BAME 2123 Microcontrollers Peripherals
Practical Report 2

Aw Yu Ning RMI2

Objectives:

The purpose of this practical is to know how to program the USART device by using STM32F429ZIT6 MCU. The USART device is to communicate with PCs or any host devices, in this case here is PC, by transmits to or receives data from via USD-to-serial converter module. Few exercises will be carried out to show how the USART device works with PC.

Setup Devices and Software:

1. STM32F429ZIT6 MCU
2. CH360 Windows device driver
3. STM32CubeMX
4. TeraTerm

Procedure:

The configurations of the UART5 are as below:

1. 115,200bps
2. Odd parity
3. 9-bit data
4. Oversampling by 16
5. 1 stop bit
6. Configure APB1 to run at 45MHz

Noted that all these configurations will not be using the functions that already created inside the STM32CubeMX.All the functions and library is going to be manual created. The code is shown in the later section. By doing this, we manage to access, configure, read or write all the peripherals in this STM32F429ZIT6 MCU with ease and without studying the ways to use the system created functions.

USART Data Transmission:

When we are using USART to transfer data, there are some flags and registers we need to take care of. Such as:

- Data Register (DR): Contains the received and going-to-be transmitted data. It depends on the jobs given to the protocol, either read from or written to. Now, there are two *shadow registers* here which we cannot access it, which are Transmit data register (TDR) and Receive data register (RDR).
    - Transmit data register (TDR) – Before we transmit the data, the data will be written into DR, but actually the data is written into this register. After the data in shift register is sent completely, then only the data in TDR will be transferred into the shift register and send out to serial port. (In the CubeMX debug tool, we cannot see/read the transmitting data in the DR but only the received data in DR.)
    - Receive data register (RDR) – In the reading operation, same as TDR, the received data is loaded into RDR register first then DR. The received data will only be transferred in to shift register when it is empty.

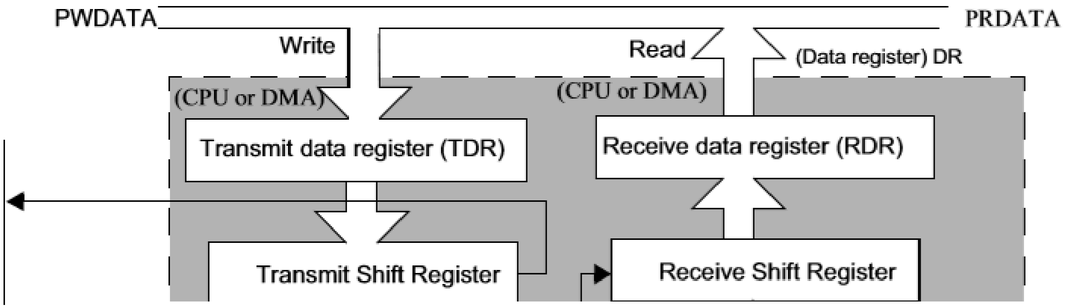


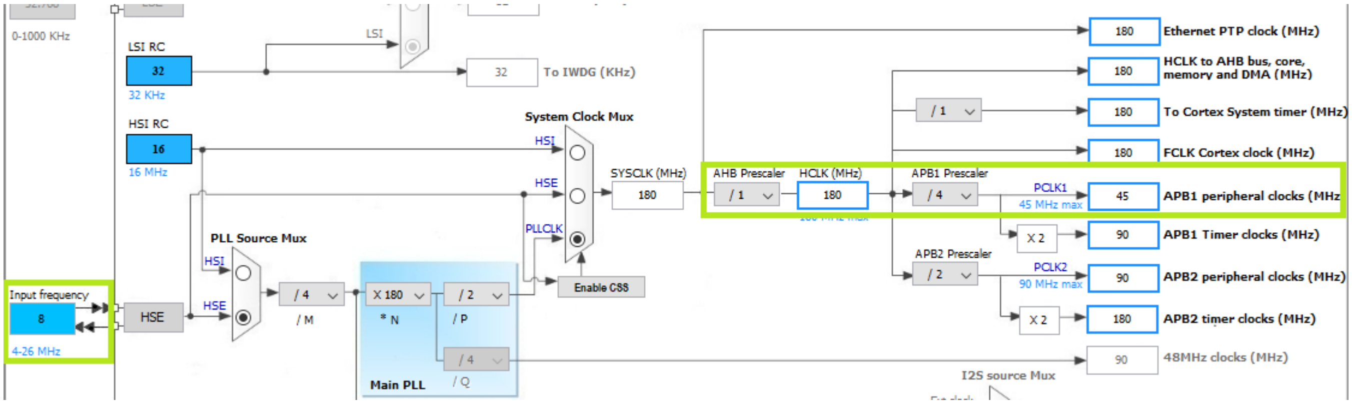
Figure 1: The register diagram of TDR and RDR in DR in STM32F429ZIT6 MCU, reference manual pg968

- ORE flags (Overrun Error): Is located in the status register, bit 3. This flag indicates an overrun error happened during the transmission of data. Transmit Shift register will not fetch data from Transmit data register (TDR) so the transferred data will be waiting in the TDR, example '3A'. Now, while the '3A' is waiting in TDR, the next transferred data, example '2D' is written into the TDR, and the shift register hasn't complete sending data, '3A' will be overwrite by '2D' and this is not allowed, therefore ORE flags is set.

- TC flags (Transfer complete flags): Located in status register, bit 6. This flags is always watching transmit shift register. When the shift register is empty, means data is transferred completely, this flags is set. It is used to check whether current data byte is transferred completely.

