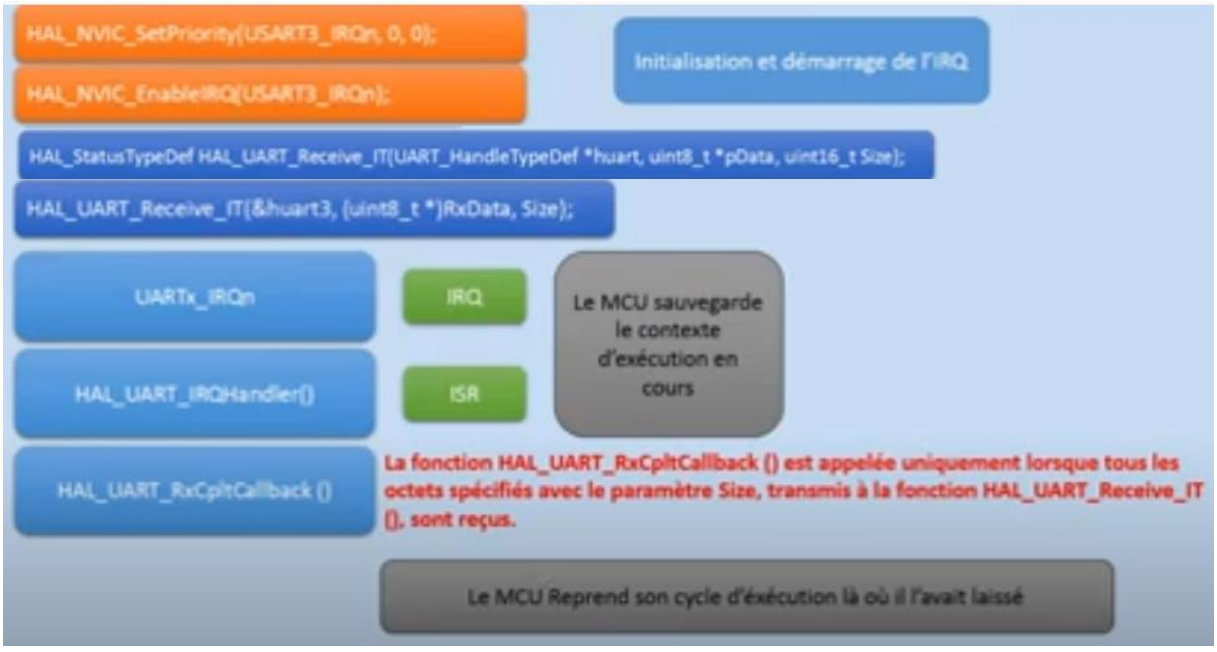


Recevoir des données sur le port UART en mode interruption

Une information va arriver sur le port uart, un évènement qui va être pris en charge par le NVIC qui va appeler l'ISR avec la priorité la plus haute (pour que le NVIC soit disponible pour la prise en charge d'autres évènements)

L'ISR va appeler une fonction de callback (Interrupt service routine) et l'ISR doit être aussi libérer pour d'autres instructions. Une fois les instructions du callback exécutés, l'ISR est rappelé pour remettre le flag d'interruptions à 0.
Ensuite la CPU reprend là où elle en était au moment de l'interruption.
Aller plus loin dans le registre car les callbacks sont directement appelés dans l'ISR.

Initialiser et démarrer le service d'interruption (Gérer par deux fonctions générer par cubemx)
Une autre fonction non générer par cubemx la fonction receive (soit dans le void setup ou le while).
On a le prototype ensuite l'appelle, quand on reçoit une donnée sur le port uart, cela déclenche une requête d'interruption et va appeler une fonction IRQn, elle-même va appeler IRQHandler.
Parallèlement à l'appel sur une des deux fonctions, ka cpu sauvegarde le contexte d'exécution en cours pour reprendre où il en était quand on aura exécuté les instructions.
Donc la fonction IRQHandler appelle RXCompletCallback seulement si le nombre d'octet reçu correspond au paramètre size de la fonction UART_Receive_IT (pas de fonction rxcomplete callback)
(déconseille de mettre nos instruction dans la fonction IRQHandler car il va avoir un problème avec les drapeau d'interruption) Alors utiliser le RXCompletCallback (c'est même encore galère).
Un fois terminer le cpu reprend où il en était.



