



Tutoriels STM32F0

Interruptions –
UART – mode réception

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

Table des matières

1.	Utilisation de l'USART en mode RX	3
1.1	Configuration de projet CubeIDE.....	3
1.2	Configuration du périphérique	3
1.3	Comprendre le processus de la réception de caractères : USART RX	4
2.	Activation des interruptions de réception de caractères USART – RX.....	13
2.1	Configuration de USART et NVIC	13
2.2	Gestionnaire d'interruption simple pour USART RX.....	15
2.3	Test de l'interruption RX de l'USART	16
2.4	Mise en mémoire tampon des octets entrants.	17
3.	Résumé.....	21

1. Utilisation de l'USART en mode RX

1.1 Configuration de projet CubeIDE

Pour ce tutoriel nous utilisons le projet « **Tutoriels_Interruptions** » du TP2.

1.2 Configuration du périphérique

Jusqu'à présent, nous avons utilisé **USART2 en mode TX** uniquement, pour envoyer des messages du MCU STM32F072RB vers le PC (utilisation du terminal PuTTY pour afficher les messages) avec la fonction **mon_printf()**. Dans ce tutoriel, nous voulons aussi recevoir des données entrantes dans le MCU STM32F072RB, envoyées depuis le PC.

Ci-dessous se trouve notre fonction qui initialise **USART2**. Cette fonction est utilisée dans les tutoriels précédents. Assurez-vous que les modes **TX** et **RX** sont activés (**ils devraient déjà l'être**). En faisant cela, la **broche PA3** est utilisée comme fonction alternative **RX de l'USART2**.

```
void BSP_Console_Init()
{
    RCC->AHBENR |= (1<<17); // ou RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    // Choisir le mode Alternate Function (AF) pour les broches PA2 et PA3
    // pour PA2 : écrire "10" sur les bits b5b4 du registre GPIOA_MODER
    GPIOA->MODER |= (1<<5);
    GPIOA->MODER &= ~(1<<4);
    // pour PA3 : écrire "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 : écrire "0001" sur les bits b11b10b9b8 du registre GPIOA_AFR1 = 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 : écrire "0001" sur les bits b15b14b13b12 du registre GPIOA_AFR2 = GPIO->AFR[1]
    GPIOA->AFR[1] &= ~(1<<15);
    GPIOA->AFR[1] &= ~(1<<14);
    GPIOA->AFR[1] &= ~(1<<13);
    GPIOA->AFR[1] |= (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 défaut
// 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 transmission : 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 b0 de USART2_CR1
USART2->CR1 |= (1<<0);
}

```

1.3 Comprendre le processus de la réception de caractères : USART RX

Pour analyser le fonctionnement du processus de réception d'un caractère par le MCU STM32 (Périphérique USART2 Rx) nous utilisons :

- Un programme qui émule un terminal sur notre PC. Ce programme permet de transmettre des caractères au MCU STM32 à travers un PORT COM émuler sur le PC via les drivers de STLINK (partie supérieure de la carte nucléo). Nous utilisons le programme PuTTY qui est léger et simple.
- Un code embarqué dans le MCU STM32 générer via les codes que nous avons développé avec l'IDE STM32CubeIDE. Nous utilisons les codes du projet STM32CubeIDE « **Tutoriels_Interruptions** » développé au TP2.