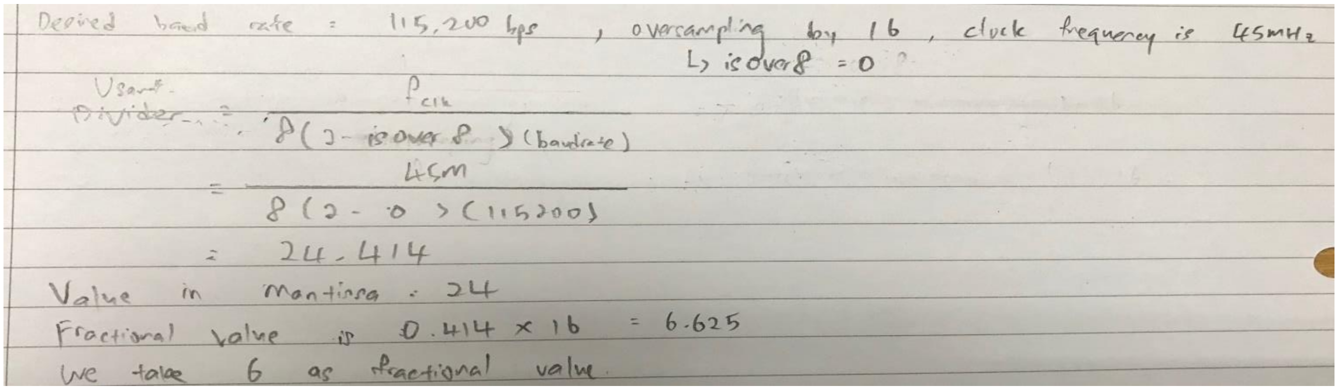
- TxE flags (Transfer not empty flags): Located in status register, bit 7. It indicates that data is transferred into transmit shift register and ready to accept the next data byte. It is usually used to check whether we are allowed to put in the next or new transferred data byte.

Clock Configurations:



To set APB1 peripheral clock to 45MHz, the AHB clock frequency is set to the max, 180MHz frequency. And in order to configure that frequency, the input clock is configured to 8MHz.

Baud Rate Computations:



Desired baud rate = 115,200 bps, oversampling by 16, clock frequency is 45MHz

↳ isOver8 = 0

$$\text{Usart Divider} = \frac{f_{clk}}{8(2 - isOver8)(baudrate)}$$

$$= \frac{45M}{8(2-0)(115200)}$$

$$= 24.414$$

Value in Mantissa = 24

Fractional value is 0.414 × 16 = 6.625

We take 6 as fractional value.

**isOver8 is a variable*

In this case here is over sampling by16, so isOver8 =0, and fractional value is multiply by 16. If it is oversampling by 8, then isOver8 = 1, and fractional value is multiply by 8. Note that Mantissa value **must not** exceed value of 2^{12} because the mantissa register is only 12 bits. Same does to fractional value and the register is 4 bits, therefore calculated fractional value **must not** have value larger than 16 in decimals.

Notice that we take the integer part of the fractional value, in this case, we take 6, and ignore 0.625. This makes the actual baud rate slightly different from desired baud rate. We will discuss about the problem later.

C codes USART configuration:

Header/Macros: The following code is the macros in *Uart.h* header file for source code.

```
typedef volatile uint32_t IOregister;

typedef struct UartRegs UartRegs;
struct UartRegs{
       IOregister   Sr;
       IOregister   Dr;
       IOregister   Brr;
       IOregister   Cr1;
       IOregister   Cr2;
       IOregister   Cr3;
       IOregister   Gtpr;
};

typedef enum {
       USART_OVER_16 = 0,
       USART_OVER_8
}OversamplingMode;

/*     To used in configure function
 *     Used bit in CR1,2,3 is only 16 bits
 *     OR them together so that we can immediately configure all register by just one
function
 *
 */
//Status Register
#define USART_PE                          1<<0         //Parity error
#define USART_FE                          1<<1         //Framing error
#define USART_NF                          1<<2         //Noise Detected flag
#define USART_ORE                         1<<3         //Overun Error
```

```
#define USART_IDLE                     1<<4         //IDLE line detected
#define USART_RXNE                     1<<5         //Read data register not empty
#define USART_TC                       1<<6         //Transmission complete
#define USART_TXE                      1<<7         //Transmit data register empty
#define USART_LBD                      1<<8         //LIN break detection flag
#define USART_CTS                      1<<9         //Clear to send flag

//control register 1
#define USART_SBK                      1LL<<0       //send break
#define USART_RWU                      1LL<<1       //receiver mode
#define USART_RE                       1LL<<2       //receiver enable
#define USART_TE                       1LL<<3       //transmitter enable
#define USART_IDLEIE                   1LL<<4       //IDLE interrupt enable
#define USART_RXNEIE                   1LL<<5       //Receive not empty interrupt enable
#define USART_TCIE                     1LL<<6       //transmission complete interrupt
enable
#define USART_TXEIE                    1LL<<7       //transmit enable interrupt enable
#define USART_PEIE                     1LL<<8       //peripheral enable
#define USART_PS_ODD                   1LL<<9       //parity selection
#define USART_PCE                      1LL<<10      //parity control enable
#define USART_WAKE                     1LL<<11      //wake up method
#define USART_M_9BIT                   1LL<<12      //word length
#define USART_UE                       1LL<<13      //usart enable
#define USART_OVER8                    1LL<<15      //oversampling mode 8
#define USART_OVER16                   0LL<<15      //oversampling mode 16

//control register 2
#define USART_LBDL                     1LL<<21      //LIN break detection length
#define USART_LBDLIE                   1LL<<22      //LIN break detection interrupt enable
#define USART_LBCL                     1LL<<24      //Last bit clock pulse
#define USART_CPHA                     1LL<<25      //clock phase
#define USART_CPOL                     1LL<<26      //clock parity
#define USART_CLKEN                    1LL<<27      //clock enable
#define USART_STOPBIT_1                0LL<<28      //stop bits mode
#define USART_STOPBIT_0DOT5            1LL<<28
#define USART_STOPBIT_2                2LL<<28
#define USART_STOPBIT_1DOT5            3LL<<28
#define USART_LINEN                    1LL<<30      //LIN mode enable

//control register 3
#define USART_EIE                      (1LL<<32)    //Error interrupt enable
#define USART_IREN                     (1LL<<33)    //IrDa mode enable
#define USART_IRLP                     (1LL<<34)    //IrDA low power
#define USART_HDSEL                    (1LL<<35)    //Half duplex selection
#define USART_NACK                     (1LL<<36)    //smartcard NACK enable
#define USART_SCEN                     (1LL<<37)    //smartcard mode enable
#define USART_DMAR                     (1LL<<38)    //DMA enable receiver
#define USART_DMAT                     (1LL<<39)    //DMA enable ransmitter
#define USART_RTSE                     (1LL<<40)    //RTS enable
#define USART_CTSE                     (1LL<<41)    //CTS enbale
#define USART_CTSIE                    (1LL<<42)    //CTS interrupt enable
#define USART_ONEBIT                   (1LL<<43)    //one sample bit method enable

#define SET_OVERSAMPLING_MODE(uart,whichMode) (uart)->Cr1 |= whichMode
#define IS_OVERSAMPLING_16(mode)                    ~(mode &= 0x80)

#define USARTIsStatus(usart,whichFlags)             (usart)->Sr & whichFlags
#define usartIsTXRegEmpty(usart)                    USARTIsStatus(usart,USART_TXE)
```

```
#define usartIsRXNERegNotEmpty(usart)              USARTIsStatus(usart,USART_RXNE)

#define uart4 (UartRegs *)0x40004C00
#define uart5 (UartRegs *)0x40005000
#define uart7 (UartRegs *)0x40007800
#define uart7 (UartRegs *)0x40007800
#define uart8 (UartRegs *)0x40007C00

//function header
void configureUSARTBaudRate(UartRegs *usart,uint32_t peripheralFreq,uint32_t
desiredBR,OversamplingMode oversampling);
void configureUSART(UartRegs *usart,long long mode,int desiredBR,uint32_t peripheralFreq);
```

