Tutoriels STM32F0

Périphériques standards

UART & printf ()

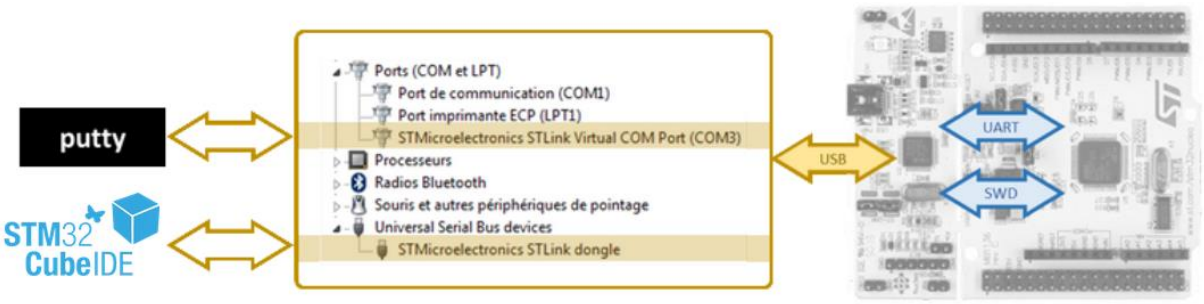
Arouna DARGA – enseignant chercheur Sorbonne Université
arouna.darga@sorbonne-universite.fr

Table des matières

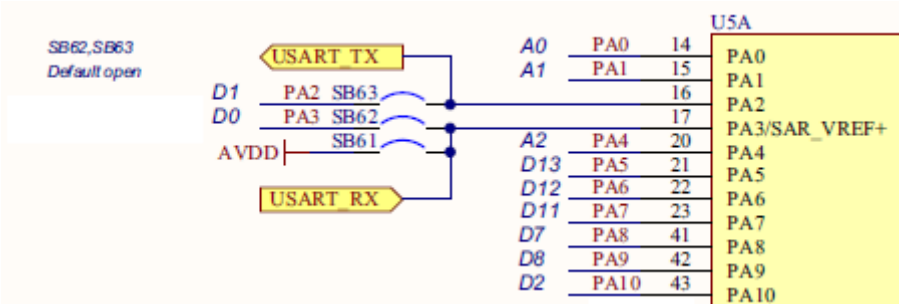
1. Console de debug

Lorsque vous ouvrez le gestionnaire de périphériques de Windows®, vous pouvez voir que le fait de connecter la carte Nucleo STM32F072RB à l'ordinateur par un câble USB, ajoute 2 périphériques à la liste des périphériques. Le dongle **ST-Link** est le dispositif que nous utilisons déjà pour programmer et déboguer le MCU. Le deuxième dispositif est un **port COM virtuel** (COM3 dans l'exemple ci-dessous). Ce port COM peut être utilisé pour échanger des données utilisateur entre l'ordinateur et le MCU. Vous pouvez utiliser n'importe quel programme de terminal série.

Nous utiliserons ici du Putty (www.putty.org). Télécharger l'exécutable.



Ce port **COM** virtuel est connecté aux broches **UART** (ou **USART**) de du MCU. En regardant les schémas, il apparaît que **PA2** est utilisé pour **TX** (transmission) et **PA3** pour **RX** (réception).



- **UART** signifie **U**niversal **A**synchronous **R**eceiver **T**ransmitter
- **USART** signifie **U**niversal **A**synchronous **S**ynchronous **R**eceiver **T**ransmitter

Dans un schéma de **communication synchrone**, il y a un fil supplémentaire qui transporte **un signal d'horloge**. Ceci est facultatif, et nous n'utiliserons que le protocole **asynchrone (UART)**. Les périphériques STM32 sont pour la plupart des USART qui peuvent fonctionner dans les deux modes. Dans les paragraphes suivants, **les termes USART ou UART sont utilisés sans aucune différence**.

