STM32 Microcontrollers

Tutorial #3.4 : USART RX Interrupt	Rev : 0
Author(s) : Laurent Latorre	Date : 17-10-2015

Contents

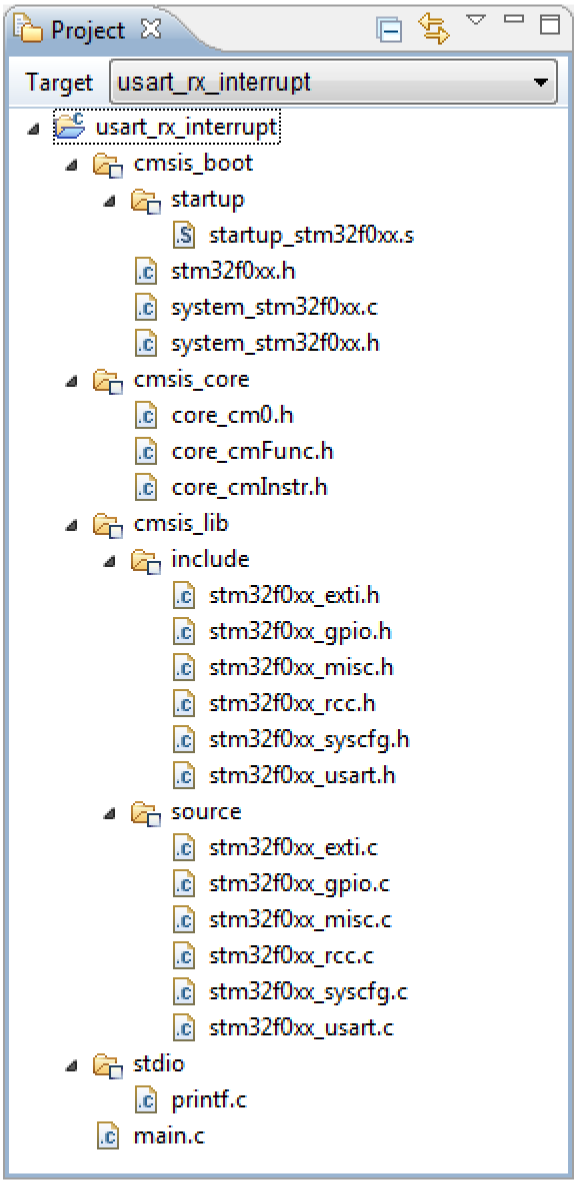
1. Start a new project

Start a new project called “**usart_rx_interrupt**” using the project (“**external_interrupts**”) as a base.

You need to have the following setuo:

- Low-level libraries (**cmsis**)
- **Startup file**s and correct clock configuration
- **RCC** driver
- **GPIO** driver with previously written init functions
  - **LED_Init()**
  - **PB_Init()**
  - **LED_Toggle()**
- **USART** driver with previously written init function
  - **USART2_Init()**
- **printf** redirection with a working **my_printf()** function to the host computer console (Putty)
- **MISC** drivers
- **EXTI** drivers
- **SYSCFG** drivers

Once you have done that, the project file structure should look as the one beside.



Target usart_rx_interrupt

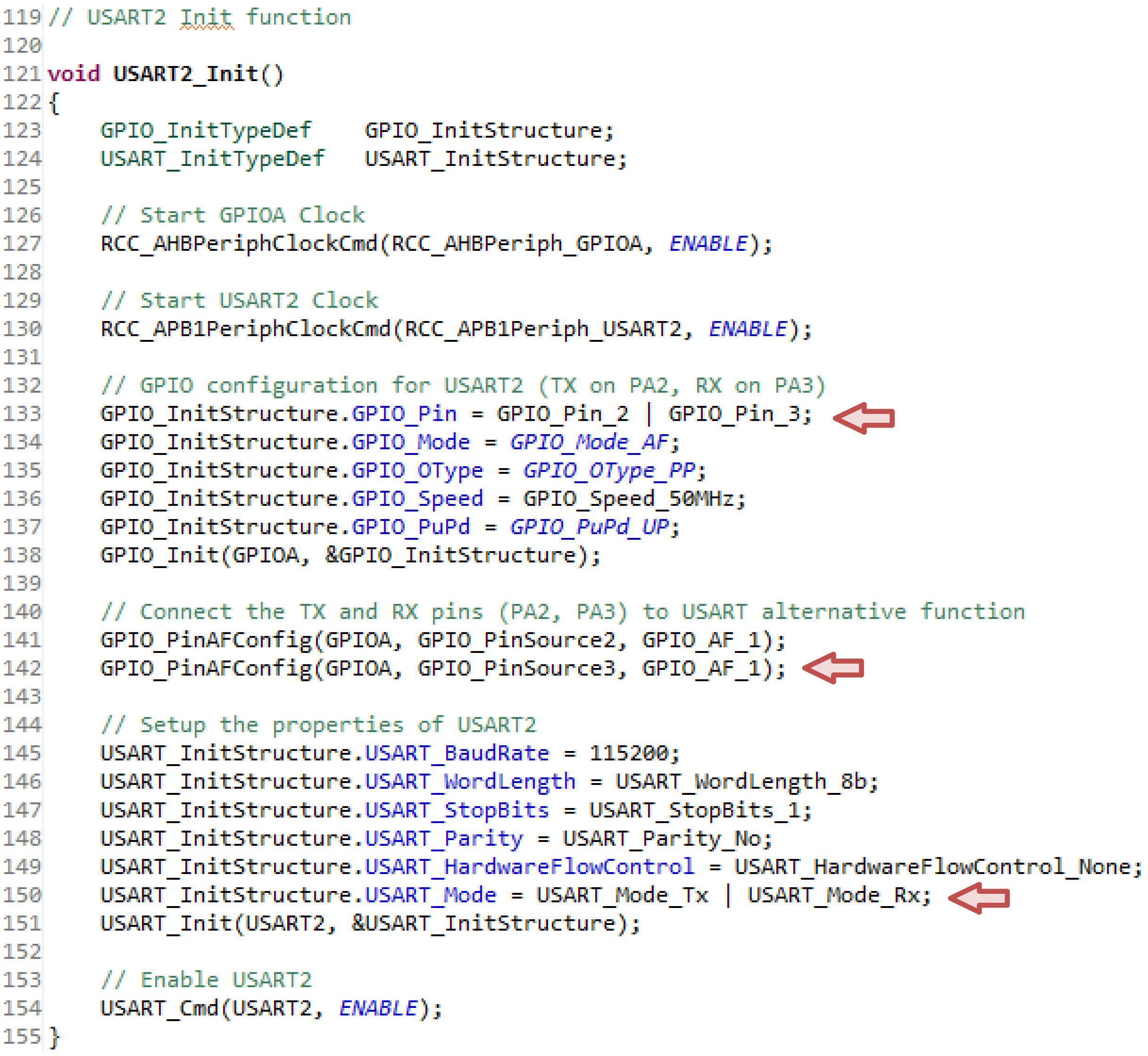
```
usart_rx_interrupt
    cmsis_boot
        startup
            startup_stm32f0xx.s
        stm32f0xx.h
        system_stm32f0xx.c
        system_stm32f0xx.h
    cmsis_core
        core_cm0.h
        core_cmFunc.h
        core_cmInstr.h
    cmsis_lib
        include
            stm32f0xx_exti.h
            stm32f0xx_gpio.h
            stm32f0xx_misc.h
            stm32f0xx_rcc.h
            stm32f0xx_syscfg.h
            stm32f0xx_usart.h
        source
            stm32f0xx_exti.c
            stm32f0xx_gpio.c
            stm32f0xx_misc.c
            stm32f0xx_rcc.c
            stm32f0xx_syscfg.c
            stm32f0xx_usart.c
    stdio
        printf.c
    main.c
```