Ecrire le code pour la fonction **main ()** ci-dessous.

```
// variable globale
uint8_t button_irq = 0;
uint8_t timebase_irq = 0;

int main(void)
{
    uint8_t      rx_byte;

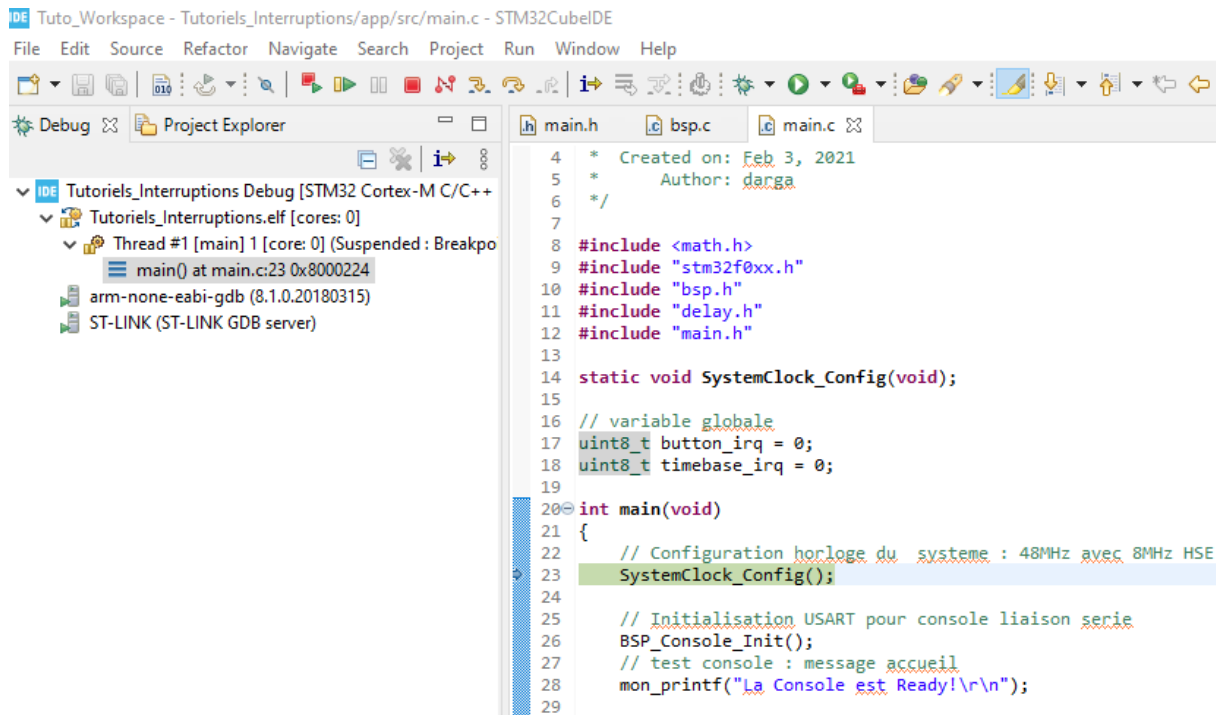
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();

    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");

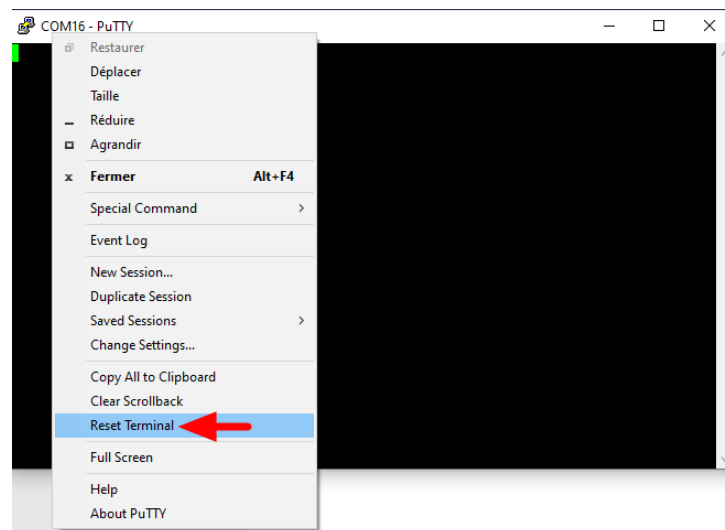
    // boucle principale des applications
    while(1)
    {
        // Ne rien faire...
    }
}
```



Ensuite :

- Sauvegarder et compiler le projet
- Lancez une session de débogage
- Ouvrez un programme de terminal série (Putty) avec les paramètres corrects du port COM.
- Une fois lancé, le débogueur s'arrêtera au début de la fonction main(), attendant que vous lanciez l'exécution.



Nettoyez le terminal **Putty** des messages précédents en cliquant sur l'icône de la fenêtre, puis sur **Reset Terminal**.



Maintenant, lancez l'exécution du programme  pendant quelques secondes, puis appuyez sur le bouton de suspension .

