

Universal Synchronous Asynchronous Receiver Transmitter protocol

Understand about USART protocol, how to configure its clock, baud rate, and working mode. Steps to setup an USART module using HAL. Example to print messages, and get user's inputs in polling and interrupt modes.

`#arm #stm32 #usart #uart`

Last update: April 28, 2021

Table of Content

1. Hardware
 - 1.1. Wires
 - 1.2. Flow control
 - 1.3. Data frame
 - 1.4. Clock
 - 1.5. Baud rate
 - 1.6. Multiprocessor
2. Memory Map
3. Register Map
4. HAL Software
5. Lab: Send Data
 - 5.1. Create project
 - 5.2. Enable USART1
 - 5.3. Generated code
 - 5.4. User code
 - 5.5. Connect UART to PC
 - 5.6. Build and Run
6. Lab: Receive by Polling
7. Lab: Receive by Interrupt
 - 7.1. Interrupt Mode
 - 7.2. Enable interrupt
 - 7.3. Handler interrupt
8. Appendix

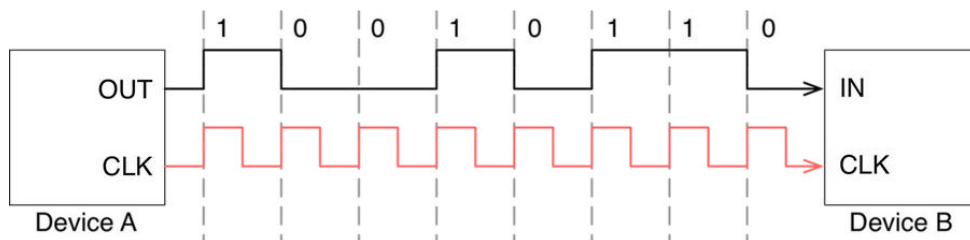
1. Hardware

Universal Synchronous/Asynchronous Receiver/Transmitter interface, also simply known as USART, is a device that translates a parallel sequence of bits (usually grouped in a byte) in a continuous stream of signals flowing on a single wire.

1.1. Wires

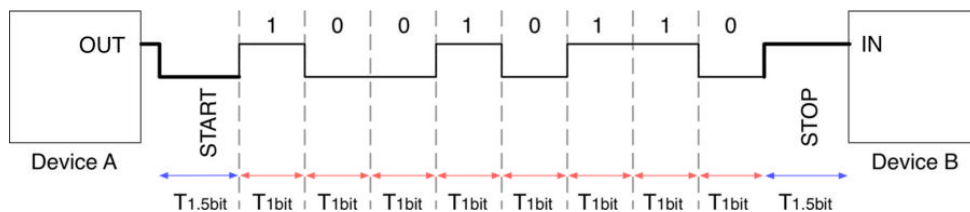
When the information flows between two devices inside a common channel, both devices (as the sender and also the receiver) have to agree on the *timing*, that defines how long it takes to transmit each individual bit of the information.

- In a *synchronous* transmission, the sender and the receiver share a common clock generated by one of the two devices



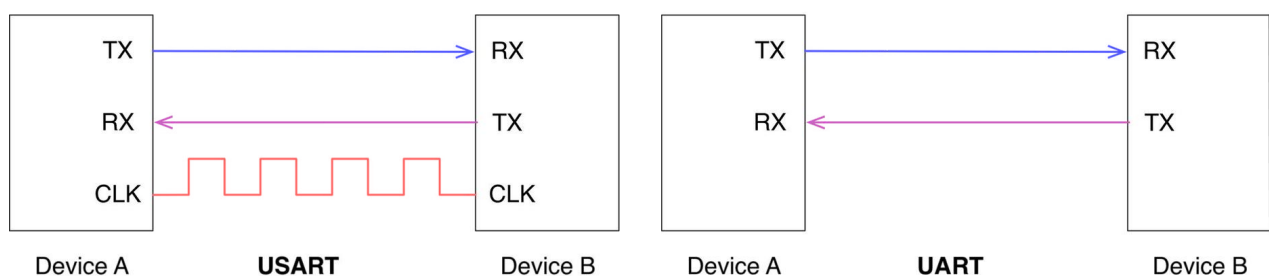
Shared clock in synchronous USART

- In an *asynchronous* transmission, the clock line is omitted, and both devices have an internal clock source and a mechanism to detect start/ stop bit.



One line of data in asynchronous USART

In a bi-direction communication, it needs a pairs of lines for Transmitter (TX) and Receiver (RX):



USART vs UART

1.2. Flow control

The presence of a dedicated clock line, or a common agreement about transmission frequency, does not guarantee that the receiver of a byte stream is able to process them at the same transmission rate of the master.