2. Using the USART in TX/RX mode

2.1. Setting up the peripheral

So far, we have been using **USART2** in **TX** mode only, to output console messages with **my_printf()** function. In this tutorial, we want to be able to receive incoming data from the terminal window as well.

Edit the **USART2_Init()** function this way:

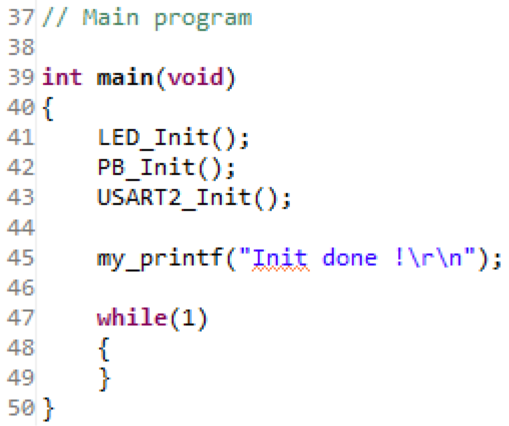


```
// USART2 Init function

void USART2_Init()
{
    GPIO_InitTypeDef    GPIO_InitStructure;
    USART_InitTypeDef   USART_InitStructure;

    // Start GPIOA Clock
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);

    // Start USART2 Clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

    // GPIO configuration for USART2 (TX on PA2, RX on PA3)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // Connect the TX and RX pins (PA2, PA3) to USART alternative function
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource2, GPIO_AF_1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource3, GPIO_AF_1);

    // Setup the properties of USART2
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
    USART_Init(USART2, &USART_InitStructure);

    // Enable USART2
    USART_Cmd(USART2, ENABLE);
}
```

Doing that, we have just connected the **PA3** pin to the **USART2 RX** function (leaving **PA2** as **TX**) and configured **USART2** to work in both **TX/RX** modes.

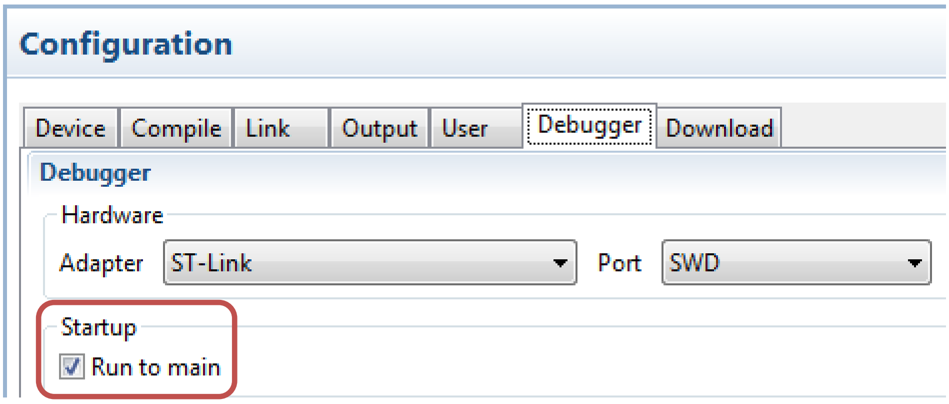
2.2. Understanding USART in RX mode

Make the **main()** function as simple as this:



```
// Main program

int main(void)
{
    LED_Init();
    PB_Init();
    USART2_Init();

    my_printf("Init done !\r\n");

    while(1)
    {
    }
}
```

In the **Configuration** tab, make sure that **Debugger** setting « **Run to main** » is checked.