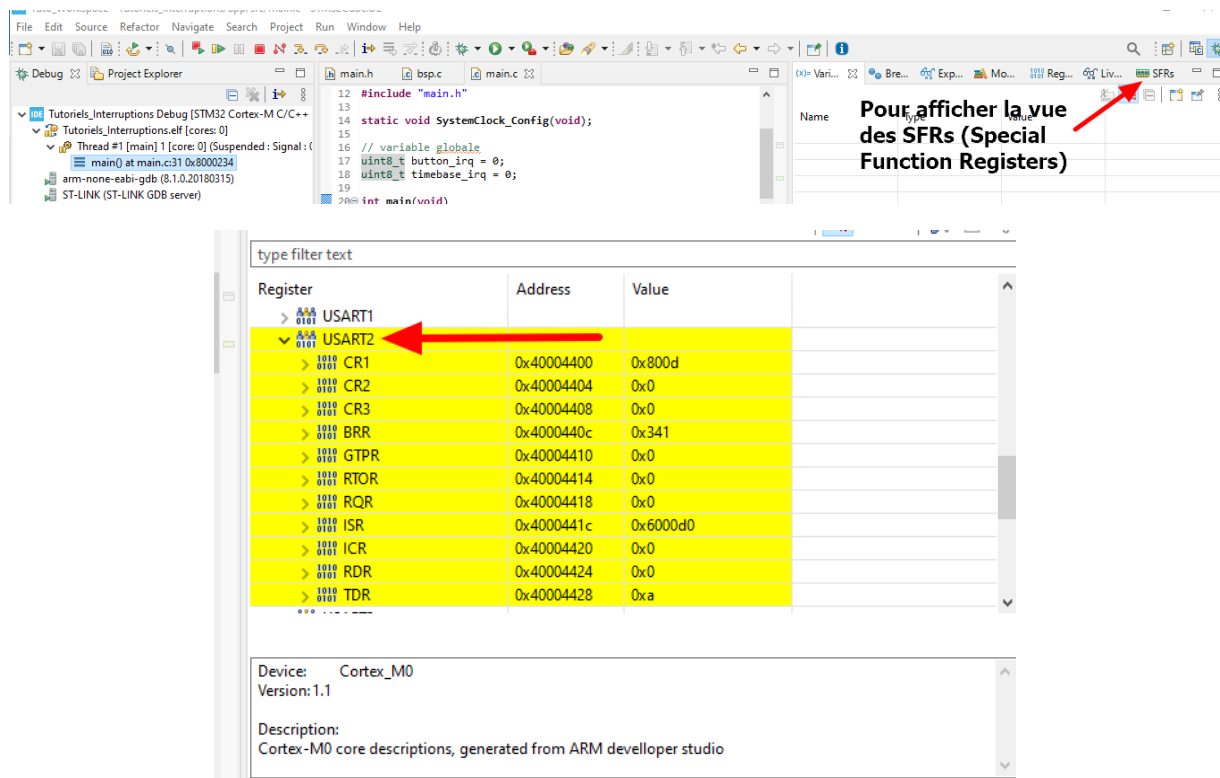
Vous devriez voir le message " *La Console est Ready !*", et le programme est piégé dans votre boucle infinie **while(1)** :

COM16 - PuTTY

La Console est Ready!

```
20 int main(void)
21 {
22     // Configuration horloge du systeme : 48MHz avec 8MHz HSE
23     SystemClock_Config();
24
25     // Initialisation USART pour console liaison serie
26     BSP_Console_Init();
27     // test console : message accueil
28     mon_printf("La Console est Ready!\r\n");
29
30     // boucle principale des applications
31     while(1)
32     {
33         // Ne rien faire....
34     }
35 }
36
37 }
```

Dans la fenêtre de vue **SFRs**, déplier **USART2**



Rafraîchissez la vue du registre USART2 en exécutant l'application quelques secondes de plus 🟢, puis suspendez à nouveau 🟡.

Prenez un moment pour dérouler chacun des registres USART2 et jetez un coup d'œil sur les différents paramètres et les informations disponibles que vous pouvez recueillir à partir de ces registres. Par exemple :

- **CR1, CR2, CR3** sont utilisés pour configurer le fonctionnement de **USART2**. Vous devriez retrouver ici ce que fait la fonction d'initialisation de votre fonction d'initialisation du fichier de code **BSP**.
- **ISR** indique le statut de **USART2**
- **RDR** est le registre de données de réception
- **TDR** est le registre de données de transmission

L'examen de l'**ISR (Interrupt & Status Register, à ne pas confondre avec Interrupt Service Routine)** nous apprend que :

- **USART2** n'est pas occupé actuellement (**BUSY = 0**)
- Le registre de données de réception est vide (**RXNE=0**)
- La dernière transmission sur **TX** est terminée (**TC = 1**)

SFRs 1010 0101 Registers RD X16 X10 X2		
type filter text		
Register	Address	Value
1010 0101 ISR	0x4000441c	0x6000d0
1010 0101 REACK	[22:1]	0x1
1010 0101 TEACK	[21:1]	0x1
1010 0101 WUF	[20:1]	0x0
1010 0101 RWU	[19:1]	0x0
1010 0101 SBKF	[18:1]	0x0
1010 0101 CMF	[17:1]	0x0
1010 0101 BUSY	[16:1]	0x0
1010 0101 ABRF	[15:1]	0x0
1010 0101 ABRE	[14:1]	0x0
1010 0101 EOBF	[12:1]	0x0
1010 0101 RTOF	[11:1]	0x0
1010 0101 CTS	[10:1]	0x0
1010 0101 CTSIF	[9:1]	0x0
1010 0101 LBDF	[8:1]	0x0
1010 0101 TXE	[7:1]	0x1
1010 0101 TC	[6:1]	0x1
1010 0101 RXNE	[5:1]	0x0
1010 0101 IDLE	[4:1]	0x1
1010 0101 ORE	[3:1]	0x0
1010 0101 NF	[2:1]	0x0
1010 0101 FE	[1:1]	0x0
1010 0101 PE	[0:1]	0x0

Vous pouvez également remarquer que le registre **TDR** contient l'octet **0x0A**, qui correspond au tout dernier octet que nous avons envoyé dans le message "**La Console est Ready!\r\n**" (c'est-à-dire le code ASCII du caractère de nouvelle ligne '\n').