Source: The source code is the functions to configure the USART, the idea of creating these functions is to make it more easy and direct to configure what we want. This is the reason of creating the macros in the header file, the *Uart.h* .

```
void configureUSARTBaudRate(UartRegs *usart,uint32_t peripheralFreq,uint32_t
desiredBR,OversamplingMode oversampling){

      float divider;
      uint8_t fractional;
      uint8_t mantissa;

      divider = (float)peripheralFreq/(8*(2-oversampling)*desiredBR);
      mantissa = (uint8_t)divider;
      fractional = (int)((float)(divider - mantissa)*(8*(2-oversampling)));

      usart->Brr = (mantissa << 4)|fractional;
}


void configureUSART(UartRegs *usart,long long mode,int desiredBR,uint32_t peripheralFreq){

      usart->Cr1 = (uint32_t)(mode & 0xffff);
      usart->Cr2 = (uint32_t)((mode >> 16) & 0xffff);
      usart->Cr3 = (uint32_t)(mode >> 32);

      if(IS_OVERSAMPLING_16(mode))
            configureUSARTBaudRate(usart,peripheralFreq,desiredBR,USART_OVER_16);
      else
            configureUSARTBaudRate(usart,peripheralFreq,desiredBR,USART_OVER_8);
}
```

Before we call the function, we need to ensure that the USART5 clock is enable in the APB1ENR register in RCC. So we will be having a macros:

```
#define Enable_UART5_CLK_GATING()            Rcc->Apb1enr |= (1<<20)
```

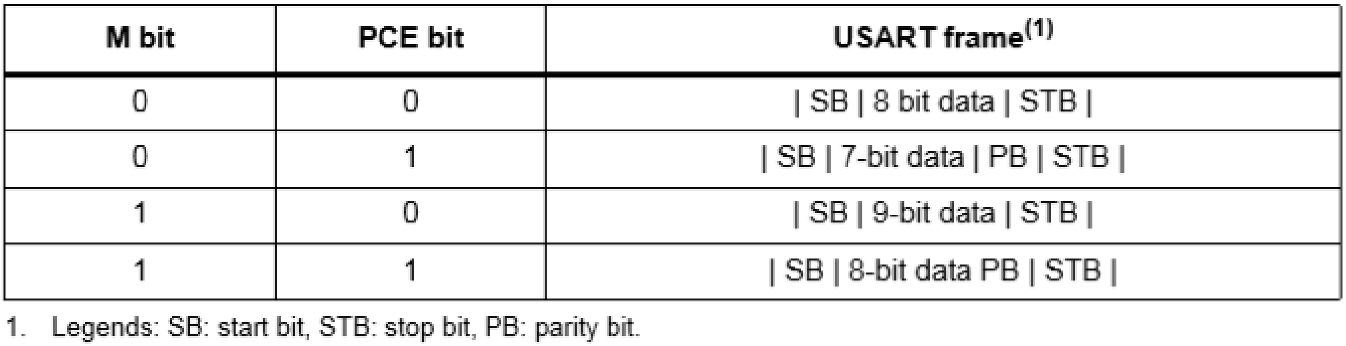
To **USE** all these functions to configure USART are as following:

```
Enable_UART5_CLK_GATING();
configureUSART(uart5,USART_TE|USART_OVER16|USART_STOPBIT_1|USART_UE|USART_PS_ODD,115200,450
00000);
```

As we can see here, we just use the functions created manually instead of going through the written functions in the CubeMX library. By doing this, we keep our code clean and easy to understand.