2. Configuration de l'USART du STM32F072RB

En ouvrant la fiche technique du STM32F072, on peut constater que les broches PA2 et PA3 sont associées au périphérique USART2, en mode AF1 (fonction alternative 1) :

Table 14. Alternate functions selected through GPIOA_AFR registers for port A

Pin name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA0		USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1	USART4_TX			COMP1_OUT
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2	USART4_RX	TIM15_CH1N		
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3				COMP2_OUT
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4				
PA4	SPI1_NSS, I2S1_WS	USART2_CK		TSC_G2_IO1	TIM14_CH1			
PA5	SPI1_SCK, I2S1_CK	CEC	TIM2_CH1_ETR	TSC_G2_IO2				
PA6	SPI1_MISO, I2S1_MCK	TIM3_CH1	TIM1_BKIN	TSC_G2_IO3	USART3_CTS	TIM16_CH1	EVENTOUT	COMP1_OUT
PA7	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM1_CH1N	TSC_G2_IO4	TIM14_CH1	TIM17_CH1	EVENTOUT	COMP2_OUT
PA8	MCO	USART1_CK	TIM1_CH1	EVENTOUT	CRS_SYNC			
PA9	TIM15_BKIN	USART1_TX	TIM1_CH2	TSC_G4_IO1				
PA10	TIM17_BKIN	USART1_RX	TIM1_CH3	TSC_G4_IO2				
PA11	EVENTOUT	USART1_CTS	TIM1_CH4	TSC_G4_IO3	CAN_RX			COMP1_OUT
PA12	EVENTOUT	USART1_RTS	TIM1_ETR	TSC_G4_IO4	CAN_TX			COMP2_OUT
PA13	SWDIO	IR_OUT	USB_NOE					
PA14	SWCLK	USART2_TX						
PA15	SPI1_NSS, I2S1_WS	USART2_RX	TIM2_CH1_ETR	EVENTOUT	USART4_RTS			

Nous allons d'abord implémenter une fonction qui initialise **USART2** pour qu'il fonctionne avec le port **COM virtuel**.

Ouvrez le projet **my_project** et ajoutez le prototype de fonction ci-dessous dans le fichier en-tête **bsp.h** :

```
11 #include "stm32f0xx.h"
12
13 /*
14  * fonctions du driver LED
15  */
16
17 void    BSP_LED_Init    (void);
18 void    BSP_LED_On  (void);
19 void    BSP_LED_Off (void);
20 void    BSP_LED_Toggle  (void);
21
22 /*
23  * fonctions du driver de bouton poussoir
24  */
25
26 void    BSP_PB_Init     (void);
27 uint8_t BSP_PB_GetState (void);
28
29 /*
30  * Fonction initialisation USART2 pour etre utiliser comme Console Debug
31  */
32
33 void    BSP_Console_Init    (void);
34
35
36
37 #endif /* BSP_INC_BSP_H_ */
38
```

Ensuite, il faut implémenter le code C dans le fichier source **bsp.c**. **Vous devez vous référer au manuel de référence pour comprendre le code ci-dessous** (Page . En gros, vous devez configurer 2 périphériques :