Maintenant, faites EXACTEMENT ce qui suit :

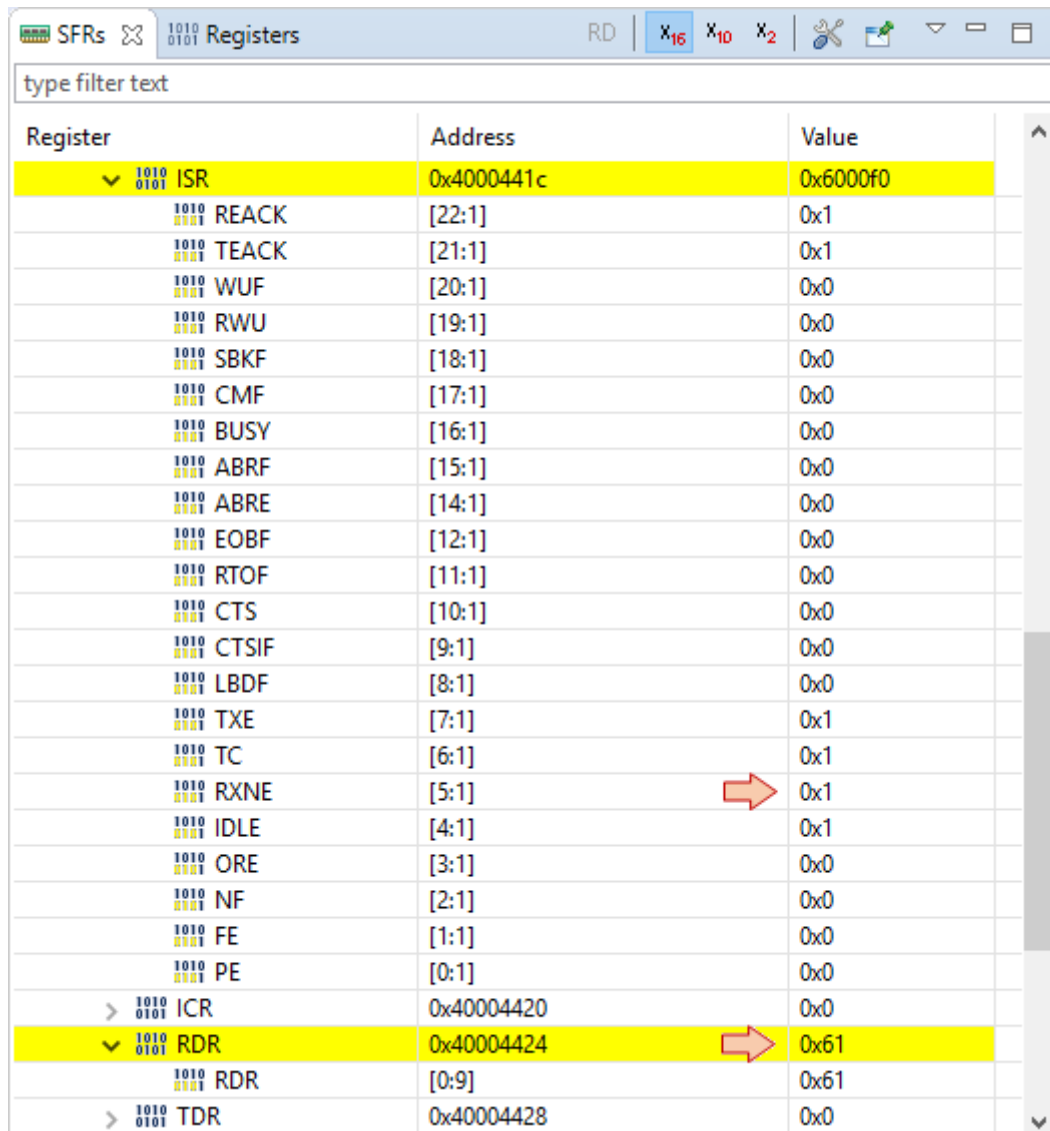
1. Dans le débogueur, appuyez sur le bouton pour relancer l'exécution du programme
2. Sélectionnez la fenêtre du terminal (Putty) pour la mettre en évidence.
3. Appuyez sur la touche 'a' du clavier de l'ordinateur (une seule fois). Vous ne verrez rien se passer dans la fenêtre du terminal, c'est normal.
4. Maintenant, appuyez sur le bouton pour mettre en pause l'exécution

En faisant cela, vous avez demandé au programme terminal (Putty) d'envoyer l'octet 'a' à la MCU sur sa broche **USART2 RX**.