Si la fonction UART_receive_IT n'est pas dans la boucle il faudra la réactiver (la boucle while(1) a IRQn, IRQHandler,RxCpltCallback). Après RxCpltCallback ou remettre dans le while(1).

Dans le manual usart

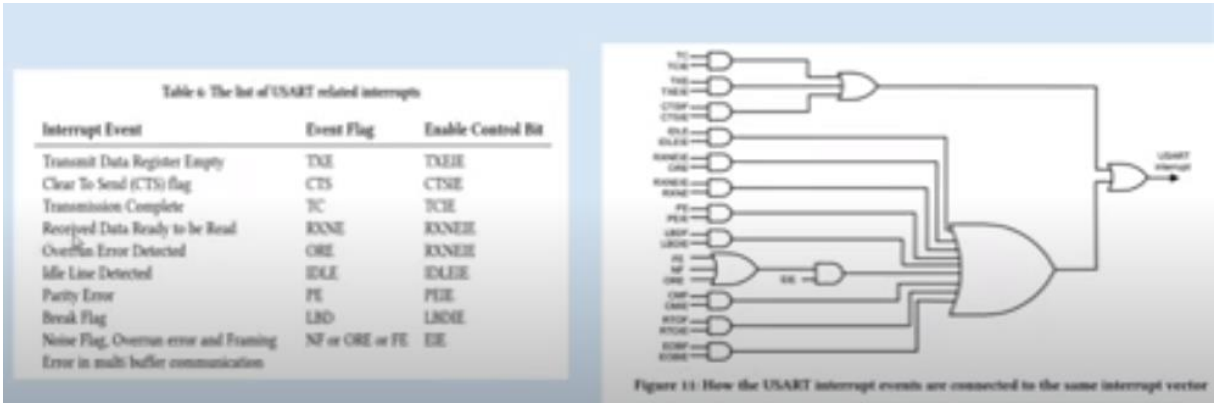


Table 6. The list of USART related interrupts

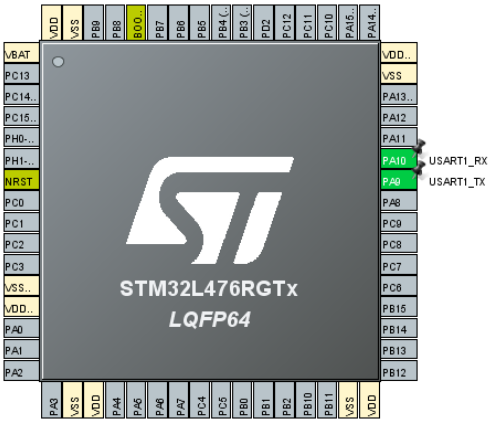
Interrupt Event	Event Flag	Enable Control Bit
Transmit Data Register Empty	TXE	TXEIE
Clear To Send (CTS) flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	RXNEIE
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multi buffer communication	NF or ORE or FE	EIE

Figure 11. How the USART interrupt events are connected to the same interrupt vector

Programme Stm32CubeMX

Téléchargement : https://www.st.com/en/development-tools/stm32cubemx.html

Téléchargement : https://www.st.com/en/development-tools/stm32cubeide.html



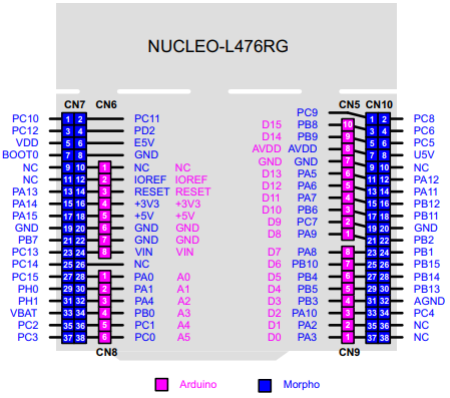
USART1 9600Baud voir en fonction du clock