Transmit characters:

According the procedure requirements, the data frame needs to have 9 bits, which include 8-bit data and 1 bit parity check bit, so the configurations need to refer back to the table. According to the table, the combination is M bit = 1, PCE bit = 1.



M bit	PCE bit	USART frame(1)
0	0	| SB | 8 bit data | STB |
0	1	| SB | 7-bit data | PB | STB |
1	0	| SB | 9-bit data | STB |
1	1	| SB | 8-bit data PB | STB |

1. Legends: SB: start bit, STB: stop bit, PB: parity bit.

Figure 2: Table format for parity control, reference manual pg 971

And we want it to be odd parity .So the configuration goes like this:

```
configureUSART(uart5,USART_TE|USART_OVER16|USART_STOPBIT_1|USART_UE|
                    USART_PS_ODD|USART_M_9BIT|USART_PCE,
                    115200,45000000);
```

Now, we are trying to send character ‘m’, so refers to the ASCII table, the value in hexadecimal is 0x6D, the value to put in DR (data register) is 0x6D. 0x6D in binary is 0110 1101. There are 5 ‘1’ in 0x6D, this makes parity bit =0. USART should send out data bits from LSB first. So the send out sequence should be

Start bit	1	0	1	1	0	1	1	0	0	Stop bit
Start bit	B0	B1	B2	B3	B4	B5	B6	B7	Parity bit	Stop bit

Now, we are sending number ‘9’, value hexadecimal in ASCII is 0x39, in binary is 0011 10011. There are 4 ‘1’ in 0x39, this makes parity bit =1 hence send out sequence should be

Start bit	1	0	0	1	1	1	0	0	1	Stop bit
Start bit	B0	B1	B2	B3	B4	B5	B6	B7	Parity bit	Stop bit

Waveform from oscilloscope:

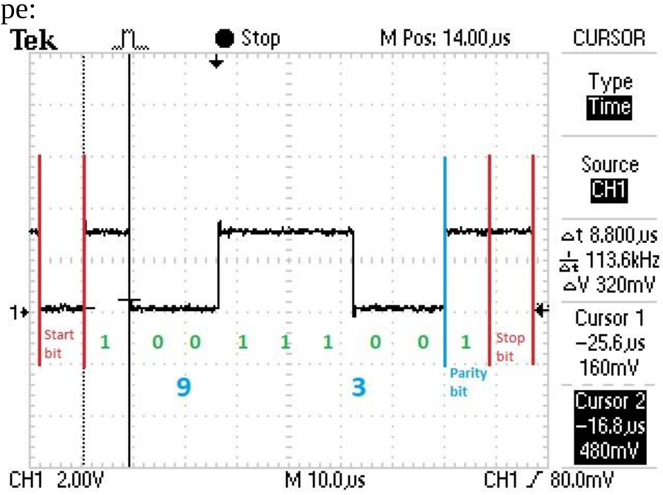


Figure 3: Waveform for value 0x39

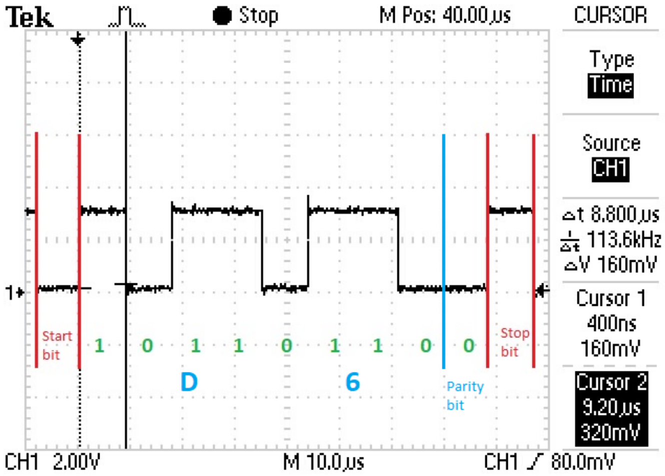


Figure 4: Waveform for value 0x6D

From the output waveform, what we estimated above before the waveform's result is correct.
Now, let's calculate the actual baud rate which we get from the oscilloscope.

1 bit period = 8.8us
Baud rate = 1 / 1 bit period
= 1 / 8.8us
= 113636.3636
≈ 113636 (bps)

Our desired baud rate is 115200 bps, the calculated actual baud rate is 113636 bps. We can notice that there is slight difference there. So, we can calculate the error percentage of this baud rate compared to desired baud rate.

$$\textbf{Error percentage} = \frac{|Actual\ BR - Desired\ BR|}{Desired\ BR} \times 100\%$$

$$= \frac{|113636 - 115200|}{115200} \times 100\%$$

$$= 1.357\%$$

Error percentage is important when we are dealing with USART communication. Says, we have the exact same baud rate for both side, then the sampling point for every bit is the same, regardless how long is the bit period, and that determines the accuracy.