- **GPIOA** → Configurer les broches **PA2** et **PA3** en mode **AF1 (USART2)**
- **USART2** → Configurer les paramètres de communication (tels que la vitesse de communication (Baud rate)

```c
/*
 * BSP_Console_Init()
 * USART2 @ 115200 Full Duplex
 * 1 start - 8-bit - 1 stop
 * TX -> PA2 (AF1)
 * RX -> PA3 (AF1)
 */

void BSP_Console_Init()
{
	//activer horloge du peripherique GPIOA
	// mettre le bit b17 du registre RCC_AHBENR a '1'
	// voir page 128 du manuel technique (User Manuel) du
Microcontrôleur STM32F072RB
	// le bit b17 de RCC_AHBENR est egalement defini =
RCC_AHBENR_GPIOAEN dans le fichier stm32f0xx.h
	RCC->AHBENR |= (1<<17);   // ou  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;


	// Choisir le mode Alternate Function (AF) pour les broches PA2 et
PA3
	// pour PA2 : ecrire "10" sur les bits b5b4 du registre GPIOA_MODER
	GPIOA->MODER |=(1<<5);
	GPIOA->MODER &=~(1<<4);
	// pour PA3 : ecrire "10" sur les bits b7b6 du registre GPIOA_MODER
	GPIOA->MODER |=(1<<7);
	GPIOA->MODER &=~(1<<6);

	// Choisir la fonction AF1 (USART2) pour les broches PA2 et PA3
	// pour PA2 : ecrire "0001" sur les bits b11b10b9b8 du registre
GPIOA_AFRL = GPIO->AFR[0]
	GPIOA->AFR[0] &= ~(1<<11);
	GPIOA->AFR[0] &= ~(1<<10);
	GPIOA->AFR[0] &= ~(1<<9);
	GPIOA->AFR[0] |=  (1<<8);

	// Choisir la fonction AF1 (USART2) pour les broches PA2 et PA3
	// pour PA3 : ecrire "0001" sur les bits b15b14b13b12 du registre
GPIOA_AFRL = GPIO->AFR[0]
	GPIOA->AFR[0] &= ~(1<<15);
	GPIOA->AFR[0] &= ~(1<<14);
	GPIOA->AFR[0] &= ~(1<<13);
	GPIOA->AFR[0] |=  (1<<12);

	//activer horloge du peripherique USART2
	// mettre '1' le bit b17 du registre (RCC_APB1ENR)
	// voir page 131 du manuel technique (User Manuel) du
Microcontrôleur STM32F072RB
	RCC -> APB1ENR |= (1<<17);

	//Reset de la configuration de USART2 : Mise a zero des registres
de control de USART2
	//USART2_CR1, USART2_CR2 , USART2_CR3
	// On utilise les valeurs par defaut
	// 8-bits de donnees
	// 1 bit START
	// 1 bit STOP
	// desactivation de CTS/RTS
	USART2->CR1 = 0x00000000;
	USART2->CR2 = 0x00000000;
```

```
	USART2->CR3 = 0x00000000;

	// Choisir la source PCLK (APB1) comme source horloge de USART2 :
Valeur par defaut
	// PCLK -> 48 MHz
	// mettre "00" sur les bits b17b16 du registre (RCC_CFGR3)
	// voir page 140 du reference manual
	RCC->CFGR3 &= ~(1<<17);
	RCC->CFGR3 &= ~(1<<16);

	// Configuration du Baud Rate = 115200
	// sans oversampling 8 bits (OVER8=0) et Fck=48MHz, USARTDIV =
48E6/115200 = 416.6666
	// BRR = 417 -> Baud Rate = 115107.9137 -> 0.08% erreur
	// avec oversampling 8 bits (OVER8=1) and Fck=48MHz, USARTDIV =
2*48E6/115200 = 833.3333
	// BRR = 833 -> Baud Rate = 115246.0984 -> 0.04% error (Meilleur
choix)

	// choix oversampling 8 bits (OVER8=1)
	// mettre a '1' le bit b15 de USART2_CR1
	USART2->CR1 |= (1<<15);
	// ecrire la valeur du Baud Rate dans le registre USART2_BRR
	USART2->BRR = 833;

	// Activer la transmision : ecrire '1' sur le bit b3 de USART2_CR1
	USART2->CR1 |= (1<<3);

	// Activer la reception : ecrire '1' sur le bit b2 de USART2_CR1
	USART2->CR1 |= (1<<2);

	// activer le peripherique USART2 en dernier
	// mettre a '1' le bit bit b0 de USART2_CR1
	USART2->CR1 |= (1<<0);
}
```