NVIC enable USART1 global iterrupt

GND----------CN7(20)

TXD----------CN10(33) PA10 USART1_RX

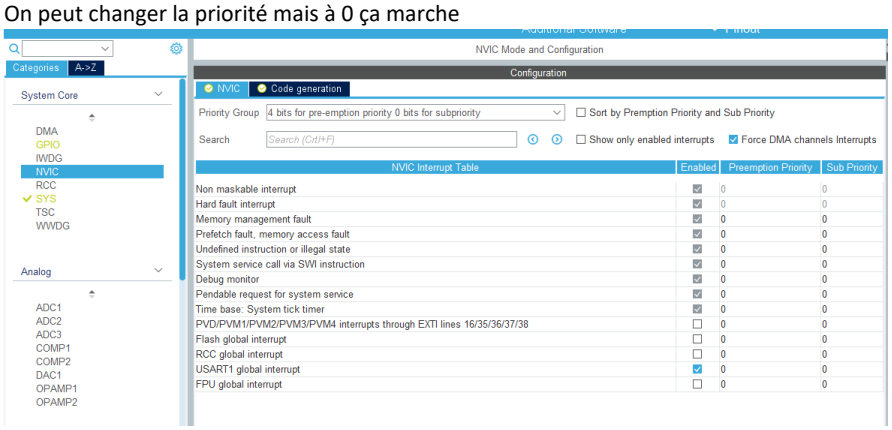
RXD----------CN10(21) PA9 USART1_TX



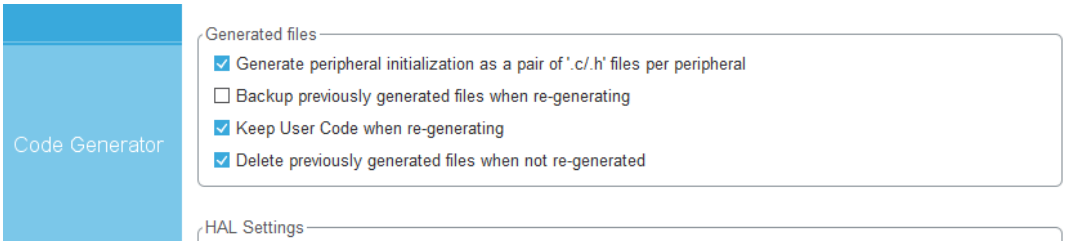
Connectivity: USART1 /Asynchrone/9600bauds

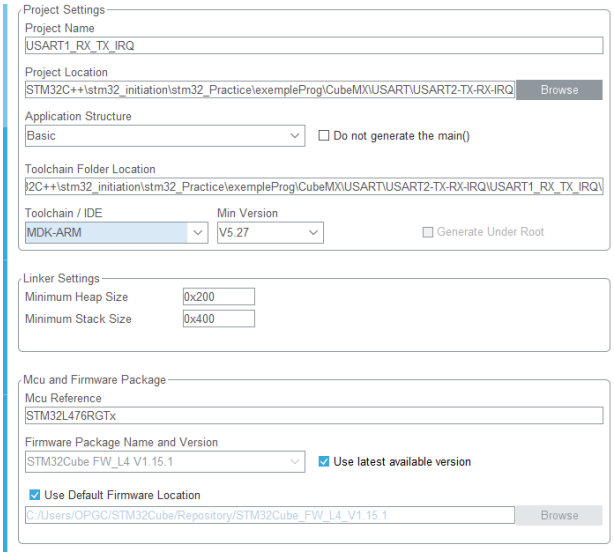
Pin N...	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pull-down	Maximum o...	Fast...	User L...	Modifie
PA9	USART1_TX	n/a	Alternate Function Push Pull	No pull-up and no pull-down	Very High	n/a		☐
PA10	USART1_RX	n/a	Alternate Function Push Pull	No pull-up and no pull-down	Very High	n/a		☐

