

Universal Asynchronous Receiver-Transmitter

Almost every microcontroller provides at least one UART peripheral, and this protocol seems to be a must-have supported feature for any project as UART is widely used for debugging, logging, data exchange, and for firmware update. Serialization also be used in many application, especially in continuous streaming, mostly working in DMA mode, some in Interrupt mode, and rarely in Polling mode.

[#arm](#) [#stm32](#) [#usart](#) [#uart](#) [#interrupt](#) [#dma](#)

Last update: 2021-06-12 23:16:18

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. STM32Cube HAL Usage

3. Lab 1: Polling mode

- 3.1. Start a new project
- 3.2. Enable USART1
- 3.3. Generated code
- 3.4. Send data
- 3.5. Connect UART to PC
- 3.6. Receive data

4. Lab 2: Interrupt Mode

- 4.1. Interruptions
- 4.2. Start a new project
- 4.3. Enable interrupt
- 4.4. Send data with interrupt
- 4.5. Receive data with interrupt

5. Lab 3: DMA mode

- 5.1. Start a new project
- 5.2. Enable DMA
- 5.3. Send data with DMA
- 5.4. Receive data with DMA
- 5.5. UART IDLE Detection
- 5.6. Process continuous received data

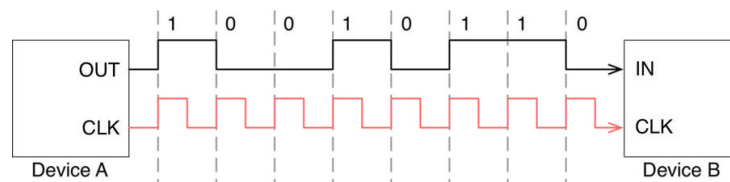
1. Hardware

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

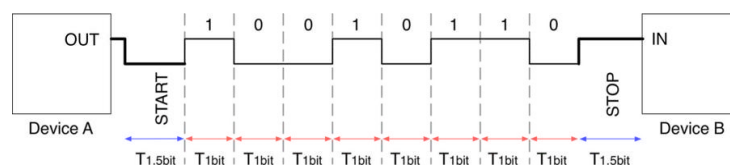
When the information flows between two devices inside a common channel, both devices (as the sender and also the receiver) have to agree on the *timing*, that defines how long it takes to transmit each individual bit of the information.

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



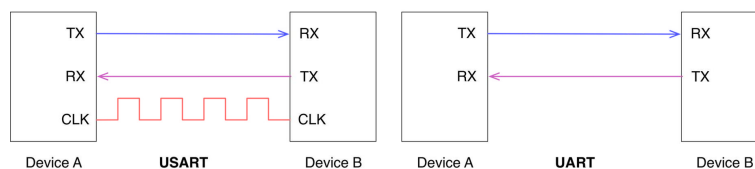
Shared clock in *synchronous* USART

- 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.

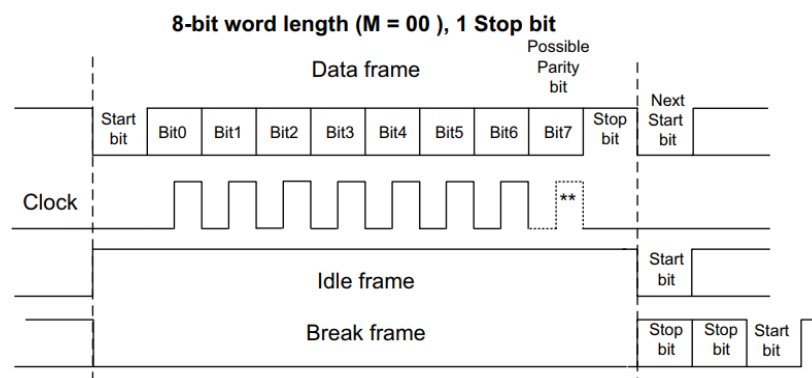
1.3. Data frame

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
- A 0.5, 1, 1.5, or 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.