Debugger

Hardware

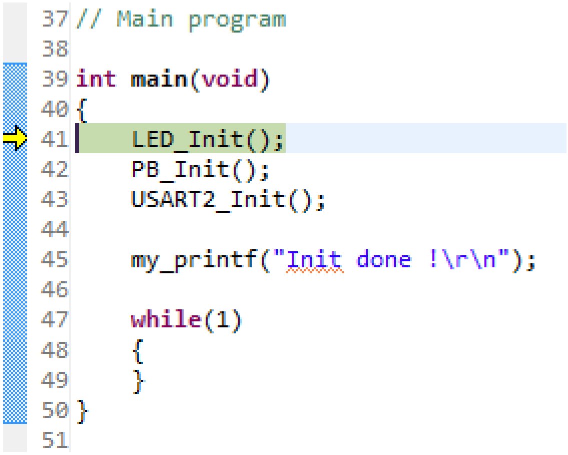
Adapter ST-Link Port SWD

Startup

☑ Run to main

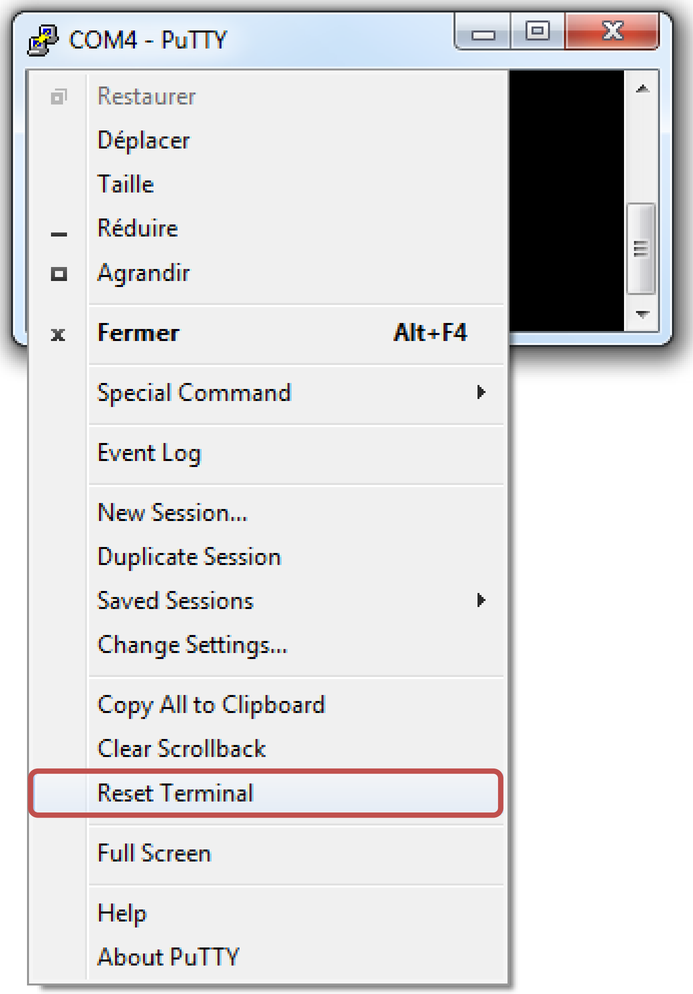
Build the program (it should build with no error or warning), start the debugger and open the terminal window (Putty) with the correct COM port settings.

Once started, the debugger will stop at the beginning of the **main()** function, waiting for you to command execution.



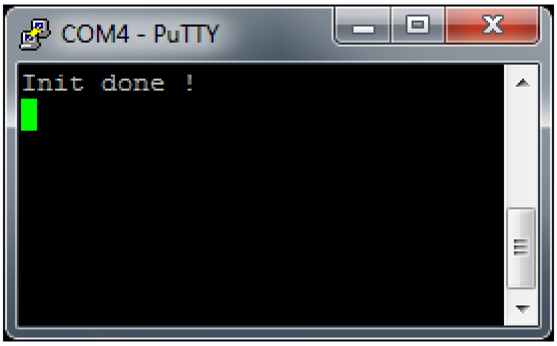
```
// Main program

int main(void)
{
    LED_Init();
    PB_Init();
    USART2_Init();

    my_printf("Init done !\r\n");

    while(1)
    {
    }
}

```

Clean the Putty terminal from previous messages by clicking on the window icon and then “Reset teminal”.

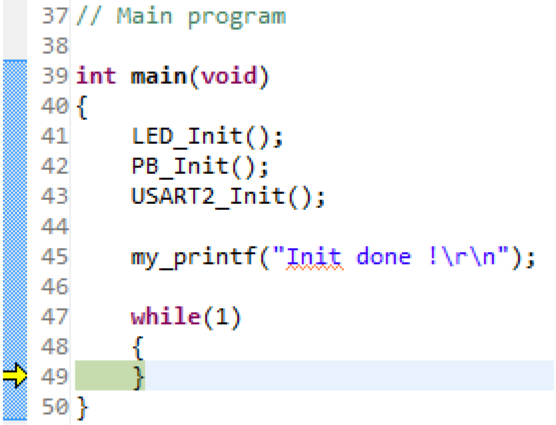


Now, start the program execution ▶ for few seconds, and then press the suspend ⏸ button.