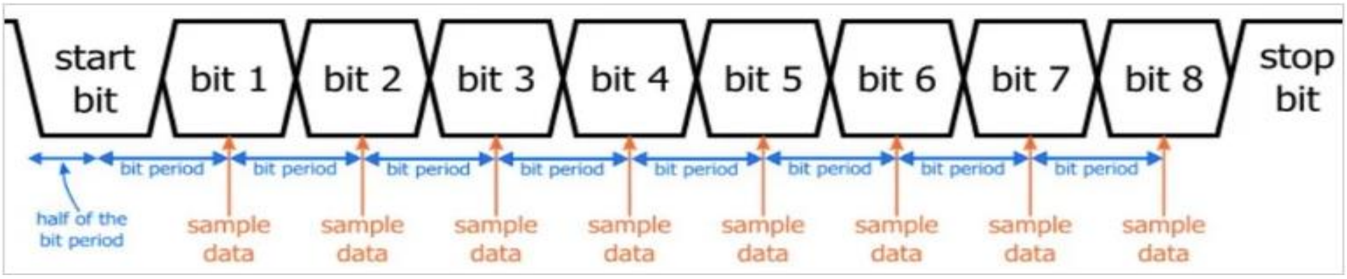


Figure 5: Waveform for exact matched BR.

If the baud rates are different, then the sampling point of the data bit is not aligned, indicates that the sampling point is different, it may sampled later or earlier than previous bit, hence, as the time goes longer, it will go more inaccurate.

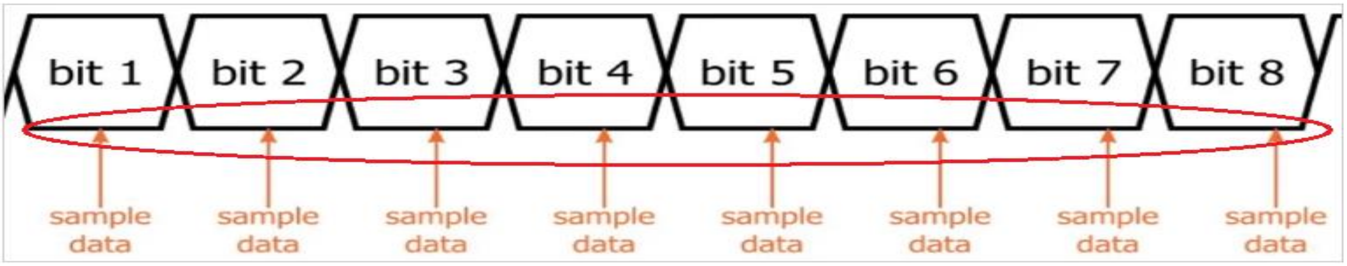


Figure 6: Different sampling point for different BR

But the unlucky thing is, we seldom have exact matched baud rate, so we want to have an error percentage as low as possible in order to have high accuracy. In this case here, the error percentage is 1.357%, which is still acceptable. Once the error percentage goes beyond 4%, we might need to change the BR value to reduce the error percentage. To have high accuracy of sampling point, the clock frequency and divider value must be chosen wisely.

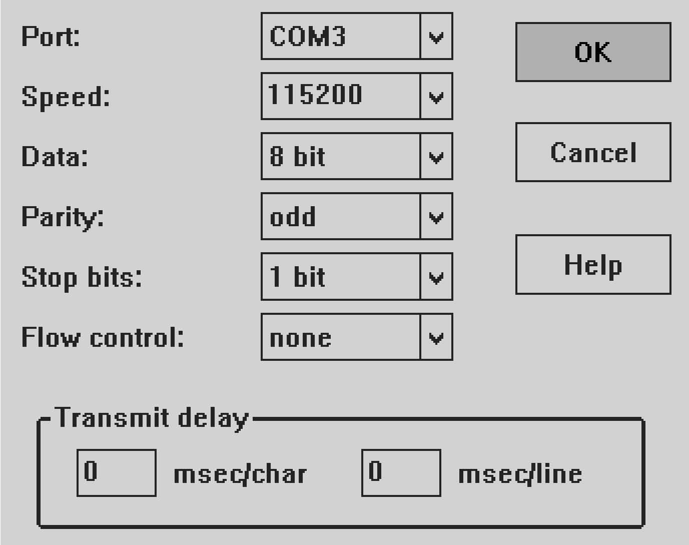
Using TeraTerm:

What we are going to do here is to transmit/send a string to, for example “Hello,WORLD!!!”. As mentioned above, before sending out the data, it needs to be ready in the TDR, which is a shadow register of DR. So the data is sent to DR and ready in TDR. In CubeMX, we cannot observe the transferring data on the debug I/O register window, so we can only observe on the TeraTerm window.

Now, as the receiver side, the configurations must be same as the configurations user defined. We will have these configurations:

1. 115,200bps
2. Odd parity
3. 9-bit data
4. Oversampling by 16
5. 1 stop bit

And we need to have the same configurations both on the TeraTerm and CubeMX in order to receive properly.



Port: COM3

Speed: 115200

Data: 8 bit

Parity: odd

Stop bits: 1 bit

Flow control: none

Transmit delay

0 msec/char 0 msec/line

Figure 7: Configurations on TeraTerm

Having all there configurations correct is important otherwise we will not receive what we want from the MCU. We will see that later.

To send out the string, just to declare a character array will do.

```
char wordsBuffer[]="Hello,WORLD!!!\t";
```

the code configurations will be the same as the above on the transmit character part:

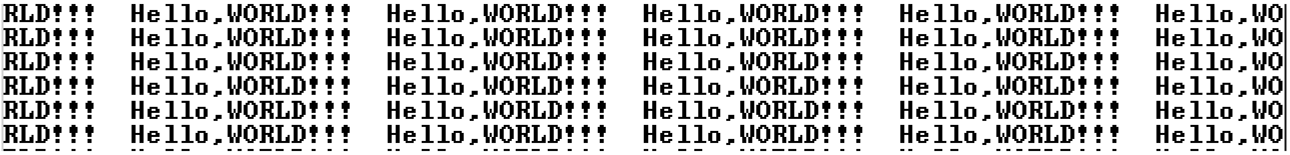
```
configureUSART(uart5,USART_TE|USART_OVER16|USART_STOPBIT_1|USART_UE|USART_PS_ODD|USART_M_9B
IT|USART_PCE,115200,45000000);
```

So the main running code will be:

```
  int length= 0;
  length = strlen(wordsBuffer);
  static int count=0;

  while (1)
  {

   if(usartIsTXRegEmpty(uart5)){
       (uart5)->Dr = wordsBuffer[count];
       count++;
             if(count == length){
             count=0;
             }
       }
 }
```

What the code here doing is just keep tracking whether the TDR is empty. Once the TDR is empty, means TxE flag is set, which indicates that next data is allowed to be sent. Static variable is to remember previous character had sent. Once reached the end of string, just revert back to 0 and repeat the same process.