For this reason, some communication standards, like the *RS232* and the *RS485*, provide the possibility to use a dedicated *Hardware Flow Control* line.

For example, two devices communicating using the *RS232* interface can share two additional lines, named Request To Send(RTS) and Clear To Send(CTS): the sender sets its RTS, which signals the receiver to begin monitoring its data input line. When ready for data, the receiver will raise its complementary line, CTS, which signals the sender to start sending data, and for the sender to begin monitoring the slave's data output line.

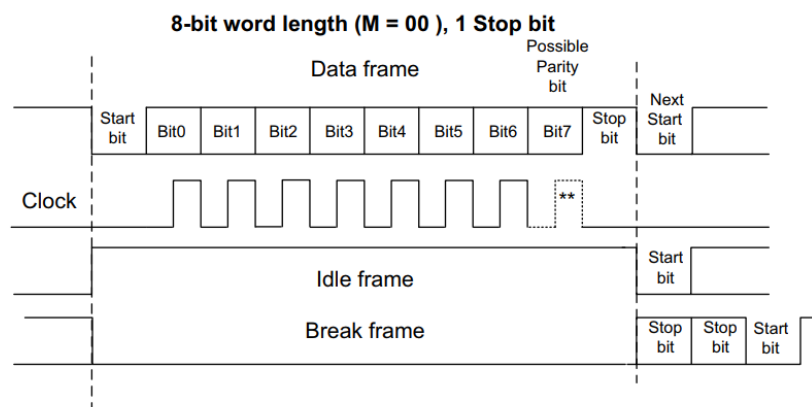
1.3. Data frame

The frames are comprised of:

- An Idle Line prior to transmission or reception
- A start bit
- A data word (7, 8 or 9 bits) least significant bit first
- 0.5, 1, 1.5, 2 stop bits indicating that the frame is complete

By default, the signal (TX or RX) is in low state during the start bit. It is in high state during the stop bit. These values can be inverted, separately for each signal, through polarity configuration control.

- An *Idle character* is interpreted as an entire frame of “1”s (the number of “1”s includes the number of stop bits).
- A *Break character* is interpreted on receiving “0”s for a frame period. At the end of the break frame, the transmitter inserts 2 stop bits.



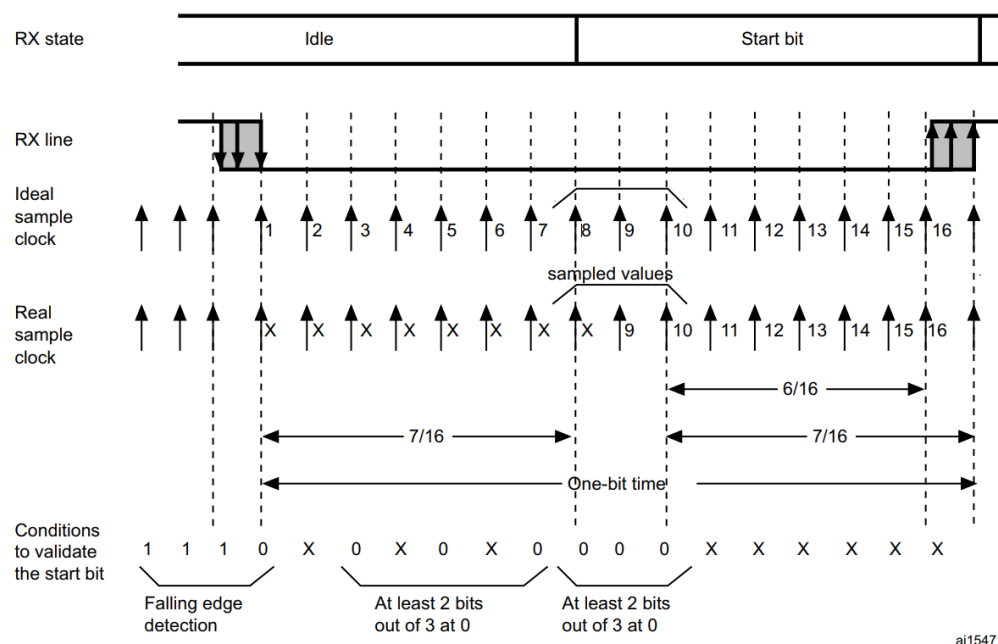
Frames in USART

1.4. Clock

The choice of the clock source is done through the Clock Control system (see Section Reset and clock control (RCC)). The clock source must be chosen before enabling the USART (by setting the UE bit).

Choosing LSE or HSI as clock source may allow the USART to receive data while the MCU is in low-power mode.