You should see the ”**Init done !**” message in the console, and the program is trapped in your ***while(1)*** never ending loop:

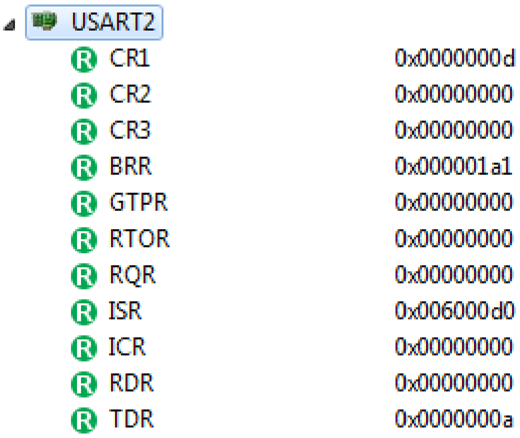


```
Init done !
```



```
// Main program

int main(void)
{
    LED_Init();
    PB_Init();
    USART2_Init();

    my_printf("Init done !\r\n");

    while(1)
    {
    }
}
```

In the debugger view, open the **Peripherals** tab if not already opened (**View →Peripherals**). Then unfold the **USART2** registers:



```
USART2
   CR1     0x0000000d
   CR2     0x00000000
   CR3     0x00000000
   BRR     0x000001a1
   GTPR    0x00000000
   RTOR    0x00000000
   RQR     0x00000000
   ISR     0x006000d0
   ICR     0x00000000
   RDR     0x00000000
   TDR     0x0000000a
```

If register contents appear in red, just start/suspend the program execution few seconds again. The red color is useful to see what registers have changed between two successive suspended states.

Take a moment to open each of the **USART2** registers and have a look on the various settings and available information you can gather from these registers. For instance:

- **CR1**, **CR2**, **CR3** are used to configure the behavior of **USART2**
- **ISR** reports statut of **USART2**
- **RDR** is the receive data register
- **TDR** is the transmit data register

Looking into **ISR** (**Interrupt & Status Register,** do not confuse with ***Interrupt Service Routine***) tells us that :

- **USART2** is currently not busy (**BUSY = 0**)
- The receive data register is empty (**RXNE=0**)

You can also notice that **TDR** is loaded with **0x0A** byte, which corresponds to the very last byte we sent in the “**Init done !\r\n**” message (i.e. the ASCII code of ‘**\n**’).

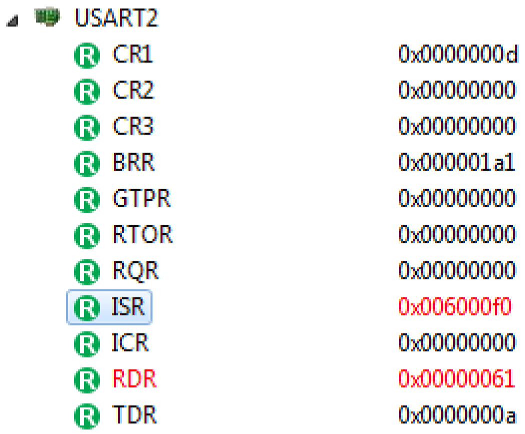
Now, do the following <u>with caution</u>:

1. Press the start button ▶ in the debugger
2. Click on the terminal (Putty) window to bring it into focus
3. Press the ‘**a**’ key on the computer keyboard (only once). You will see nothing happening in the terminal window, this is normal.
4. Now, press the suspend ⏸ button in the debugger

Doing this, you have asked the teminal programm (Putty) to send the byte ‘**a**’ to the STM32 on its **USART2 RX** pin.

If you have done the above steps well, you should see some changes in the USART2 peripheral registers:

- **ISR** has changed
- **RDR** now contains the byte **0x61**, which is the ASCII code of the ‘**a**’ character (the one you pressed)



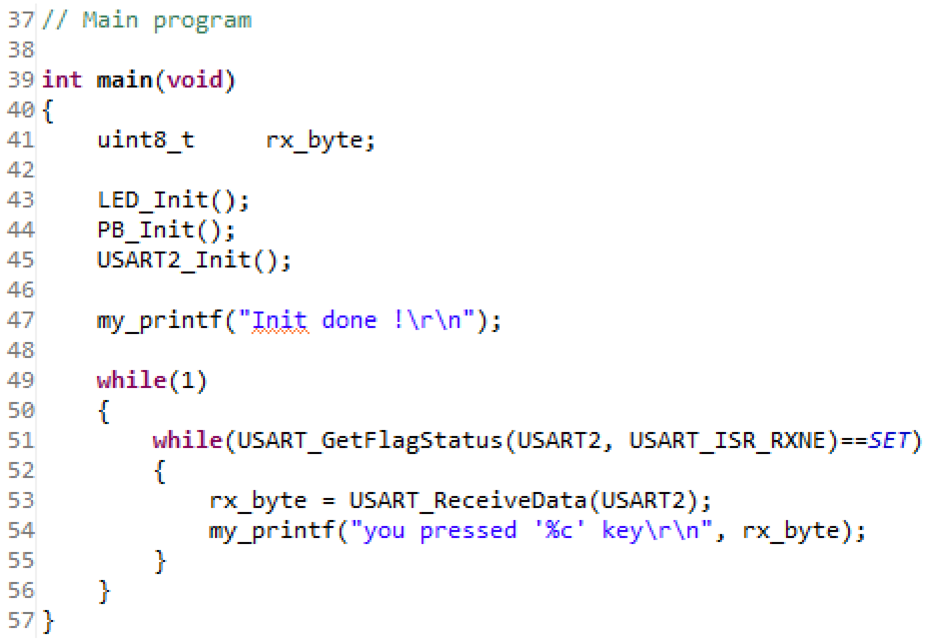
```
USART2
    CR1      0x0000000d
    CR2      0x00000000
    CR3      0x00000000
    BRR      0x000001a1
    GTPR     0x00000000
    RTOR     0x00000000
    RQR      0x00000000
    ISR      0x006000f0
    ICR      0x00000000
    RDR      0x00000061
    TDR      0x0000000a
```

Open the detail view of **ISR** and observe that now **RXNE** bit is set to ‘**1**’, which tells us that a data is available in the **RDR** register.