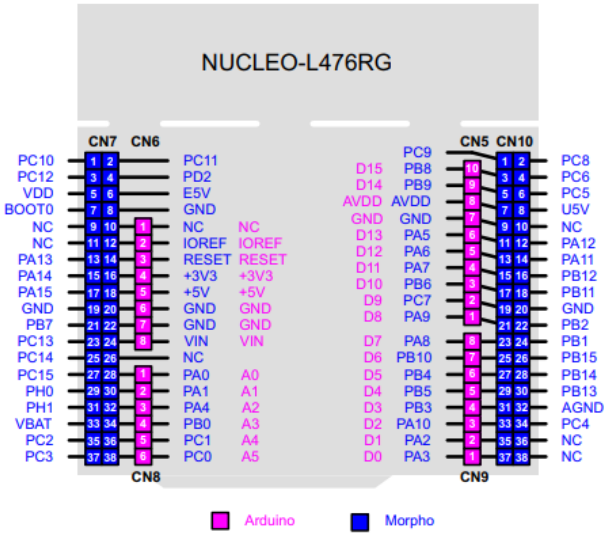
On peut changer la priorité mais à 0 ça marche



Créer le projet pour avoir les codes usart.h et gpio.h







```c
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "usart.h"
#include "gpio.h"
#include "string.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
#define BUFFERDATA 2
#define BUFFERRX 50

char cartInit[30] = "\ncarte INIT\n"; //envoi avant while
char rxOk[30] = "\nle message est reçu est:\n";
char rxBuffer[BUFFERRX];

uint8_t newmsg=0, rxData[BUFFERDATA], rxIndex=0, enter=10;//avec termite sur pc cocher envoi la touche entree

/* USER CODE END PTD */
```

```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
   //char *welcome="bonjour|\n\r";
  HAL_UART_Transmit(&huart1, (uint8_t *)cartInit, strlen(cartInit),500);
  HAL_UART_Receive_IT(&huart1, rxData,1);
  //je veux 1 octet a chaque fois que je reçois

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    if(newmsg){
      HAL_UART_Transmit(&huart1, (uint8_t *)rxOk, strlen(rxOk),500);

      HAL_UART_Transmit(&huart1, (uint8_t *)rxBuffer, strlen(rxBuffer),500);

      for(int i=0; i< BUFFERRX; i++){ rxBuffer[i]=0;  }
        //nettoyer le tableau rxbuffer
      newmsg=0;
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
}
```

```c
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

  if(huart->Instance== USART1){
    if(rxData[0] != enter){
         rxBuffer[rxIndex] = rxData[0] ;//copie dans l'indie 0
         rxIndex++;
    }else{
      newmsg=1;
      rxIndex=0;
    }
    HAL_UART_Receive_IT(&huart1, rxData,1);
  }
}
//fonction en cas de bug
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart){
    if(huart->ErrorCode == HAL_UART_ERROR_ORE)
    {  HAL_UART_Receive_IT(&huart1, rxData,1);}
}
/*
liste des codes erreur
UART Error Code
HAL_UART_ERROR_NONE No error
HAL_UART_ERROR_PE Parity error
HAL_UART_ERROR_NE Noise error
HAL_UART_ERROR_FE Frame error
HAL_UART_ERROR_ORE Overrun error
HAL_UART_ERROR_DMA DMA transfer error
*/
/* USER CODE END 4 */
```

Code Python3 raspberryPi
mais changer le enter=13 ce qui équivaut à CR carriage return = ascii : 13 = \r

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Test du port série
#reception carte stm init puis emet salut et on reçoit salut
import serial
ser = serial.Serial("/dev/ttyS0", baudrate=9600, timeout=10.0)
print("Port serie :  " + ser.name)

print("\n-----Carte stm32 si ok-----\n")
message = ser.readline()
print(type(message))
print(message)
chaine = message.decode()
print(type(chaine))
print("voici le message recu du stm32: "+ chaine)

print("\n-----Carte stm32 si ok-----\n")
messageAEnvoyer = "Salut\r"
ser.write(messageAEnvoyer.encode())
print("le message envoye a la carte est: "+messageAEnvoyer)

print("\n-----Echo de la carte stm32-----\n")
echo = ser.readline()
print(type(echo))
print(echo)
chaineEcho = echo.decode()
print(type(chaineEcho))
print(chaineEcho)
ser.close()
```