Clock source is used to do oversampling by 16 or by 8 to detect the start bit. It samples the RX line and try to detect a falling edge and following patterns of zeros.



Detect start bit using oversampling

1.5. Baud rate

Baud rate determines the speed of transmitting and receiving, as the speed is depend on the clock source and USARTDIV value.

USARTDIV is an unsigned fixed point number that is coded on the USART_BRR register.

- When $OVER8 = 0$, $BRR = USARTDIV$.
- When $OVER8 = 1$
 - $BRR[2:0] = USARTDIV[3:0]$ shifted 1 bit to the right.
 - $BRR[3]$ must be kept cleared.
 - $BRR[15:4] = USARTDIV[15:4]$

Example: To obtain 9600 baud with $f_{CK} = 8 \text{ MHz}$.

- *In case of oversampling by 16:*

$$BRR = USARTDIV = 8\,000\,000/9600 = 833d = 0341h$$

- *In case of oversampling by 8:*

$$USARTDIV = 2 * 8\,000\,000/9600 = 1666,66 (\sim 1667d) = 683h$$

$$BRR[3:0] = 3h \gg 1 = 1h$$

$$BRR = 0x681$$

Auto baud rate detection

The USART is able to detect and automatically set the USART_BRR register value based on the reception of one character. Automatic baud rate detection is useful under two circumstances:

- The communication speed of the system is not known in advance
- The system is using a relatively low accuracy clock source and this mechanism allows the correct baud rate to be obtained without measuring the clock deviation.

Before activating the auto baud rate detection, the auto baud rate detection mode must be chosen. There are various modes based on different character patterns.

Prior to activating auto baud rate detection, the USART_BRR register must be initialized by writing a non-zero baud rate value.

1.6. Multiprocessor

In multiprocessor communication, the following bits are to be kept cleared:

- LINEN bit in the USART_CR2 register,
- HDSEL, IREN and SCEN bits in the USART_CR3 register.

It is possible to perform multiprocessor communication with the USART (with several USARTs connected in a network). For instance one of the USARTs can be the master, its TX output connected to the RX inputs of the other USARTs. The others are slaves, their respective TX outputs are logically ANDed together and connected to the RX input of the master.

2. Memory Map

Refer to Datasheet document of the target MCU. For example, DS8668 for STM32F051x.

3. Register Map

Refer to Reference Manual document of the target MCU. For example, RM0091 for STM32F051x.

4. HAL Software

The Hardware Abstract Layer (HAL) is designed so that it abstracts from the specific peripheral memory mapping. But, it also provides a general and more user-friendly way to configure the peripheral, without forcing the programmers to know how to configure its registers in detail.

ST HAL define a handler for USART ports like below:

```
typedef struct {
    USART_TypeDef *Instance; /* USART registers base address */
    USART_InitTypeDef Init; /* USART communication parameters */
    USART_AdvFeatureInitTypeDef AdvancedInit; /* USART Advanced Features
initialization parameters */
    uint8_t *pTxBuffPtr; /* Pointer to USART Tx transfer Buffer */
    uint16_t TxXferSize; /* USART Tx Transfer size */
    uint16_t TxXferCount; /* USART Tx Transfer Counter */
    uint8_t *pRxBuffPtr; /* Pointer to USART Rx transfer Buffer */
    uint16_t RxXferSize; /* USART Rx Transfer size */
    uint16_t RxXferCount; /* USART Rx Transfer Counter */
    DMA_HandleTypeDef *hdmatx; /* USART Tx DMA Handle parameters */
    DMA_HandleTypeDef *hdmarx; /* USART Rx DMA Handle parameters */
    HAL_LockTypeDef Lock; /* Locking object */
    __IO HAL_USART_StateTypeDef State; /* USART communication state */
    __IO HAL_USART_ErrorTypeDef ErrorCode; /* USART Error code */
} USART_HandleTypeDef;
```

excerpt from *Description of STM32F0 HAL and low-layer drivers*

How to use USART HAL

1. Declare a `USART_HandleTypeDef` handle structure (eg. `USART_HandleTypeDef huart`).
2. Initialize the USART low level resources by implementing the `HAL_USART_MspInit()` API when needed:
 - Enable the USARTx interface clock.
 - USART pins configuration:
 - Enable the clock for the USART GPIOs.
 - Configure these USART pins as alternate function pull-up.
 - NVIC configuration if use interrupt process (`HAL_USART_Transmit_IT()` and `HAL_USART_Receive_IT()` APIs):
 - Configure the USARTx interrupt priority.
 - Enable the NVIC USART IRQ handle.
 - USART interrupts handling:

- DMA Configuration if use DMA process (`HAL_UART_Transmit_DMA()` and `HAL_UART_Receive_DMA()` APIs):
 - Declare a DMA handle structure for the Tx/Rx channel.
 - Enable the DMAx interface clock.
 - Configure the declared DMA handle structure with the required Tx/Rx parameters.
 - Configure the DMA Tx/Rx channel.
 - Associate the initialized DMA handle to the UART DMA Tx/Rx handle.
 - Configure the priority and enable the NVIC for the transfer complete interrupt on the DMA Tx/Rx channel.
- 3. Program the Baud Rate, Word Length, Stop Bit, Parity, Hardware flow control and Mode (Receiver/Transmitter) in the huart handle Init structure.
- 4. If required, program UART advanced features (TX/RX pins swap, auto Baud rate detection,...) in the huart handle AdvancedInit structure.
- 5. For the UART asynchronous mode, initialize the UART registers by calling the `HAL_UART_Init()` API.
- 6. For the UART Half duplex mode, initialize the UART registers by calling the `HAL_HalfDuplex_Init()` API.
- 7. For the UART Multiprocessor mode, initialize the UART registers by calling the `HAL_MultiProcessor_Init()` API.
- 8. For the UART RS485 Driver Enabled mode, initialize the UART registers by calling the `HAL_RS485Ex_Init()` API.

5. Lab: Send Data

This project aims to learn how to configure USART via STM32CubeIDE and STM32CubeMX.

Target board: STM32F0 Discovery

Application requirements:

- Enable USART1 on board
- Transmit a log with increasing counter every 1 second

5.1. Create project

Create new project via CubeMX and use default settings for the discovery board.

5.2. Enable USART1

Open *Connectivity* section in *Pinout & Configs* tab and select USART1 module, then edit some settings:

- Mode: Asynchronous
- Parameter:
 - Baud rate: 115200 bps
 - Word length: 8 b (including Parity)
 - Parity: None
 - Stop bits: 1

The screenshot shows the STM32CubeMX Pinout & Configuration window for the F051_Disco_UART.ioc project. The left sidebar shows the 'Connectivity' section with USART1 selected. The main window displays the 'USART1 Mode and Configuration' settings.

Mode: Asynchronous

Configuration:

- Parameter Settings:**
 - Baud Rate: 115200 Bits/s
 - Word Length: 8 Bits (including Parity)
 - Parity: None
 - Stop Bits: 1
- Advanced Parameters:**
 - Data Direction: Receive and Transmit
 - Over Sampling: 16 Samples
 - Single Sample: Disable
- Advanced Features:**
 - Auto Baudrate: Disable
 - TX Pin Active Level Inversion: Disable
 - RX Pin Active Level Inversion: Disable
 - Data Inversion: Disable
 - TX and RX Pins Swapping: Disable
 - Overrun: Enable
 - DMA on RX Error: Enable
 - MSB First: Disable

Search Signals:

Pin Name	Signal on Pin	GPIO output...	GPIO mode	GPIO Pull-u...	Maximum o...	Fast Mode	User Label	Modified
PA9	USART1_TX	n/a	Alternate F...	No pull-up a...	High	n/a		<input type="checkbox"/>
PA10	USART1_RX	n/a	Alternate F...	No pull-up a...	High	n/a		<input type="checkbox"/>

Enable USART1

Note that PA9 and PA10 are automatically configured to Alternative Function to use as USART1 pinout.

5.3. Generated code

When generate code from Pin configs, there are some noticeable code blocks:

Peripheral instance

IDE will add an instance handler for the USART1 module in *main.c*. This instance will be used for manage USART1 peripheral then it should be global access:

```
UART_HandleTypeDef huart1;
```

Init functions

The function `SystemClock_Config()` is included to setup the system clock, bus clocks. In addition, it will set the clock source for the USART1:

```
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
    RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

    /** Initializes the RCC Oscillators according to the specified parameters
    * in the RCC_OscInitTypeDef structure.
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSISState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL12;
    RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }
    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                  |RCC_CLOCKTYPE_PCLK1;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
    {
        Error_Handler();
    }
}
```

```

PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART1;
PeriphClkInit.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK1;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
    Error_Handler();
}
}

```

The function `MX_USART1_UART_Init()` initiates the USART1 instance with the values put into the init struct. This function, at the end, calls to `HAL_UART_Init()` which is an HAL function to check the init params and finally calls to `HAL_UART_MspInit()` to do low-level configs.

```

static void MX_USART1_UART_Init(void)
{
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 115200;
    huart1.Init.WordLength = UART_WORDLENGTH_8B;
    huart1.Init.StopBits = UART_STOPBITS_1;
    huart1.Init.Parity = UART_PARITY_NONE;
    huart1.Init.Mode = UART_MODE_TX_RX;
    huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart1.Init.OverSampling = UART_OVERSAMPLING_16;
    huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart1) != HAL_OK)
    {
        Error_Handler();
    }
}