Si vous avez bien fait les étapes ci-dessus, vous devriez voir des changements dans les registres du périphérique USART2 :

L'**ISR** a changé. Déroulez-le et observez que le bit **RXNE (RX Not Empty)** est maintenant à '1', ce qui nous indique qu'une donnée est disponible dans le registre **RDR**.

Le **RDR** contient maintenant l'**octet 0x61**, qui est le code **ASCII du caractère 'a'** (celui que vous avez appuyé).



Register	Address	Value
1010 0101 ISR	0x4000441c	0x6000f0
1010 0101 REACK	[22:1]	0x1
1010 0101 TEACK	[21:1]	0x1
1010 0101 WUF	[20:1]	0x0
1010 0101 RWU	[19:1]	0x0
1010 0101 SBKF	[18:1]	0x0
1010 0101 CMF	[17:1]	0x0
1010 0101 BUSY	[16:1]	0x0
1010 0101 ABRF	[15:1]	0x0
1010 0101 ABRE	[14:1]	0x0
1010 0101 EOBIF	[12:1]	0x0
1010 0101 RTOF	[11:1]	0x0
1010 0101 CTS	[10:1]	0x0
1010 0101 CTSIF	[9:1]	0x0
1010 0101 LBDF	[8:1]	0x0
1010 0101 TXE	[7:1]	0x1
1010 0101 TC	[6:1]	0x1
1010 0101 RXNE	[5:1]	0x1
1010 0101 IDLE	[4:1]	0x1
1010 0101 ORE	[3:1]	0x0
1010 0101 NF	[2:1]	0x0
1010 0101 FE	[1:1]	0x0
1010 0101 PE	[0:1]	0x0
> 1010 0101 ICR	0x40004420	0x0
1010 0101 RDR	0x40004424	0x61
1010 0101 RDR	[0:9]	0x61
> 1010 0101 TDR	0x40004428	0x0

Répétez les étapes ci-dessus, mais au lieu d'appuyer sur la touche 'a' à l'étape 3, appuyez sur la touche 'b' (une seule fois). Vous verrez que l'octet 0x62 est maintenant arrivé dans le registre **RDR**. Vous pouvez continuer à répéter les étapes ci-dessus avec différentes touches du clavier, et vérifier que la dernière touche enfoncée est toujours arrivée dans le registre **RDR**.

Si vous essayez d'appuyer sur 2 touches consécutives du clavier à partir de l'application terminal, vous verrez que seule la première est stockée dans le registre **RDR**. De plus, le drapeau **ORE (Overrun error)** est mis à 1. Ce bit est mis à 1 par le matériel lorsque les données en cours de réception ne peuvent pas être transférées dans le **RDR** parce qu'il n'est pas vide. Si cela se produit, le processus **RX** ne fonctionnera plus jusqu'à ce que vous effaciez **ORE en écrivant 1 au bit ORECF**, dans le registre **ICR**. Vous pouvez faire cela à partir de la vue registre du débogueur et vous assurer que ORE se réinitialise.

Vous pouvez désactiver de façon permanente la détection d'erreur de dépassement en mettant le bit **OVRDIS** dans le registre **CR3**.

Terminez la session de débogage et modifiez la fonction `main()` comme indiquer ci-dessous :

```
// variable globale
uint8_t button_irq = 0;
uint8_t timebase_irq = 0;

int main(void)
{
    uint8_t rx_byte;
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    // boucle principale des applications
    while(1)
    {
        // is il y a quelque chose a lire dans le registre RDR de USART2
        if ((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
        {
            // lire le caractère et le renvoyer sur le PC
            rx_byte = USART2->RDR;
            mon_printf("Vous avez tapé sur la touche '%c' \r\n", rx_byte);
        }
    }
}
```

Ce que fait cet exemple est assez évident. Nous testons continuellement le **drapeau RXNE de USART2**. Quand il est activé, nous transférons le contenu du registre **RDR** dans la variable locale **rx_byte**.