```
>>> %Run recpt_emet_stm32IRQ.py
  Port serie :  /dev/ttyS0

  -----Carte stm32 si ok-----

  <class 'bytes'>
  b'carte INIT'
  <class 'str'>
  voici le message recu du stm32: carte INIT

  -----Carte stm32 si ok-----

  le message envoye a la carte est: Salut

  -----Echo de la carte stm32-----

  <class 'bytes'>
  b'le message est recu est:Salut'
  <class 'str'>
  le message est recu est:Salut
```

Python2

```
Port serie :  /dev/ttyS0

-----Carte stm32 si ok-----

<type 'str'>
carte INIT
voici le message recu du stm32: carte INIT

-----Carte stm32 si ok-----

le message envoye a la carte est: Salut

-----Echo de la carte stm32-----

<type 'str'>
le message recu de stm est: le message est recu est:Salut
```

Recevoir des données transmises par l'uart ordinateur Vers STM32 en mode polling et sous forme de question réponse.

Le pc envoie une question le micro-contrôleur envoie un message de confirmation et la réponse sera le même message.
3 modes pour recevoir avec l'uart :
Polling : (opération bloquante la cpu est bloqué) si l'opération est appelée la cpu ne charge plus rien tant que la réception est en cours. Et riense fait, si la bien reçu l'information d'être en réception car la cpu peut être occupée à autre chose.
Interupt : la tâche du microcontrôleur est interrompue, une interruption se lève et la cpu procède à l'écoute du port et de la réception de la trame et ensuite la cpu reprend ses activités là où elle est à laisser. Aucune réception ne sera manquée.
Il faut gérer l'interruption et la fonction de callback:
DMA : le mode DMA c'est ce qui a de mieux pour l'approche d'une application temps réel.
Prise en compte :
Par voie d'interruption la cpu est mobilisé tout le temps de l'interruption donc tant que des trames arrivent la cpu ne fait pas autres choses elle est bloquée.
Alors ici la DMA à chaque réception de trame elle lève une interruption lorsque la trame n'est pas réussite, mais complète ce qui permet de gagner un peu de temps.

MODE POLLING

```
/* USER CODE BEGIN PV */
char salut[30]="\n Allo l'ordi\n";
char rxOk[30]="\n nouveau message!\n";

char rxBuffer[30];//c'est ce qu'on reçoit depuis l'ordi

/* USER CODE END PV */
```

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
//on prévient que la carte à démarrer on envoit salut à l'ordi
HAL_UART_Transmit(&huart1,(uint8_t*) salut, strlen(salut), 500);
//HAL_Delay(500);
```

```
  while (1)
  {  // la chaine qu'on reçoit stocké dans rxBuffer
    // et 30 la longueur du tableau rxBuffer
//chaine connu à l'avance ou le sizeof
    HAL_UART_Receive(&huart1, (uint8_t*)rxBuffer,30,500);
    /* USER CODE END WHILE */
    //avant de faire on s'assure qu'on a reçu quelque chose
    volatile uint8_t newMesg=0;
    for(int i=0;i<=strlen(rxBuffer); i++){
     if(rxBuffer[i] != 00){
       newMesg=1; //on a reçut on met la variable
     }
    }
    if(newMesg){
      if(HAL_UART_Transmit(&huart1,(uint8_t*) rxOk, strlen(rxOk), 500) != HAL_OK)
         Error_Handler();
       if(HAL_UART_Transmit(&huart1,(uint8_t*) rxBuffer, strlen(rxBuffer), 500) != HAL_OK)
         Error_Handler();
    }
    //on efface le message reçut
     for(int i=0; i<=strlen(rxBuffer);i++){
          rxBuffer[i]=00;
     }

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Attention au modèle HAL le Error_Handler à une boucle while(1) on peut rester coincé donc on peut modifier et traiter l'interruption le code erreur sinon on ne l'utilise pas.
Tuto super :
https://www.carnetdumaker.net/forum/topics/140-les-stm32-en-videos/#partie-12-recevoir-des-donnees-depuis-le-pc-par-uart-interrupt-mode