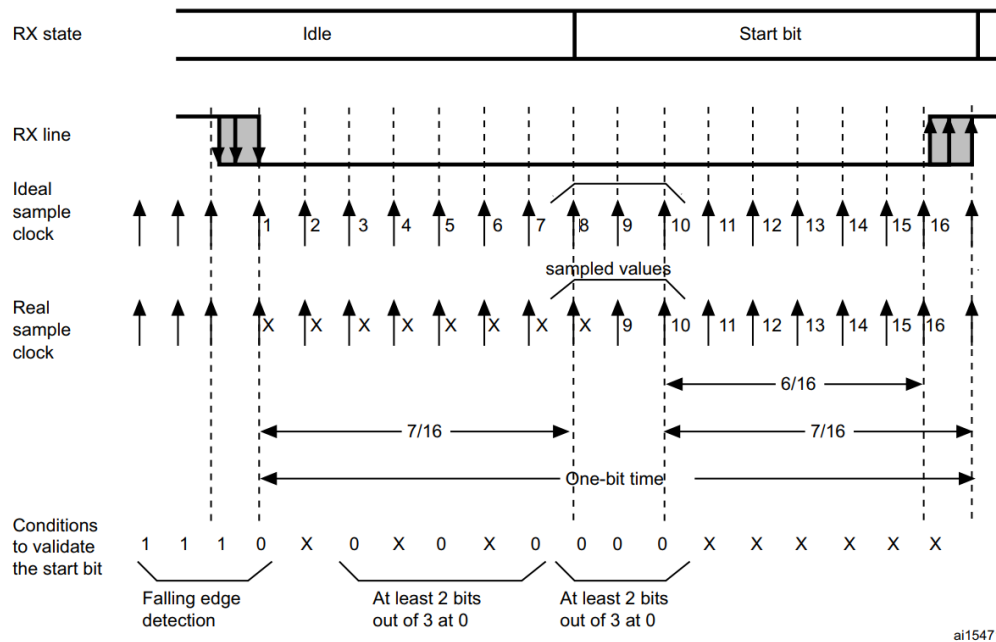
Frames in USART

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 core clock frequency at 8 MHz.

- In case of oversampling by 16:
 - BRR** = **USARTDIV** = $8\,000\,000/9600 = 833d = 0341h$

- In case of oversampling by 8:
 $\text{USARTDIV} = 2 * 8\,000\,000 / 9600 = 1666,66 (\sim 1667d) = 683h$
 $\text{BRR}[3:0] = 3h \gg 1 = 1h$
 $\text{BRR} = 0x681$

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. STM32Cube HAL Usage

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.

excerpt from [Description of STM32F0 HAL and low-layer drivers](#)

How to use USART HAL

1. Declare a `UART_HandleTypeDef` handle structure (eg. `UART_HandleTypeDef huart`).

2. Initialize the UART low level resources by implementing the `HAL_UART_MspInit()` API when needed:

- Enable the USARTx interface clock.
- UART pins configuration:
 - Enable the clock for the UART GPIOs.
 - Configure these UART pins as alternate function pull-up.
- NVIC configuration if use interrupt process (`HAL_UART_Transmit_IT()` and `HAL_UART_Receive_IT()` APIs):
 - Configure the USARTx interrupt priority.
 - Enable the NVIC USART IRQ handle.
- UART 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.

3. Lab 1: Polling mode

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

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.

Requirements:

- Increase a counter by 1 and print its value to UART1 every second
- Get user commands:
 - **stop** to pause increasing the counter
 - **resume** to resume increasing the counter

Target board:

Any board which has STM32 MCUs. This tutorial will be using the STM32F0 Discovery board, which features an STM32F051R8 Cortex-M0 MCU.

| STM32F051R8 | Mode | External peripheral |
|-------------|--------------------|---------------------|
| PA9 | Alternate Function | UART1 TX |
| PA10 | Alternate Function | UART1 RX |
| PC9 | GPIO Output | Green LED |

3.1. Start a new project

Open STM32CubeIDE and create a new STM32 with STM32F051R8 MCU by selecting the target board or just the target MCU.