Repeat the above steps, but instead of hitting the key ‘**a**’ at step 3, hit the key ‘**b**’ (only once). You will see that **0X62** byte has now arrived in the **RDR** register. You can continue repeating the above steps with differents keyboard keys, and verify that the last pressed key has always arrived in the **RDR** register.

That means that the first ‘**a**’ character that we pressed is now lost. **USART** peripheral is therefore not storing successive incoming bytes. It is your responsibility to read incomming bytes when available, and before a new one arrive and take place in the **RDR** register.

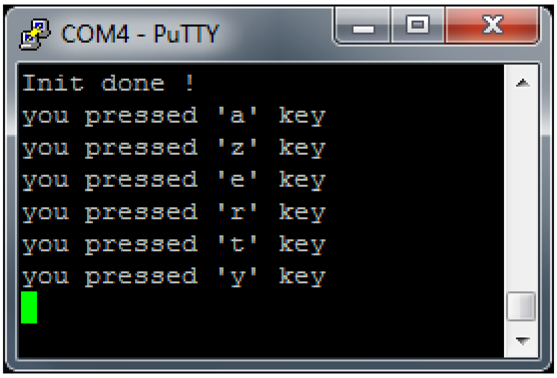
Quit the debugger and edit the main() function:



```
37 // Main program
38
39 int main(void)
40 {
41     uint8_t      rx_byte;
42
43     LED_Init();
44     PB_Init();
45     USART2_Init();
46
47     my_printf("Init done !\r\n");
48
49     while(1)
50     {
51         while(USART_GetFlagStatus(USART2, USART_ISR_RXNE)==SET)
52         {
53             rx_byte = USART_ReceiveData(USART2);
54             my_printf("you pressed '%c' key\r\n", rx_byte);
55         }
56     }
57 }
```

What this example does is pretty obvious. We continuously test the USART2 RXNE flag. When it is set, we transfer the **RDR** register content into the **rx_byte** local variable.

Build the program, flash the target and hit some keyboard keys in the console windows:



```
Init done !
you pressed 'a' key
you pressed 'z' key
you pressed 'e' key
you pressed 'r' key
you pressed 't' key
you pressed 'y' key
```

You should open the debugger, set a breakpoint on line **53** and look at the **RXNE** flag when you step over (F10) the **USART_ReceiveData()** function. You will see that reading the **RDR** register automatically resets the **RXNE** flag. Therefore, there is no need to take care of this register from the application program.

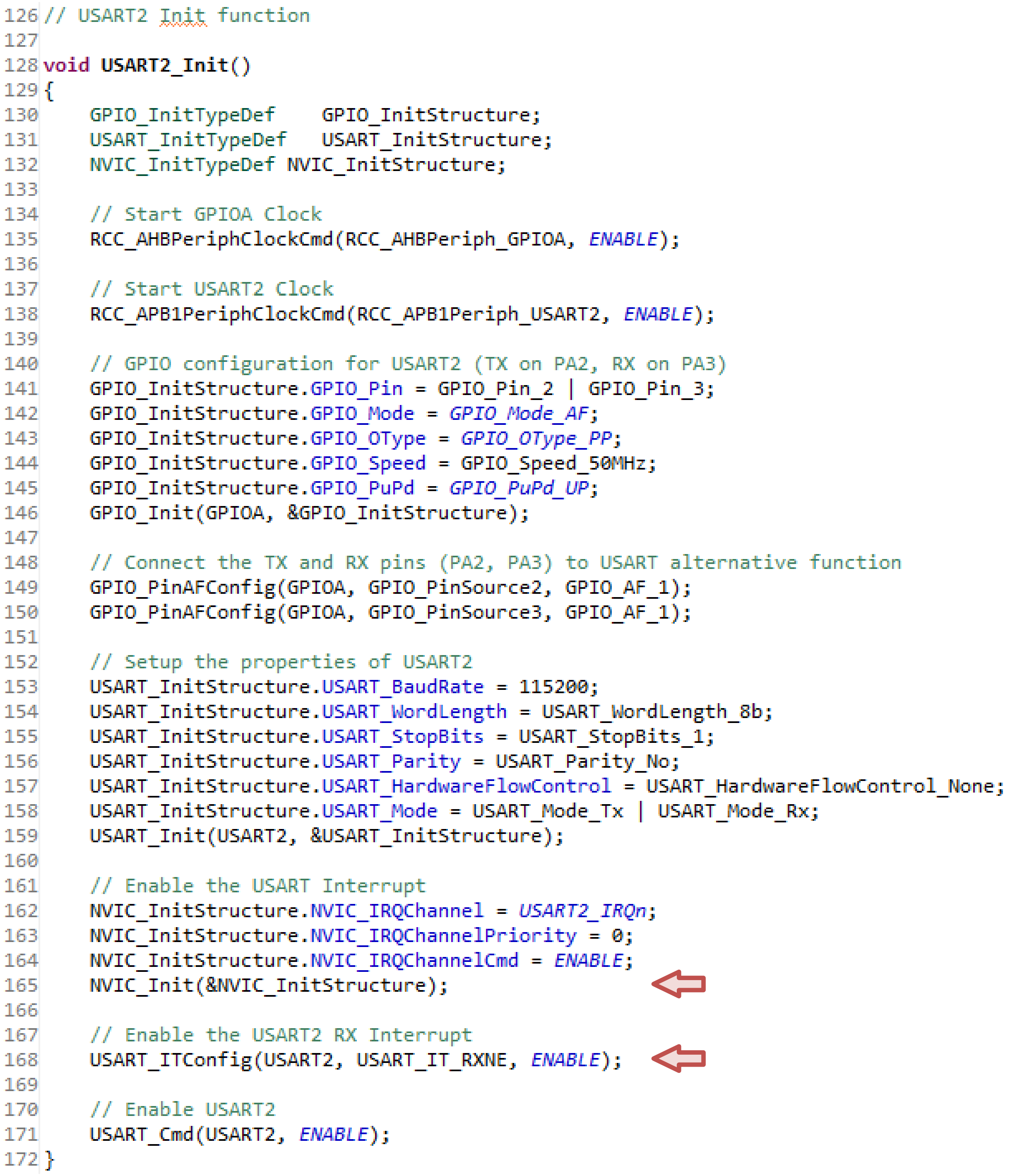
The above **main()** function is just polling data on the **USART2 RX** register. We already discussed the drawbacks of the polling approach for the push-button:

- In the above example, the CPU is fully busy reading the **RXNE** flag
- If there are some other important tasks to do between **RXNE** readings, you will likely miss quite a few incoming bytes.

Here comes the interruption approach !

2.3. Setting USART RX to generate interrupt

Edit the **USART2_Init()** function to enable the interrupt upon **RX** event:



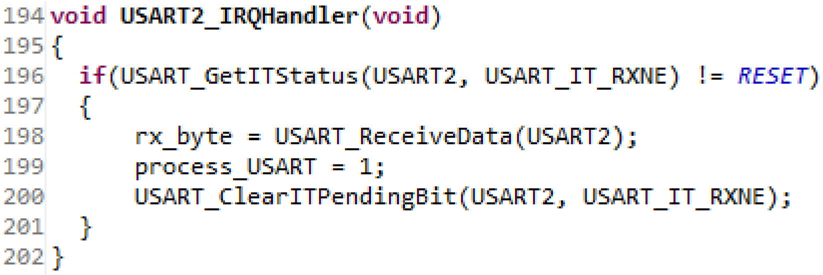
```
// USART2 Init function

void USART2_Init()
{
    GPIO_InitTypeDef    GPIO_InitStructure;
    USART_InitTypeDef   USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    // Start GPIOA Clock
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);

    // Start USART2 Clock
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

    // GPIO configuration for USART2 (TX on PA2, RX on PA3)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // Connect the TX and RX pins (PA2, PA3) to USART alternative function
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource2, GPIO_AF_1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource3, GPIO_AF_1);

    // Setup the properties of USART2
    USART_InitStructure.USART_BaudRate = 115200;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
    USART_Init(USART2, &USART_InitStructure);

    // Enable the USART Interrupt
    NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    // Enable the USART2 RX Interrupt
    USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);

    // Enable USART2
    USART_Cmd(USART2, ENABLE);
}
```

As usual now, we have:

- To tell the NVIC controller that USART2 interrupt signals are allowed, with a priority level to be set as you want.
- To tell the USART2 peripheral to generate an interrupt signal upon change of the RXNE event flag.

2.4. Basic USART RX interrupt handler

Now, write a simple interrupt handler:

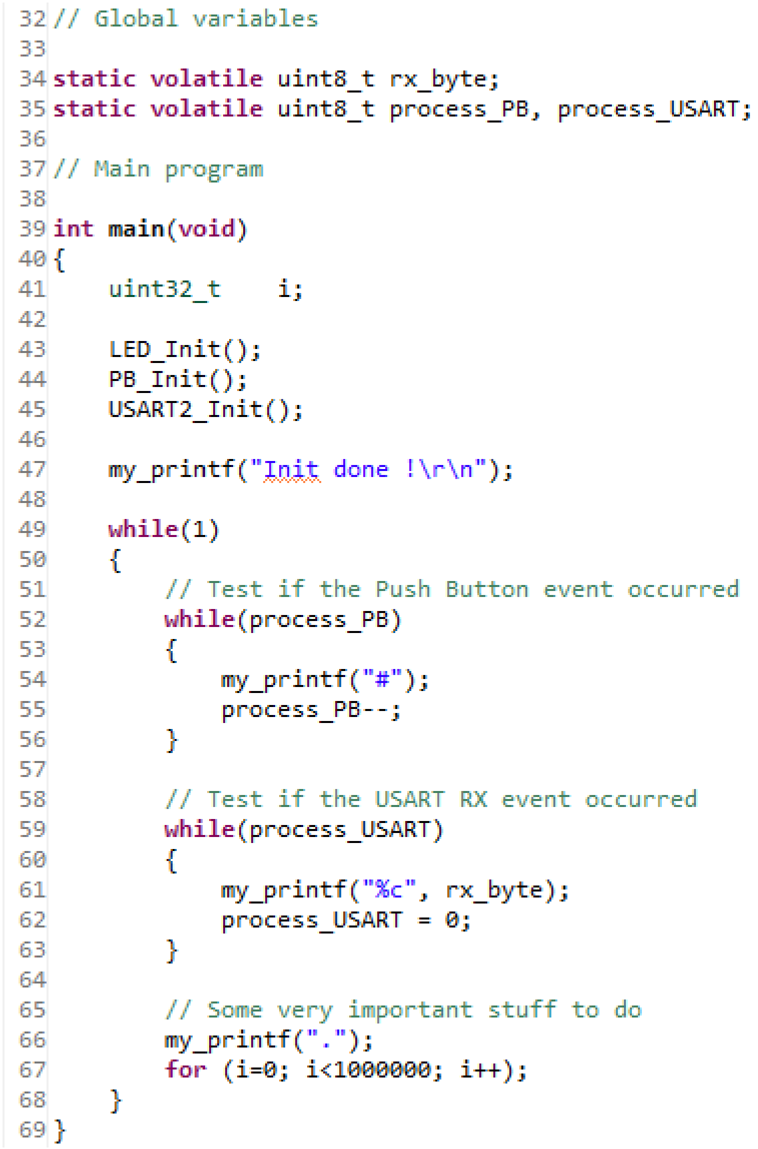


```
void USART2_IRQHandler(void)
{
  if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
  {
      rx_byte = USART_ReceiveData(USART2);
      process_USART = 1;
      USART_ClearITPendingBit(USART2, USART_IT_RXNE);
  }
}
```