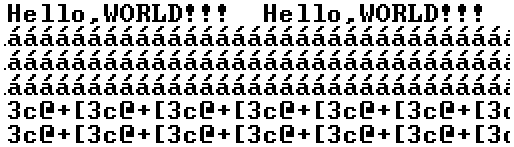
Result:



```
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
RLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WORLD!!!  Hello,WO
```

Figure 8: using BR value 115200

Let’s see what if we have a different baud rate on TeraTerm. The baud rate is changed to 9600.



```
Hello,WORLD!!!  Hello,WORLD!!!
.ááááááááááááááááááááááááááááááá
.ááááááááááááááááááááááááááááááá
.ááááááááááááááááááááááááááááááá
3c@+[3c@+[3c@+[3c@+[3c@+[3c@+[3
3c@+[3c@+[3c@+[3c@+[3c@+[3c@+[3
```

Figure 9: using BR value of 9600

We can see TeraTerm is receiving dummy character which is not the string we are expecting. This because receiver side sampling at the wrong data range although the transmitter is transmitting the correct data. So this is the consequence of using different baud rate values and having high baud rate error percentage.

From here, let’s start with using interrupt to do the same thing.

First, what is interrupt? The interrupt is a condition that processor pauses current task temporary and go for a different task, the task will be written into Interrupt Service Routine (ISR). After the ISR task is completed, processor will then only continue the task previously paused.

In this case here, the ISR function must be written ass following:

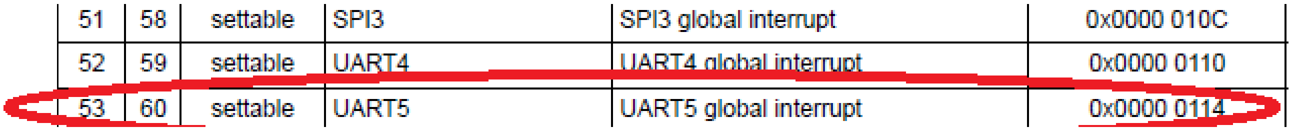
```
        extern char wordsBuffer[];

void UART5_IRQHandler(void){
        int length= strlen(wordsBuffer);

        if(usartIsTXRegEmpty(uart5)){
                (uart5)->Dr = wordsBuffer[count];
                count++;
                if(count == length){
                        count=0;
                }
        }
}
```

Be careful of ISR function name, all the ISR function name had been written in the CUbeMX. So, we need to find out the predefined name in the *startup_stm32f429xx.s*. We cannot change the function name in the file because it is system defined therefore CubeMX will change back whatever name we changed in the file when we regenerate the code. But fortunately we can put the ISR in the other source file, this is because all the function name is weakly bonded,

```
  .weak      UART5_IRQHandler
```

When we compile, the linker will linked to ISR in other file (UART.c) instead of the system interrupt file(stm32f4xx_it.c),.

Next thing is to enable all the interrupt related bits. Here, Nested Vector Interrupt Controller (NVIC) is introduced. NVIC enables low latency interrupt processing and efficient processing of late arriving interrupts. All interrupts including the core exceptions are managed by the NVIC. To enable uart5 interrupt, we need to refer to the table in reference manual pg755, table 62, to find relative position of uart5 in NVIC.



51	58	settable	SPI3	SPI3 global interrupt	0x0000 010C
52	59	settable	UART4	UART4 global interrupt	0x0000 0110
53	60	settable	UART5	UART5 global interrupt	0x0000 0114

Figure 10: Position of UAR5 in vector table, refrence manual table 62.

From the figure above, the position of interrupts of UART5 is 53, and 60 indicates the priority of interrupt. So, a function is written to configure the NVIC. The bit we need to write to is in Interrupt Set Enable register (ISER).

```
void nvicEnableInterrupt(int interruptNumber)
{
	int n,bit;

	n = interruptNumber/32;
	bit = interruptNumber%32;

	nvic->ISER[n] = 1 << bit;
}
```

Now we just to need to also enable the *RXNEIE* bit in uart:

```
#define Enable_TxE_Interrupt(usart)                        (usart)->Cr1 |= USART_TXEIE

nvicEnableInterrupt(53);
Enable_TxE_Interrupt(uart5);
While(1){};
```

All the pevious configurations will just remain the same, removes the while loop code, and once the TDR is empty , interrupt is generated and ISR will be service.

Turn on, turn off and blink LED using TeraTerm:

Example above is using UART to transmit, now we are transmitting and receiving simultaneously for displaying on console, then type a command to turn on ,turn off or blink led. For example, if we type “TURN ON LED” on TeraTerm, which is the PC host, the character is received by MCU in a buffer and transmitted again back to PC and display it. So we will see the character we typed is displaying on the console. The use *strcmp* to identify the command whether to turn on or turn off the LED. For “BLINK LED” we want it to blink 4 times in 1 second, here both HAL library and interrupt are used.