```

The function `HAL_UART_MspInit()` is generated in `stm32f0xx_hal_msp.c` to override the function declared in HAL Lib. This low-level config will setup the peripheral clocks, and set alternative functions on GPIO pins.

```

void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    if(huart->Instance==USART1)
    {
        /* Peripheral clock enable */
        __HAL_RCC_USART1_CLK_ENABLE();
        __HAL_RCC_GPIOA_CLK_ENABLE();

        /**USART1 GPIO Configuration
        PA9      -----> USART1_TX
        PA10     -----> USART1_RX
        */
        GPIO_InitStruct.Pin = GPIO_PIN_9|GPIO_PIN_10;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
        GPIO_InitStruct.Alternate = GPIO_AF1_USART1;
    }
}

```

```

    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}
}

```

5.4. User code

With generated code, just need to use `HAL_UART_Transmit()` function to send a buffer over the USART instance. Let's make a buffer, a counter, and a message every 1 second.

```

#include <stdio.h> // sprintf
#include <string.h> // strlen

int main(void)
{
    char counter = 0;
    char buffer[16] = {0}; // counter=xxx\n\r

    while (1)
    {
        sprintf(buffer, "counter=%03d\n\r", counter++);
        HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(buffer),
        HAL_MAX_DELAY);
        HAL_Delay(1000);
    }
}

```

5.5. Connect UART to PC

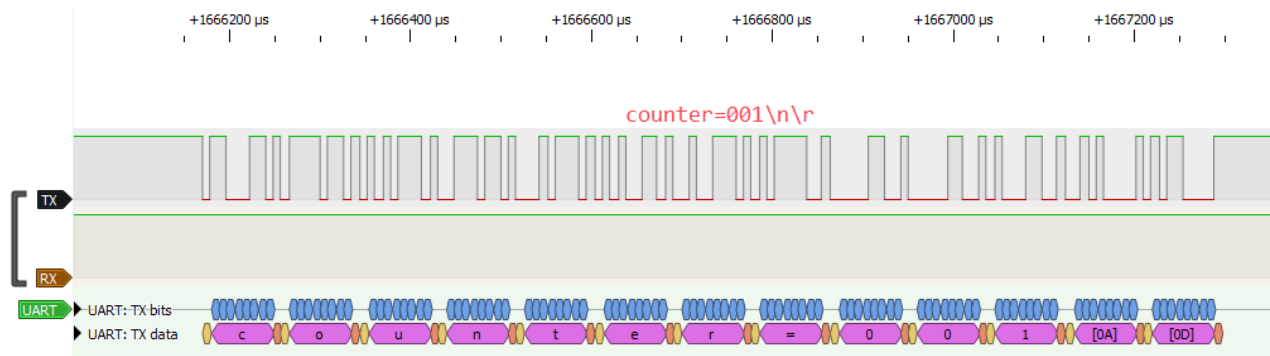
Because STM32F0 Discovery does not have a Virtual COM port on ST-LINK/V2, so use a TTL-to-USB converter.

Then connect pins PA9 and PA10 to UART terminal on PC. It's recommend to check the voltage because board is running at 3.3V while PC USB or COM port is running at 5V.

Another option is that an Arduino Uno can be used as a COM bridge if put the MCU into inactive state by connecting RESET pin to GND, and directly use both TX, RX pins which are connected to Arduino Virtual COM port.

5.6. Build and Run

Build and run the code on the target board, and open a COM terminal on PC to see the message from the target board. Use a digital logic analyser to see raw bits transferred in RX and TX pins.



UART output on digital logic analyser

6. Lab: Receive by Polling

In *polling* mode, also called *blocking* mode, the main application, or one of its threads, synchronously waits for the data transmission and reception. This is the most simple form of data communication using this peripheral, and it can be used when the transmit rate is not too much low and when the UART is not used as critical peripheral.

Based on the Lab 1, next step is to read from UART in polling mode.