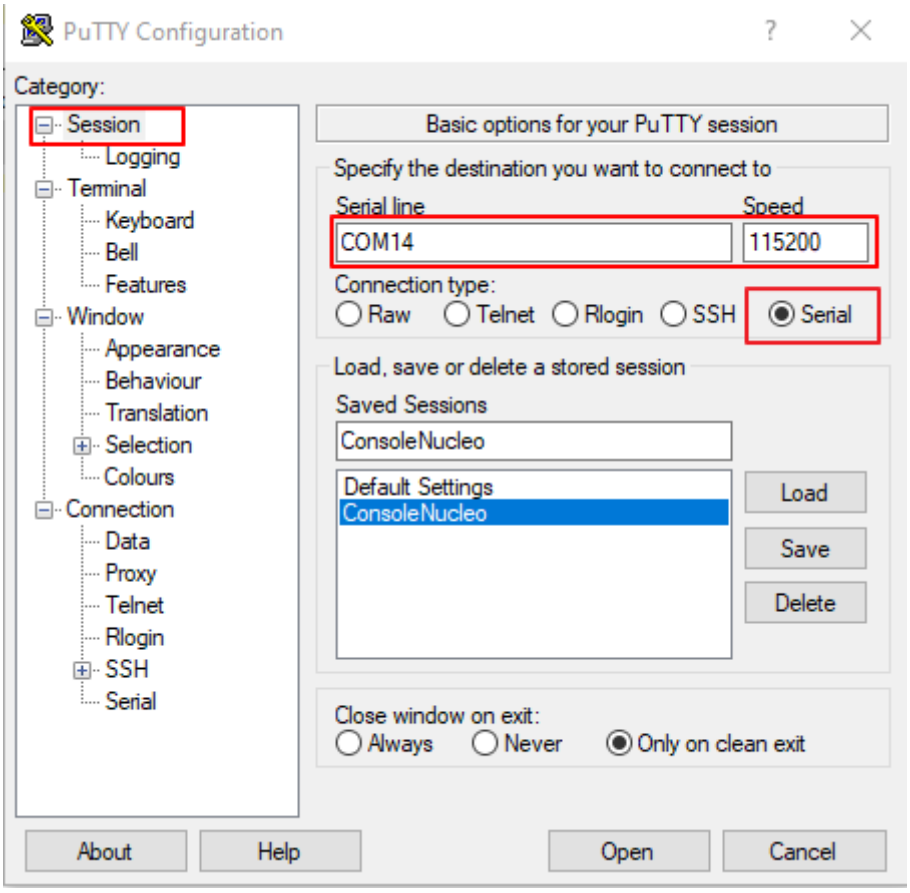
Nous pouvons maintenant réaliser quelques essais dans le fichier **main.c** :

```
void main()
{
	uint8_t	i, envoi;

	// configuration de horloges du system
	SystemClock_Config();

	// Initialiser les broches pour LED et Bouton
	BSP_LED_Init();
	BSP_PB_Init();

	// Initialiser la liason serie : Console de Debug
	BSP_Console_Init();

	// Boucle Infini
	while(1)
	{
		// Si le bouton bleu est appuyer
		if (BSP_PB_GetState() == 1)
		{
			BSP_LED_On();	// Allumer la led verte

			// Envoyer le caractere '#' une seule fois
		if (envoi == 0)
		{

		// attendre que le buffer de transmission soit vide ou que
		// la transmission precedente soit finie
		// Bit b6 du registre USART2_ISR
		// '1' : transmission precedente terminee = buffer vide
		// '0' : transmission encoours : buffer non vide

			while ((USART2->ISR & (1<<6)) != (1<<6));

		// arriver ici = buffer vide, on peut envoyer une data

				// envoi du caractere
				USART2->TDR = '#';

				//mise a '1' du drapeau envoi
				envoi = 1;
			}
		}

		// Si le bouton est non appuyer
		else
		{
		BSP_LED_Off();	// eteindre la led verte
		envoi  = 0;	// mettre le drapeau envoi a '0' : pas envoi
		}
	}
}
```

Le code ci-dessus appelle la fonction d'initialisation. Ensuite, chaque fois que vous appuyez sur le bouton utilisateur, le caractère "#" est placé dans le registre de transmission des données (**TDR**) de l'**USART2**, après vous être assuré que l'**USART2** n'est pas occupé en vérifiant l'état de son drapeau Transfert terminé (**TC**).