Compilez le programme, flashez la cible et tapez quelques touches de clavier dans les fenêtres de la console :



```
COM16 - PuTTY
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche ' '
Vous avez tapé sur la touche 'e'
Vous avez tapé sur la touche 'r'
Vous avez tapé sur la touche 'd'
Vous avez tapé sur la touche 's'
Vous avez tapé sur la touche 'd'
Vous avez tapé sur la touche 'd'
Vous avez tapé sur la touche 'd'
Vous avez tapé sur la touche 'd'
Vous avez tapé sur la touche 'd'
```

Maintenant, ouvrez une session de débogage et placez un point d'arrêt (double cliquez sur le bord de l'éditeur de texte, avant le numéro de la ligne) juste avant de lire le contenu du registre RDR :

```
31
32 // boucle principale des applications
33 while(1)
34 {
35     // is il y a quelque chose a lire dans le registre RDR de USART2
36     if ((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
37     {
38         // lire le caractère et le renvoyer sur le PC
39         rx_byte = USART2->RDR;
40         mon_printf("Vous avez tapé sur la touche '%c' \r\n", rx_byte);
41     }
42 }
43
44
45 }
46
```

Ensuite, lancez l'exécution  et utilisez le programme de terminal série pour envoyer quelque chose en frappant une touche du clavier. Le code devrait être suspendu au point d'arrêt. Vérifiez la valeur de **RXNE** à partir d'ici, elle devrait être à 1.

```
31
32 // boucle principale des applications
33 while(1)
34 {
35     // is il y a quelque chose a lire dans le registre RDR de USART2
36     if ((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
37     {
38         // lire le caractère et le renvoyer sur le PC
39         rx_byte = USART2->RDR;
40         mon_printf("Vous avez tapé sur la touche '%c' \r\n", rx_byte);
41     }
42 }
43
44
45 }
46
```

Ensuite, passez sur une ligne (c'est-à-dire lisez le contenu de **RDR**). Vous devriez voir que **RXNE** est effacé par cette action. En conclusion, la lecture du registre **RDR** réinitialise automatiquement le drapeau **RXNE**. Il n'est donc pas nécessaire de s'occuper de ce registre dans le programme d'application.

La fonction **main()** ci-dessus ne fait qu'interroger les données du registre RX de USART2. Nous avons déjà discuté des inconvénients de l'approche polling pour le bouton-poussoir :

- Dans l'exemple ci-dessus, le CPU est entièrement occupé à lire le drapeau RXNE.
- S'il y a d'autres tâches importantes à faire entre les lectures de RXNE, vous manquerez probablement un bon nombre d'octets entrants.

Utilisons l'interruption !

2. Activation des interruptions de réception de caractères USART – RX

2.1 Configuration de USART et NVIC

Modifiez la fonction **BSP_Console_Init()** pour activer l'interruption lors de l'événement RXNE :

```
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_AFR1 =
    GPIOA->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_AFR1 =
    GPIOA->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 transmission : 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);

// Action la demande interruption de evenement RXNE : caractere recu
USART2->CR1 |= USART_CR1_RXNEIE ;

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

```

Ensuite, activez l'interruption **USART2** pour qu'elle soit transmise au contrôleur NVIC avec la priorité 1 (ceci est fait dans la fonction **BSP_NVIC_Init()**) :

```
// Mettre la priorite 1 pour les interruptions de USART2
NVIC_SetPriority(USART2_IRQn, 1);
// Autoriser les demandes interruptions de USART2
NVIC_EnableIRQ(USART2_IRQn);
```

2.2 Gestionnaire d'interruption simple pour USART RX

Maintenant, écrivez un simple gestionnaire d'interruptions pour la réception des caractères sur UART2 RX :

```
/*
 * This function handles USART2 interrupts
 */

extern uint8_t console_rx_byte;
extern uint8_t console_rx_irq;

void USART2_IRQHandler()
{
    // Test pour verifier si la demande vient de RXNE
    if ((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
    {
        // Le drapeau RXNE est automatiquement remis a ZERO si lecture de RDR.
        // Lecture et sauvegarde du caractere
        console_rx_byte = USART2->RDR;
        // signalement de lecture : synchronisation avec le main
        console_rx_irq = 1;
    }
}
```

Notez que les variables **console_rx_byte** et **console_rx_irq** doivent être déclarées comme variables globales pour être disponibles à la fois depuis la fonction **main()** et depuis la fonction **USART2_IRQHandler()**.