⚠ Polling mode

- Block the program flow
- Have to wait for the exact number of characters

Target board: STM32F0 Discovery

Application requirements:

- Read data from UART
- If data is "stop", then do not print counter value
- If data is "resume", then resuming printing counter value

Use this function `HAL_UART_Receive(&huart1, (uint8_t *)buffer, 6, 2000)`, which means:

- All received data is written into *buffer* (re-use it ^^)
- Function will exit if one of the below condition meets:
 - 6 chars are received, or
 - 2000 ms timeout, /* use HAL_MAX_DELAY will block the while loop */

Let's see the modified code, with return result is checked to print the debug information.

```

const char msg_ok[] = "\tOK\n\r";
const char msg_busy[] = "\tBUSY\n\r";
const char msg_error[] = "\tERROR\n\r";
const char msg_timeout[] = "\tTIME OUT\n\r";
int main(void)
{
    char pause = 0;
    char counter = 0;
    char buffer[16] = {0};
    HAL_StatusTypeDef ret;

    while (1)
    {
        if (pause == 0) {
            sprintf(buffer, "counter=%03d\n\r", counter++);
            HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(buffer),
HAL_MAX_DELAY);
        }

        ret = HAL_UART_Receive(&huart1, (uint8_t *)buffer, 6, 2000); // try to
use HAL_MAX_DELAY to see the effect

        if (ret == HAL_OK) {
            HAL_UART_Transmit(&huart1, msg_ok, strlen(msg_ok), HAL_MAX_DELAY);
        } else if (ret == HAL_BUSY) {
            HAL_UART_Transmit(&huart1, msg_busy, strlen(msg_busy),
HAL_MAX_DELAY);
        } else if (ret == HAL_ERROR) {
            HAL_UART_Transmit(&huart1, msg_error, strlen(msg_error),
HAL_MAX_DELAY);
        } else if (ret == HAL_TIMEOUT) {
            HAL_UART_Transmit(&huart1, msg_timeout, strlen(msg_timeout),
HAL_MAX_DELAY);
        }

        if (strncmp(buffer, "stop", 4) == 0) {
            pause = 1;
        } else if (strncmp(buffer, "resume", 6) == 0) {
            pause = 0;
        }
    }
}

```

The screenshot shows an IDE with two panels. The left panel displays C code in `main.c` (lines 101-124) and `startup_stm32f051r8tx.s`. The code implements a UART receiver loop that checks for `HAL_OK` or `HAL_TIMEOUT` status and manages a buffer. The right panel shows a 'Variable Watch' window with the following data:

Name	Type	Value
buffer[0]	char	115 's'
buffer[1]	char	115 's'
buffer[2]	char	115 's'
buffer[3]	char	115 's'
buffer[4]	char	115 's'
buffer[5]	char	115 's'
buffer[6]	char	114 'r'
buffer[7]	char	61 '='
buffer[8]	char	48 '0'
buffer[9]	char	48 '0'
buffer[10]	char	49 '1'
buffer[11]	char	10 '\n'
buffer[12]	char	13 '\r'
buffer[13]	char	0 '\0'
buffer[14]	char	0 '\0'
buffer[15]	char	0 '\0'
ret	HAL_StatusTypeDef	HAL_OK

Debug to see the received status

Bug: Uncontrollable input

It's hard to input correct command because the timeout behavior may break the flow, and the number of remaining characters is not predictable.

Timeout mechanism

It is important to remark that the timeout mechanism offered used in the receiving function works only if the `HAL_IncTick()` routine is called every *1ms*, as done by the code generated by CubeMX (the function that increments the HAL tick counter is called inside the SysTick timer ISR).

7. Lab: Receive by Interrupt

7.1. Interrupt Mode

Every USART peripheral provides the interrupts listed below:

Interrupt Event	Event Flag	Enable Control Bit
Transmit Data Register Empty	TXE	TXEIE
Clear To Send (CTS) flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE

Interrupt Event	Event Flag	Enable Control Bit
Overrun Error Detected	ORE	RXNEIE
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multi buffer communication	NF or ORE or FE	EIE

These events generate an interrupt if the corresponding *Enable Control Bit* is set. However, STM32 MCUs are designed so that all these IRQs are bound to just one ISR for every USART peripheral. It is up to the user code to analyze the corresponding *Event Flag* to infer which interrupt has generated the request.

The CubeHAL is designed to automatically do this job for us. Then user is warned about the interrupt generation thanks to a series of callback functions invoked by the `HAL_UART_IRQHandler()`, which must be called inside the ISR.