Sauvegardez tout, construisez le projet et exécutez.

Lancez un programme de terminal série **COM sur COM3** (votre numéro COM est probablement différent). Configurer le **baudrate à 115200 bauds**. L'exemple ci-dessous illustre la configuration de **Putty** :



Jouez ensuite avec le **bouton utilisateur** de Nucleo (bleu). Vous devriez voir un **"#"** s'afficher à chaque fois que vous appuyez sur le bouton :

Bien ! Mais nous n'avons toujours pas de fonction **printf()** disponible. Nous ne pouvons envoyer les caractères qu'un par un. N'ayez pas peur, **vous n'aurez pas à coder votre propre fonction printf()** (mais nous n'en sommes pas très loin...)

	Enregistrez votre projet

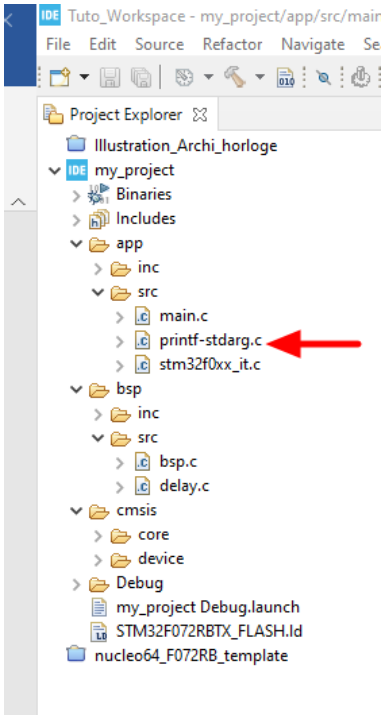
3. Mise en œuvre de printf()

Une solution serait d'utiliser la fonction **printf()** qui est fournie avec le compilateur **GCC newlib-nano** et de rediriger sa sortie vers **USART2**. Ce n'est pas ce que nous allons faire.

Une autre alternative consiste à obtenir une version allégée de la fonction printf() ici :

https://www.menie.org/georges/embedded/printf-stdarg.c

Téléchargez le fichier source **printf-stdarg.c** à partir du lien ci-dessus. Copiez-le dans votre dossier /app/src et rafraîchissez la vue de l'explorateur de projet dans STM32CubeIDE :



Pour que cette version **printf()** fonctionne, vous devez fournir votre **propre implémentation d'une fonction putchar()**. Cette implémentation est simple, et c'est en gros ce que nous allons faire

Ouvrez le ficher **printf-stdarg.c** et repérez la fonction **princhar()** au-dessus du code :

```
#include <stdarg.h>

static void printchar(char **str, int c)
{
	extern int putchar(int c);

	if (str) {
		**str = c;
		++(*str);
	}
	else (void)putchar(c);
}
```

Au lieu d'écrire une fonction putchar(), nous mettrons directement en œuvre les deux lignes de code nécessaires :

```
#include <stdarg.h>
#include "stm32f0xx.h"

static void printchar(char **str, int c)
{

	if (str) {
		**str = c;
		++(*str);
	}
	else
	{
		// attendre que le buffer de transmission soit vide ou que
		// la transmission precedente soit finie
		// Bit b6 du registre USART2_ISR
		// '1' : transmission precedente terminee = buffer vide
		// '0' : transmission encoours : buffer non vide
		while ((USART2->ISR & (1<<6)) != (1<<6));

		// arriver ici = buffer vide, on peut envoyer une data

		// envoi du caractere
		USART2->TDR = c;

	}
}
```