Note that both **rx_byte** and **process_USART** variables should be declared as global variables to be seen both from the **main()** function and from the **USART2_IRQHandler()** function.

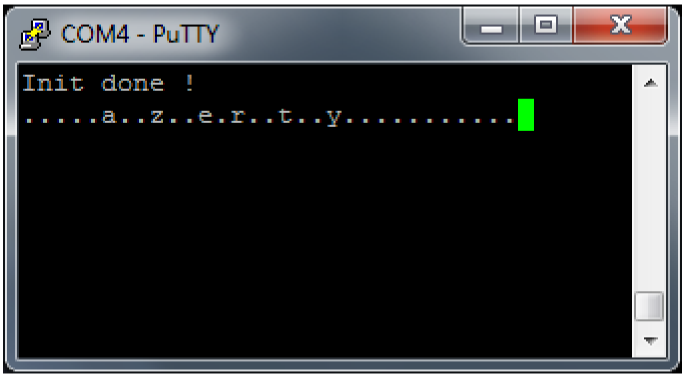
2.5. Testing the USART RX interrupt

Try the following **main()** function:



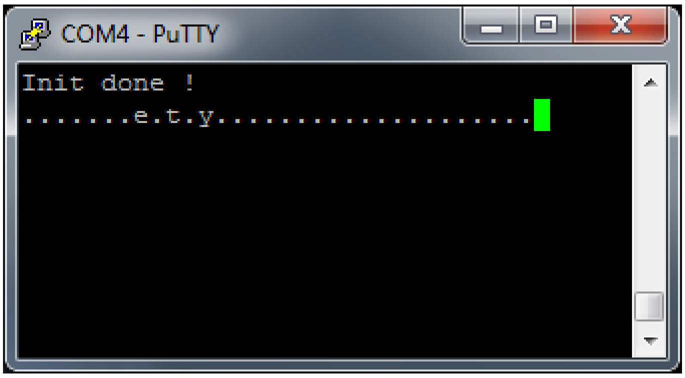
```c
// Global variables

static volatile uint8_t rx_byte;
static volatile uint8_t process_PB, process_USART;

// Main program

int main(void)
{
    uint32_t    i;

    LED_Init();
    PB_Init();
    USART2_Init();

    my_printf("Init done !\r\n");

    while(1)
    {
        // Test if the Push Button event occurred
        while(process_PB)
        {
            my_printf("#");
            process_PB--;
        }

        // Test if the USART RX event occurred
        while(process_USART)
        {
            my_printf("%c", rx_byte);
            process_USART = 0;
        }

        // Some very important stuff to do
        my_printf(".");
        for (i=0; i<1000000; i++);
    }
}
```

Build the project and flash the target. Open the console and hit some keyboard keys:



```
Init done !
.....a..z..e.r..t..y..........
```

Now try hitting several keys quickly one after each other. Below is a result of hitting '**a**', '**z**', '**e**', '**r**', '**t**', '**y**' quite fast:



```
Init done !
.......e.t.y....................
```

You can see that ‘**a**’, ‘**z**’ and ‘**r**’ are missing…

Well, the interrupt process guaranties that any byte arriving in the **USART2 RDR** register is immediately transferred into our variable **rx_byte**. But what is preventing **rx_byte** to be overwritten by a new incomming byte before it is printed out?

Hum… nothing!

3. Buffering incomming bytes

To overcome the loss of bytes, we have to implement an input buffer that can store more than one incomming byte until the **main()** function finds time to process all the received bytes.

First, let us change the **rx_byte** variable into an array of bytes:

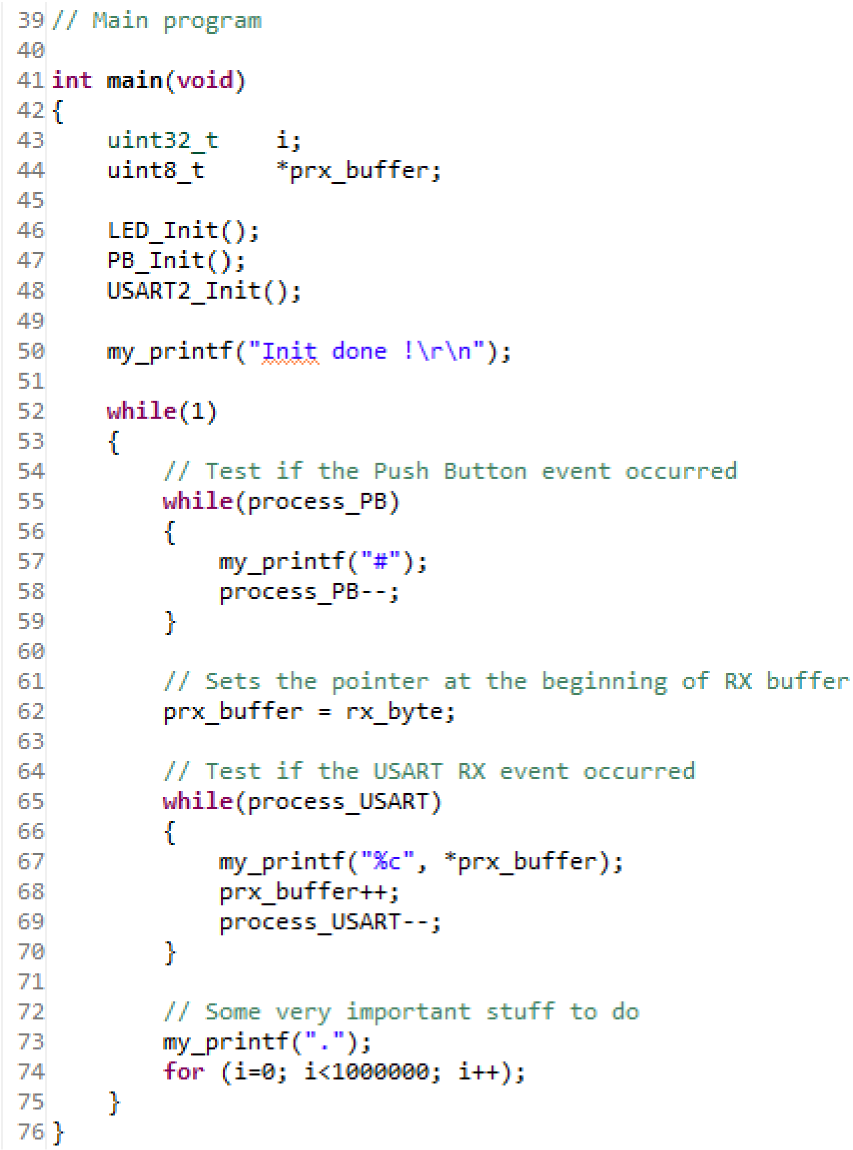
```c
// Global variables

#define RX_BUFFER_SIZE  10

static uint8_t  rx_byte[RX_BUFFER_SIZE];
static uint8_t  process_PB, process_USART;

```

Next, we can use the **process_USART** variable as an index for writing in the **rx_byte** array:

```c
void USART2_IRQHandler(void)
{
  if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
  {
      rx_byte[process_USART] = USART_ReceiveData(USART2);
      process_USART++;
      USART_ClearITPendingBit(USART2, USART_IT_RXNE);
  }
}
```

As a result, the handler increments the **process_USART** variable each time a new byte fills the buffer. Ressetting **process_USART** to zero is left to the **main()** function when received bytes have been corecctly processed (i.e. printed out in our case).

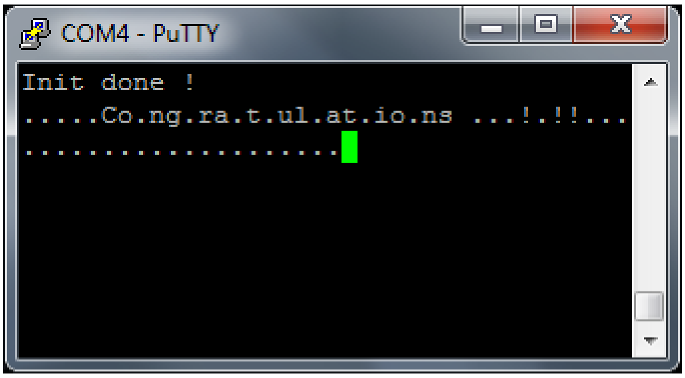
The **main()** function can now empty the buffer only between important tasks:



```c
// Main program

int main(void)
{
    uint32_t    i;
    uint8_t     *prx_buffer;

    LED_Init();
    PB_Init();
    USART2_Init();

    my_printf("Init done !\r\n");

    while(1)
    {
        // Test if the Push Button event occurred
        while(process_PB)
        {
            my_printf("#");
            process_PB--;
        }

        // Sets the pointer at the beginning of RX buffer
        prx_buffer = rx_byte;

        // Test if the USART RX event occurred
        while(process_USART)
        {
            my_printf("%c", *prx_buffer);
            prx_buffer++;
            process_USART--;
        }

        // Some very important stuff to do
        my_printf(".");
        for (i=0; i<1000000; i++);
    }
}
```

If you don't like pointers, you may use an index variable the same way…

Build the project and flash the target. Open the console and hit some keyboard keys as fast as you can:



```
Init done !
.....Co.ng.ra.t.ul.at.io.ns ...!.!!...
.................
```

No incoming byte is lost!

Note that a smaller buffer size would have been enough for this particular application, as we can see that a maximum of 2 bytes (you can get 3 if you are fast) are stored in the buffer before it is printed out. It is recommended to slightly oversize the buffer for safety reasons.

4. Summary

In this tutorial, you have learned how to receive bytes from **USART** peripheral using both polling and interrupt based methods.

Note that in this example, incoming data rate is very slow because it relies on human hitting keys on a keyboard. In many applications, you will get serial data from another IC (a sensor for instance) with a continuous high rate flow of incoming bytes. Correctly buffering these incoming bytes is therefore crucial.

If **RX** line is very busy, working with interruptions might still not be a good idea. Because interrupts will occur very often, the important tasks will be disturbed very often (even for short **ISR**).

That’s where **DMA** controller comes into the game, but that’s for next tutorial…