2.3 Test de l'interruption RX de l'USART

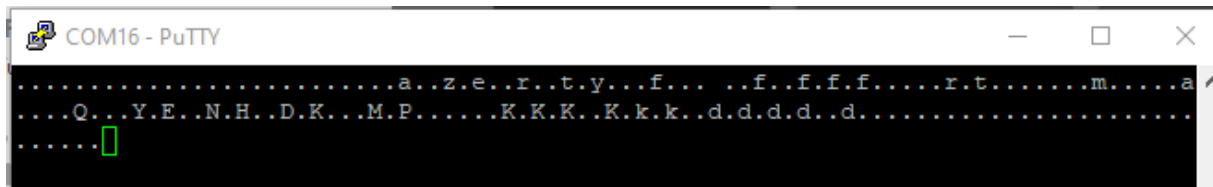
Testez la fonction main() suivante :

```
// variable globale
uint8_t button_irq = 0;
uint8_t timebase_irq = 0;

uint8_t console_rx_byte;
uint8_t console_rx_irq = 0;

int main(void)
{
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    // initialisation de Bouton bleu en interruption externe
    BSP_PB_IT_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    // Initialisation du Timer TIM6
    BSP_TIMER_Timebase_Init();
    // initialisation NVIC
    BSP_NVIC_Init();
    // boucle principale des applications
    while(1)
    {
        // Traitement du bouton en cas d'interruption
        if (button_irq == 1)
        {
            mon_printf("#");
            button_irq = 0;
        }
        // Traiter l'octet entrant de la console lors de l'interruption
        if (console_rx_irq == 1)
        {
            mon_printf("%c", console_rx_byte);
            console_rx_irq = 0;
        }
        // Traitement de la tache principale lors de l'interruption de la base de temps
        if (timebase_irq == 1)
        {
            mon_printf(".");
            timebase_irq = 0;
        }
    }
}
```

Compilez le projet et flashez la cible. Ouvrez la console et tapez quelques touches du clavier. Vous pouvez également cliquer sur le bouton utilisateur :



Cela semble fonctionner très bien. Aucune action de bouton ou de clavier n'est perdue, et la tâche est exécutée en toute sécurité toutes les 200 ms.

En fait, le risque de perdre les événements de la console existe toujours...

2.4 Mise en mémoire tampon des octets entrants.

Dans le code précédent, la tâche principale est lancée toutes les 200 ms, mais elle se termine assez rapidement. Que se passerait-il si plus d'une interruption **USART** se produisait au moment où la tâche principale s'exécute ?

Pour mettre le problème en lumière, ajustons un peu les timings...

Premièrement, changez le paramètre de la base de temps du Timer **TIM6** pour une période de **1s** entre les interruptions du Timer :

```
/*
 * BSP_TIMER_Timebase_Init()
 * TIM6 cadenser a 48MHz
 * Prescaler = 48000 -> periode de comptage = 1ms
 * Auto-reload = 1000 -> periode de debordement = 1000ms
 */

void BSP_TIMER_Timebase_Init()
{
    // activer horloge du peripherique TIM6
    // mettre a '1' le bit b4 (TIM6EN) du registre RCC_APB1ENR
    // voir page 131 du manuel de reference
    RCC->APB1ENR |= (1<<4); // le bit b4 est defini comme etant
    RCC_APB1ENR_TIM6EN

    // Faire un Reset de configuration du TIM6 : mise a zero des registres
    // TIM6_CR1 et TIM6_CR2
    // voir page 543 a 544 du manuel de reference
    TIM6->CR1 = 0x0000;
    TIM6->CR2 = 0x0000;

    // Configuration frequence de comptage
    // Prescaler : registre TIM6_PSC
    // Fck = 48MHz -> /48000 = 1KHz frequence de comptage
    TIM6->PSC = (uint16_t) 48000 -1;

    // Configuration periode des evenements
    // Prescaler : registre TIM6_ARR
    // 1000 /10 = 100Hz
    TIM6->ARR = (uint16_t) 1000 -1;
```

```

// Activation auto-reload preload : prechargement
// mettre a '1' le bit b7 du registre TIM6_CR1
TIM6->CR1 |= (1<<7);

// Activation de la demande interruption de debordement
// mettre a '1' le bit b0 du registre TIM6_DIER
// page 544
TIM6->DIER|= (1<<0);

// Demarrer le Timer TIM6
// Mettre a '1' le bit b0 du registre TIM6_CR1
TIM6->CR1 |= (1<<0);
}

```

Ensuite, il faut ajouter le temps nécessaire à l'exécution de la tâche principale. Nous pouvons utiliser notre fonction de délai simple, et définir le délai de sorte que la tâche principale s'étende sur la plus grande partie de la période de boucle de 1s (disons 900ms) :

Compilez et exécutez l'application. Vous devriez obtenir l'affichage '.' toutes les secondes sur le Terminal PuTTY.

Maintenant, essayez de frapper plusieurs touches rapidement l'une après l'autre. Voici le résultat de la frappe rapide de 'a', 'z', 'e', 'r', 't', 'y' :



Vous pouvez voir qu'il manque des caractères...

Eh bien, le processus d'interruption de l'**USART** garantit que tout octet arrivant dans le registre RDR de l'**USART2** est immédiatement transféré dans notre variable **console_rx_byte**. Mais qu'est-ce qui empêche **console_rx_byte** d'être rempli par un nouvel octet entrant avant qu'il ne soit envoyé au PC et affiché sur le Terminal ?

Hum... rien !