Localisez maintenant les fonctions printf() et sprintf() au bas du code, et changez les noms des fonctions en avec un nom que vous aimez :

```
int mon_printf(const char *format, ...)
{
        va_list args;

        va_start( args, format );
        return print( 0, format, args );
}

int mon_sprintf(char *out, const char *format, ...)
{
        va_list args;

        va_start( args, format );
        return print( &out, format, args );
}
```

Pourquoi changeons-nous les noms de ces fonctions ? Parce que **printf()** et **sprintf()** sont des fonctions connues du **linker**. Conserver ces noms produit **des erreurs de linker** si des bibliothèques standard telles que <**stdio.h**> ne sont pas incluses.

En utilisant la librairie **printf-stdarg.c**, nous avons une fonction mon_printf() autosuffisante qui ne nécessite aucune bibliothèque liée telle que **<stdio.h>.**

Nous devons néanmoins déclarer des prototypes de fonctions quelque part. Faisons cela dans un fichier en-tête **main.h** :

```
/*
 * main.h
 *
 *  Created on: Feb 3, 2021
 *      Author: darga
 */

#ifndef APP_INC_MAIN_H_
#define APP_INC_MAIN_H_

/*
 * printf() et sprintf() de la lib. printf-stdarg.c
 */

int mon_printf    (const char *format, ...);
int mon_sprintf   (char *out, const char *format, ...);

#endif /* APP_INC_MAIN_H_ */
```

Prêt pour un test ?

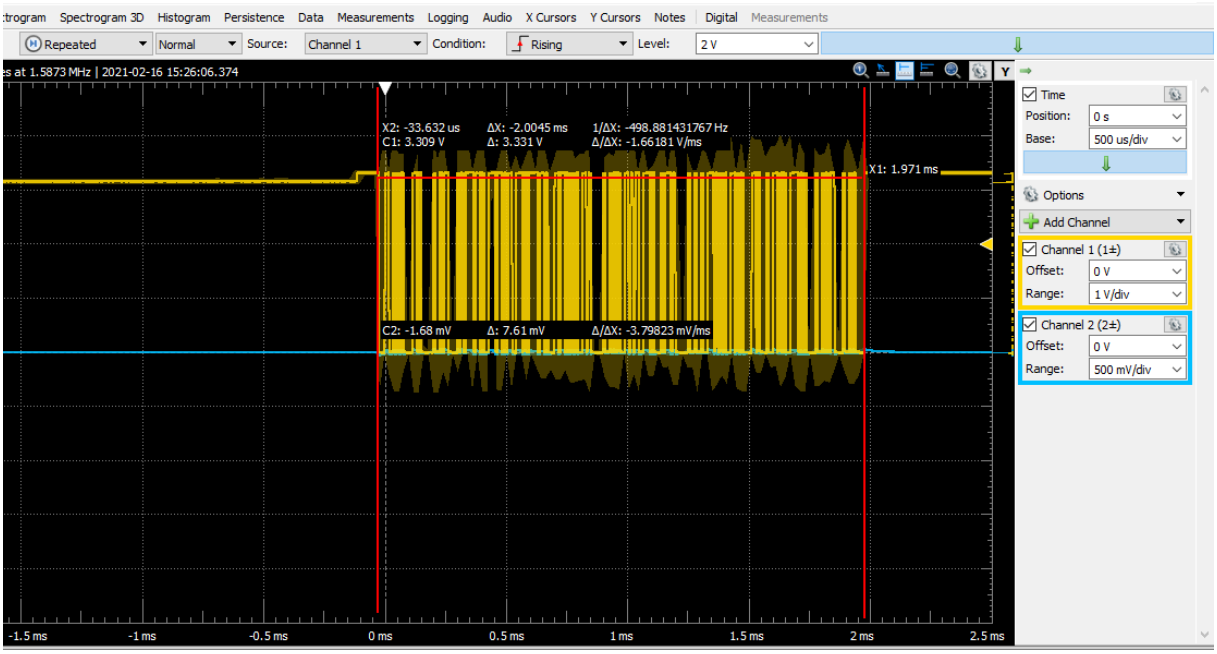
N'oubliez pas d'inclure le **main.h** dans le **main.c** maintenant...

```
#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"
#include "main.h"

static void SystemClock_Config(void);

int main()
{
	uint8_t i, envoi;

	// configuration de horloges du system
	SystemClock_Config();

	// Initialiser les broches pour LED et Bouton
	BSP_LED_Init();
	BSP_PB_Init();

	// Initialiser la liason serie : Console de Debug
	BSP_Console_Init();

	// test du printf
	mon_printf("La Console est ready!\r\n");

	envoi = 0;
	i = 0;

	// Boucle Infini
	while(1)
	{
		// Si le bouton bleu est appuyer
		if (BSP_PB_GetState() == 1)
		{
			BSP_LED_On();	// Allumer la led verte

			// Envoyer le caractere '#i' une seule fois
			if (envoi == 0)
			{

				mon_printf("#%d\r\n", i);

				//mise a '1' du drapeau envoi
				envoi = 1;
				i++;
			}
		}

		// Si le bouton est non appuyer
		else
		{
			BSP_LED_Off();	// eteindre la led verte
			envoi  = 0;	// mettre le drapeau envoi a '0' : pas
envoi
		}
	}
}
```