From a technical point of view, there is not so much difference between UART transmission in polling and in interrupt mode. Both the methods transfer an array of bytes using the UART *Data Register* (DR) with the following algorithm:

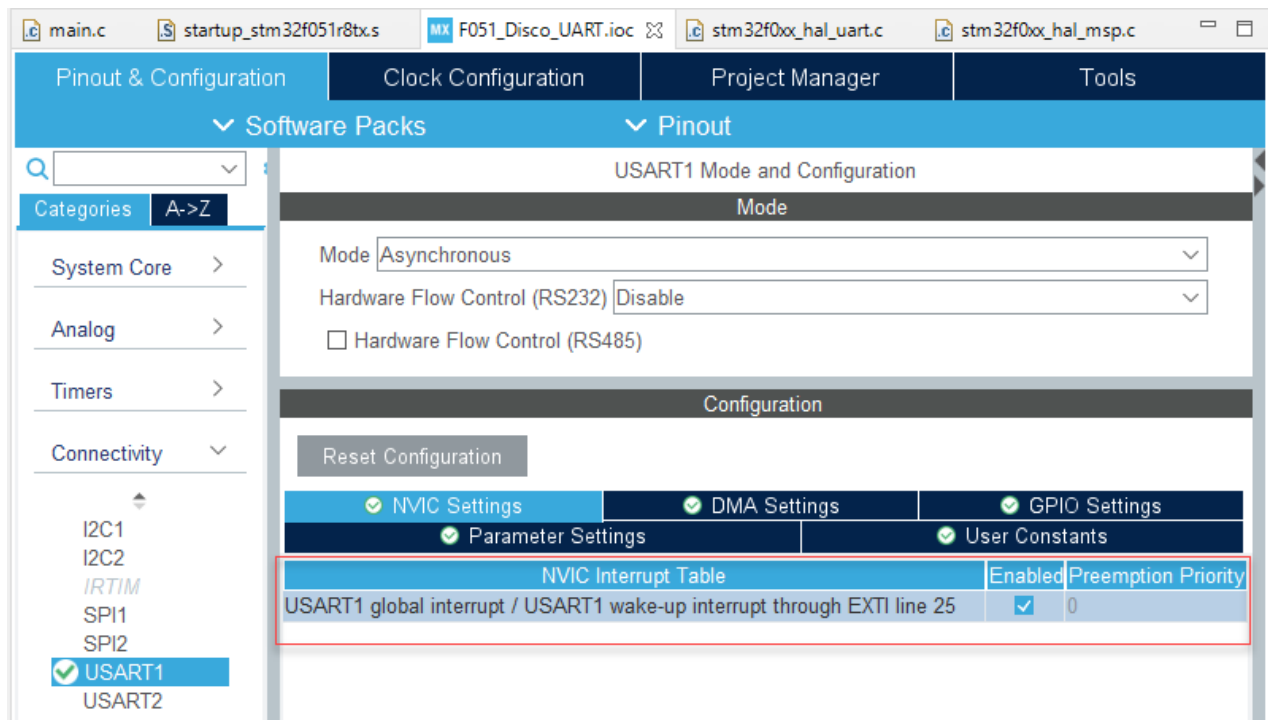
- For data transmission, place a byte inside the `USART->DR` register and wait until the *Transmit Data Register Empty* (TXE) flag is asserted true.
- For data reception, wait until the *Received Data Ready to be Read* (RXNE) is not asserted true, and then store the content of the `USART->DR` register inside the application memory.

The difference between the two methods consists in how they wait for the completion of data transmission:

- In polling mode, the `HAL_UART_Receive()` / `HAL_UART_Transmit()` functions are designed so that it waits for the corresponding event flag to be set, for every byte of data.
- In interrupt mode, the function `HAL_UART_Receive_IT()` / `HAL_UART_Transmit_IT()` are designed so that they do not wait for data transmission completion, but the dirty job to place a new byte inside the DR register, or to load its content inside the application memory, is accomplished by the ISR routine when the `RXNEIE` / `TXEIE` interrupt is generated.

7.2. Enable interrupt

Go to USART1 module, select *NVIC Settings* and enable the interrupt.



Enable interrupt for USART1

After generating code, the functions to enable interrupt are written in function `HAL_UART_MspInit()` in `stm32f0xx_hal_msp.c` file:

```
HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(USART1_IRQn);
```

The interrupt handler is added to `stm32f0xx_it.c` file too. Trace the function `HAL_UART_IRQHandler()` to understand about how it processes the data. Basically, it checks the error, check the state, and mode of the USART instance; then it save or transfer data on RX or TX wire.

```
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart1);
}
```

7.3. Handler interrupt

Send data

Finally, use `HAL_UART_Transmit_IT()` function to send data.

```
while(1) {
    if (pause == 0) {
        sprintf(buffer, "counter=%03d\n\r", counter++);
```

```

        HAL_UART_Transmit_IT(&huart1, (uint8_t *)buffer, strlen(buffer));
    }
    HAL_Delay(1000);
}

```

⚠ Race condition in Interrupt Mode

Consider below code:

```

void printWelcomeMessage(void) {
    HAL_UART_Transmit_IT(&huart1, buffer1, COUNTOF(buffer1));
    HAL_UART_Transmit_IT(&huart1, buffer2, COUNTOF(buffer2));
    HAL_UART_Transmit_IT(&huart1, buffer3, COUNTOF(buffer3));
}

```

The above code will never work correctly, since each call to the function `HAL_UART_Transmit_IT()` is much faster than the UART transmission, and the subsequent calls to the `HAL_UART_Transmit_IT()` will fail.