So the configuration procedure will be:

1) Enable all the clock gating.
2) Configure all the pins, LED pin, Transmit pin (PC12), Receive pin (PD2)
3) Enable transmission and receiver mode.

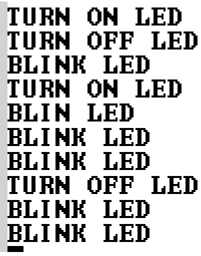
So the code goes like this:

```
  Reset_UART5();
  Unreset_UART5();
  Reset_GPIOC();
  Unreset_GPIOC();
  Reset_GPIOD();
  Unreset_GPIOD();
  Reset_GPIOG();
  Unreset_GPIOG();

  Enable_UART5_CLK_GATING();
  Enable_GPIOC_CLOCK_GATING();
  Enable_GPIOD_CLOCK_GATING();
  Enable_GPIOG_CLOCK_GATING();

//For green led
configureGPIO_Pin(GpioG,GpioPin13,GPIO_OUTPUT|GPIO_PUSH_PULL|GPIO_VERY_HI_SPEED|GPIO_NO_PUL
L);

//PC12 is TX for uart5
configureGPIO_Pin(GpioC,GpioPin12,GPIO_ALT_FUNC|GPIO_PUSH_PULL|GPIO_VERY_HI_SPEED|GPIO_NO_P
ULL);
  configureAltFunction(GpioC,GpioPin12,AF_8);

//PD2 is RX for uart5
configureGPIO_Pin(GpioD,GpioPin2,GPIO_ALT_FUNC|GPIO_PUSH_PULL|GPIO_MED_SPEED|GPIO_NO_PULL);
  configureAltFunction(GpioD,GpioPin2,AF_8);

configureUSART(uart5,USART_TE|USART_OVER16|USART_STOPBIT_1|USART_UE|                  \
                                 USART_PS_ODD|USART_M_9BIT|USART_PCE|USART_RE,        \
                                 115200,45000000);
```

We need to ensure that what we configured is correct else it will be not working.So for the main code is put in the while loop:

```
  while (1)
  {
         if(usartIsRXNERegNotEmpty(uart5)){
               receiveBuffer[count] = (uart5)->Dr;

               if(usartIsTXRegEmpty(uart5))
                      (uart5)->Dr = receiveBuffer[count];

               if(receiveBuffer[count] == '\n'){
                      receiveBuffer[count] = '\0';

                      if(!strcmp(receiveBuffer,"TURN ON LED")){
                            gpioWritePins(GpioG,GpioPin13,GPIO_Pin_Set);
                      }
                      else if(!strcmp(receiveBuffer,"TURN OFF LED")){
                            gpioWritePins(GpioG,GpioPin13,GPIO_Pin_Reset);
                      }
                     else if(!strcmp(receiveBuffer,"BLINK LED")){
                            for(int i=0;i<8;i++){
                                   gpioTogglePins(GpioG,GpioPin13);
                                   HAL_Delay(125);
                            }
                      }
                      count=0;
               }
               else
                      count++;

         }
  }
```

The console on the TeraTerm will be like this:



```
TURN ON LED
TURN OFF LED
BLINK LED
TURN ON LED
BLIN LED
BLINK LED
BLINK LED
TURN OFF LED
BLINK LED
BLINK LED
```

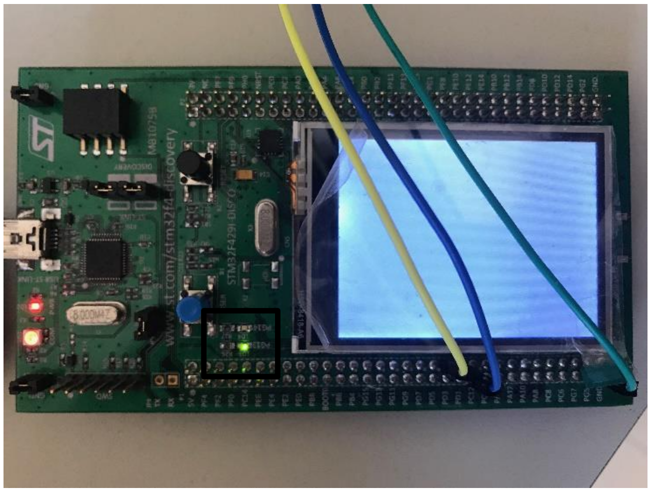


Figure 11: “TURN ON LED” command is sent

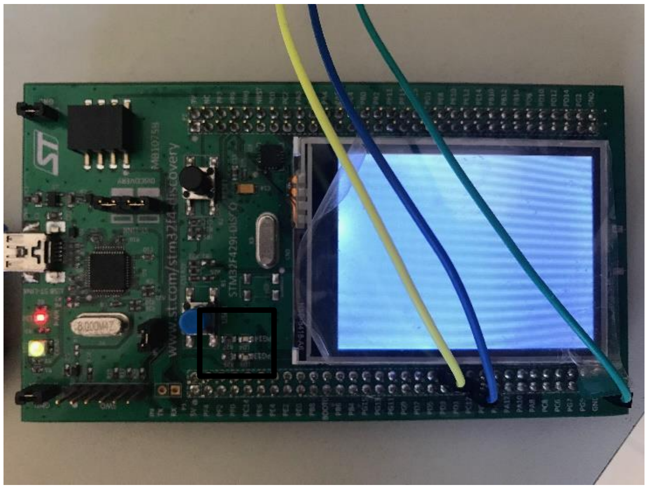


Figure 12: "TURN OFF LED" command is sent

Now, let’s try use the interrupt to do the same task, and the “BLINK LED” is going to use timer 2 interrupt instead of using HAL_Delay.