Sauvegarder, compiler, exécuter et ouvrir un terminal COM.

Vous disposez maintenant d'une printf() fonctionnelle. Vous devez garder à l'esprit que l'utilisation de printf() prend du temps car elle repose sur une interface série lente. Par exemple, le message "La Console est Ready!\r\n" prend environ **2 ms** pour être imprimé à 115200 bauds. C'est assez long si l'on considère ce que le CPU serait capable de faire dans un délai aussi long.

La capture d'écran ci-dessous montre le signal **USART** correspondant au message "La Console est Ready!\r\n". Vous pouvez le sonder en utilisant le connecteur RX disponible sur le dongle **ST-Link** (les connecteurs PA2 et PA3 sont en fait déconnectés de la MCU. Reportez-vous aux schémas de la carte).



Pendant le processus printf(), l'unité centrale passe le plus clair de son temps à attendre que l'USART soit disponible pour l'envoi du prochain personnage. Bien que l'utilisation de printf() soit d'une grande aide lors du débogage, vous devez être conscient qu'elle ralentira considérablement l'exécution du code.

Voici quelques conseils pour améliorer les choses :

- Utilisation de la fonction **Semihosting** au lieu de **USART**. Vous pouvez rediriger la **fonction printf()** vers la **console de débogage via OpenOCD**. Elle ne fonctionne que lorsque vous êtes en mode debug, alors que notre fonction printf() fonctionne même lorsque les applications tournent. Néanmoins, le semi-hébergement peut être plus lent que l'utilisation de l'USART.
- Évitez les longs messages. En fin de compte, vous ne pouvez envoyer qu'un seul caractère en chargeant directement le registre TDR de USART. Ce processus n'implique aucun temps d'attente puisque le processus d'envoi de l'USART est parallèle à l'exécution du CPU. Il n'est pas applicable aux chaînes de caractères mais peut être très utile. Utilisez-le lorsque vous souhaitez surveiller des points de contrôle de votre code en temps réel.
- Vous pouvez entièrement paralléliser le processus d'envoi des chaînes de caractères USART en utilisant le DMA (**plus d'informations à ce sujet plus tard**).
- Le débit en bauds peut être augmenté bien au-delà des débits en bauds standard RS232 puisque le signal réel est transporté par l'USB qui est une interface série rapide. Des vitesses de transmission allant jusqu'à 1 million de bauds fonctionnent parfaitement.
- **STM-Studio®** peut vous aider à espionner les variables, en temps réel, sans aucun effet sur les performances d'exécution.