If speed is not a strict requirement for the application, and the use of the `HAL_UART_Transmit_IT()` is limited to few parts of the application, the above code could be rearranged in the following way:

```

void printWelcomeMessage(void) {
    char *strings[] = {buffer1, buffer2, buffer3};
    for (uint8_t i = 0; i < 3; i++) {
        HAL_UART_Transmit_IT(&huart1, strings[i], COUNTOF(strings[i]));
        while (HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX ||
               HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX_RX);
    }
}

```

Receive data

Next step is to read data using interrupt. Because it's unknown time when a character comes, so the buffer for receiving will be filled in at anytime, even when buffer is being used in *printf()*, therefore, should use a new buffer to store received data, e.g. `cmd[]`.

When the receiver get enough characters, it will fire an interrupt, so the interrupt can be used to start handling data. HAL has a weak callback `HAL_UART_RxCpltCallback()`, so it can be overridden to turn on a flag which indicates that data is received successfully:

```

char rx_int = 0;
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart == &huart1) {
        rx_int = 1;
    }
}


```

```

    }
}

```

In the main function, process received data only when the flag is on:

 Add delay after calling `HAL_UART_Transmit_IT()` to make sure the transmission will be complete.

```

const char msg_rx_int[] = "\tRX INT\n\r";

int main(void)
{
    char cmd[6] = {0};

    while(1) {
        if (pause == 0) {
            sprintf(buffer, "I:counter=%03d\n\r", counter++);
            HAL_UART_Transmit_IT(&huart1, (uint8_t *)buffer, strlen(buffer));
        }

        HAL_Delay(50); // Must have delay for transmission to complete

        /* use buffer may not work, because sprintf in main loop may be filling
        the buffer */
        ret = HAL_UART_Receive_IT(&huart1, (uint8_t *)cmd, 6);

        if (ret == HAL_OK) {
            HAL_UART_Transmit(&huart1, msg_ok, strlen(msg_ok), HAL_MAX_DELAY);
        } else if (ret == HAL_BUSY) {
            HAL_UART_Transmit(&huart1, msg_busy, strlen(msg_busy),
            HAL_MAX_DELAY);
        } else if (ret == HAL_ERROR) {
            HAL_UART_Transmit(&huart1, msg_error, strlen(msg_error),
            HAL_MAX_DELAY);
        } else if (ret == HAL_TIMEOUT) {
            HAL_UART_Transmit(&huart1, msg_timeout, strlen(msg_timeout),
            HAL_MAX_DELAY);
        }

        if (rx_int == 1) {
            rx_int = 0;
            HAL_UART_Transmit(&huart1, msg_rx_int, strlen(msg_rx_int),
            HAL_MAX_DELAY);
            HAL_UART_Transmit(&huart1, cmd, strlen(cmd), HAL_MAX_DELAY);
            if (strncmp(cmd, "stop", 4) == 0) {
                pause = 1;
            } else if (strncmp(cmd, "resume", 6) == 0) {
                pause = 0;
            }
        }
    }
}

```

```

        HAL_Delay(500);
    }
}

```

Build and run application code, it should work much better than the Polling version, because whenever a character is sent to the board, it will be received correctly.

```

I:counter= 0          pause = 0
    OK
I:counter= 1
    BUSY
I:counter= 2
    BUSY
I:counter= 3
    BUSY
I:counter= 4
    BUSY
I:counter= 5          pause = 1
    RX INT
stoppp OK
    BUSY
    BUSY
    BUSY
    BUSY
    BUSY
    RX INT          pause = 0
resumeI:counter= 6
    OK
I:counter= 7
    BUSY

```

Communicate with UART in interrupt mode

Bug: Input length is fixed

The above implementation has an issue: The receiving interrupt only is fired when it receives enough number of characters. In the above example, enter *stopxx* for stop command will work, but *stop* will never do.

To fix this, set the receive mode to get only one byte at a time, then check for the *newline* `\n` or *carriage return* `\r` character to determine input sentences. However, this will lead to run the interrupt handler many times.

8. Appendix

Windows 10 does not support PL2303 USB to Serial, but here is the fix for this problem:

<https://github.com/johnstevenson/pl2303-win10>. This will install an old but compatible driver for EOL PL2303 chips.