Pour pallier la perte d'octets, nous allons devoir implémenter un buffer d'entrée capable de stocker plus d'un octet entrant jusqu'à ce que la fonction **main()** trouve le temps de traiter tous les octets reçus.

Tout d'abord, transformons la variable **rx_byte** en un tableau de 10 octets :

```
uint8_t console_rx_byte [10];
```

Ensuite, nous pouvons utiliser la variable **console_rx_irq** comme index pour écrire dans le tableau **console_rx_byte** :

Par conséquent, le gestionnaire incrémente la variable **console_rx_irq** chaque fois qu'un nouvel octet arrive dans le buffer. La remise à zéro de **console_rx_irq** est laissée à la fonction **main()** lorsque les octets reçus ont été correctement traités (c'est-à-dire envoyés au PC et affichés).

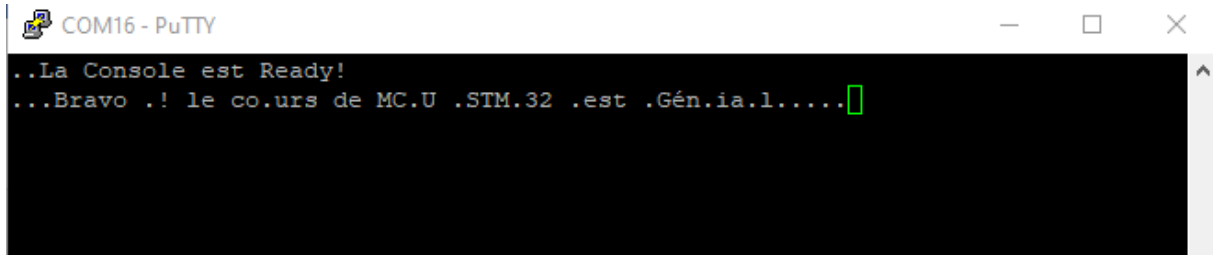
La fonction **main()** peut maintenant vider le buffer entre deux exécutions de la tâche principale :

```
// variable globale
uint8_t button_irq = 0;
uint8_t timebase_irq = 0;
uint8_t console_rx_byte [10];
uint8_t console_rx_irq = 0;

int main(void)
{
    uint8_t *prx_buffer;
    // Configuration horloge du systeme : 48MHz avec 8MHz HSE
    SystemClock_Config();
    // initialisation de Bouton bleu en interruption externe
    BSP_PB_IT_Init();
    // Initialisation USART pour console liaison serie
    BSP_Console_Init();
    // test console : message accueil
    mon_printf("La Console est Ready!\r\n");
    // Initialisation du Timer TIM6
    BSP_TIMER_Timebase_Init();
    // initialisation NVIC
    BSP_NVIC_Init();
    // boucle principale des applications
    while(1)
    {
        // Traitement du bouton en cas d'interruption
        if (button_irq == 1)
        {
            mon_printf("#");
            button_irq = 0;
        }
        //Placer le pointeur au début du buffer de UART RX.
        prx_buffer = console_rx_byte;
        // Traiter l'octet entrant de la console lors de l'interruption
        while (console_rx_irq > 0)
        {
            mon_printf("%c", *prx_buffer);
            prx_buffer++;
            console_rx_irq--;
        }
        //Traitement de la tache principale lors de l'interruption de la base de temps
        if (timebase_irq == 1)
        {
            mon_printf(".");
            timebase_irq = 0;
            BSP_DELAY_ms(900);
        }
    }
}
```

Si vous avez peur des pointeurs, vous pouvez utiliser un index de tableau de la même manière...

Compilez le projet et flashez la cible. Ouvrez la console et tapez quelques touches du clavier aussi vite que possible :



Aucun octet entrant n'est perdu !

Notez que la taille du buffer dépend de l'application. Ici, nous pouvons voir qu'un maximum de 5 octets (vous pouvez en avoir plus si vous êtes rapide) sont stockés dans le buffer avant qu'il ne soit imprimé. Il est prudent de surdimensionner légèrement le buffer.

3. Résumé

Dans ce tutoriel, vous avez appris à recevoir des octets du périphérique **USART** en utilisant les méthodes de polling et d'interruption.

Notez que dans cet exemple, le débit des données entrantes est très lent car il dépend de la frappe humaine sur un clavier. Dans de nombreuses applications, vous recevrez des données série d'un autre circuit intégré (un capteur par exemple) avec un flux continu et élevé d'octets entrants. Une mise en mémoire tampon correcte de ces octets entrants est donc cruciale.

Si la ligne **RX** est très occupée, travailler avec des interruptions peut encore ne pas être une bonne idée. Parce que les interruptions se produiront très souvent, les tâches importantes seront perturbées très souvent (même pour de courtes ISR).

C'est là que le contrôleur **DMA** entre en jeu. Rendez-vous au prochain tutoriel !