Interrupts of timer is generated when the counter (TIM_CNT) is overflow in upcounting mode, or underflow in downcounting mode. We need to set Auto-Reload Register (ARR) a value that gives a correct delay, then counter will count up from 0 to ARR value, once overflow occurs, counter is reset back to 0. In down counting moew, the counter starts to count down from ARR value to 0, once underflow is detected, interrupts is generated.

So first thing is to write the ISR function and is written in *Timer.c*:

```
void TIM2_IRQHandler(void){

	if(timer_GET_FLAG(timer2,timer_UIF)){
		gpioTogglePins(GpioG,GpioPin13);
		togglenum++;
		timer_CLEAR_FLAG(timer2,timer_UIF);
		 if(togglenum>8){
		  timer_DISABLE_INTERRUPT(timer2,timer_UIE);
		  timer_COUNTER_DISABLE(timer2);
		  togglenum = 0;
		 }
	}
}
```

togglenum variable is to keep tracking the toggling event. We need it to blink 4 times in 1 second, total toggling is 8 times, so interrupt has to be generated 8 times, 1 interrupt period is 1s/8 = 125ms.

Let’s calculate the ARR value to gain that 125ms delay by using 45MHz APB1 clock frequency. Let the clock divides by 50, prescaler is 49

$$\text{Divided clock frequency} = \frac{fclk}{prescaler+1}$$

$$= \frac{45MHz}{49+1}$$

$$= 0.9\text{MHz}$$

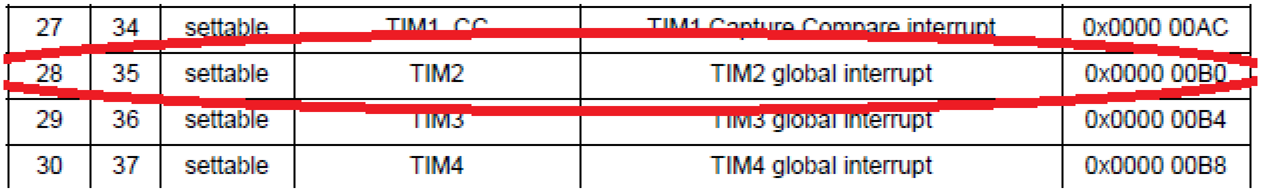
$$\text{ARR value} = \frac{Desired\ \ interrupt\ period}{\frac{1}{divided\ clock\ frequency}}$$

$$= \frac{125ms}{\frac{1}{0.9MHz}}$$

$$= 112500$$

So now the counter count up to 112500 and once overflow, then overflow interrupt generated and enter ISR to do the toggling, after the toggling task done, we need to clear the Update Interrupt Flag by software else it wont be entering the ISR for the next interrupt.

Next is to find the position of timer 2 interrupt in NVIC and configure it.



27	34	settable	TIM1_CC	TIM1 Capture Compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8

Figure 13: NVIC position of Timer 2

To configure it, the code goes:

```
nvicEnableInterrupt(28);
```

Now all the pre-work job is done, put the main code in the ISR will do.

```
void UART5_IRQHandler(void){

	if(usartIsRXNERegNotEmpty(uart5)){
		receiveBuffer[count] = (uart5)->Dr;

		if(usartIsTXRegEmpty(uart5))
			(uart5)->Dr = receiveBuffer[count];

		if(receiveBuffer[count] == '\n'){
			receiveBuffer[count] = '\0';

			if(!strcmp(receiveBuffer,"TURN ON LED")){
				gpioWritePins(GpioG,GpioPin13,GPIO_Pin_Set);

			}
			else if(!strcmp(receiveBuffer,"TURN OFF LED")){
				gpioWritePins(GpioG,GpioPin13,GPIO_Pin_Reset);
			}
		       else if(!strcmp(receiveBuffer,"BLINK LED")){
				timer2->Arr = 112500;
				timer2->Prescaler = 49;
```

```
                        timer2->Counter = 0;

                        nvicEnableInterrupt(28);
                        timer_ENABLE_INTERRUPT(timer2,timer_UIE);
                        timer_COUNTER_ENABLE(timer2);

                    }
                    count=0;

                }
                else
                    count++;
}
```

Conclusion:

Before we use any peripherals, we need to know how the peripheral should be configured. For receive and transmit, we must ensure that Transmission Enable bit and Receive Enable bit are set. Next is the communication mode we chose, whether we want to have a parity check bit or 2 stop bit, and make sure that both side of the communication is having the same mode. Then the important thing is the baud rate, because of we ignored some digits during the calculations, hence, we are having different value from desired baud rate. So it is important to check the error percentage is as low as possible.

Baud rate = 1 / 1 bit period

Error percentage = $\frac{|Actual\ BR - Desired\ BR|}{Desired\ BR} \times 100\%$

Next is whether we want to use interrupt to do the transmission or receiving action. If that is needed, remember to check the NVIC table to the related UART and configure to the proper position. Then only we can enable the interrupt in USART level. Here, the ISR function name must be correct according to the predefined name in the system. Nonetheless, if we are using the timer interrupt, do the same thing as UART did, configure the correct position in NVIC and also enable the relative timer's interrupt. To determine the interrupt period we want, during the calculations, we need to know the frequency of the peripheral, different peripherals may having different frequency because some of the peripherals are on the different bus. Then calculate the ARR value

Divided clock frequency = $\frac{fclk}{prescaler+1}$

ARR value = $\frac{Desired\ \ interrupt\ period}{\frac{1}{divided\ clock\ frequency}}$

After the ISR is serviced, we need to clear the update interrupt flag for the timer. So that's the procedure of doing the interrupts.