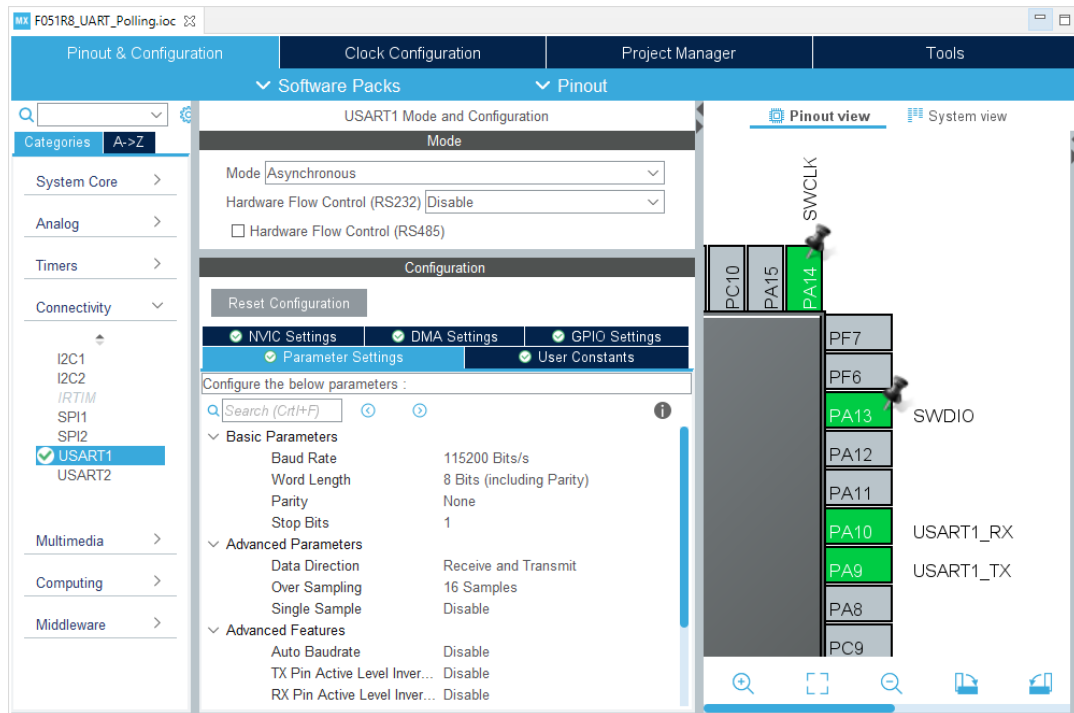
Make sure to configure below settings:

- Set the HCLK to *48 MHz*
- Set the Debug mode to Debug Serial Wire or Trace Asynchronous SW

3.2. 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 bits (including Parity)*
 - Parity: *None*
 - Stop bits: *1*



Enable USART1

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

3.3. Generated code

When generate code from 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.HSIState = 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()` inits 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);
    }
}

```

3.4. Send data

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

```

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

char counter = 0;
char buffer[16] = {0}; // counter=xxx\n\r

```

```
int main(void) {
    while (1)
    {
        counter++;
        sprintf(buffer, "counter=%03d\n\r", counter);
        HAL_UART_Transmit(&huart1, (uint8_t *)buffer, strlen(buffer),
                           HAL_MAX_DELAY);

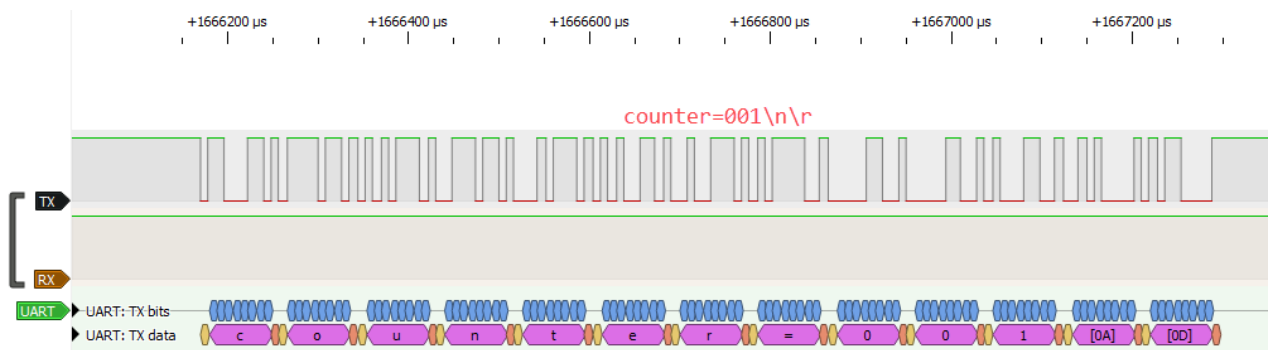
        HAL_Delay(1000);
    }
}
```

3.5. Connect UART to PC

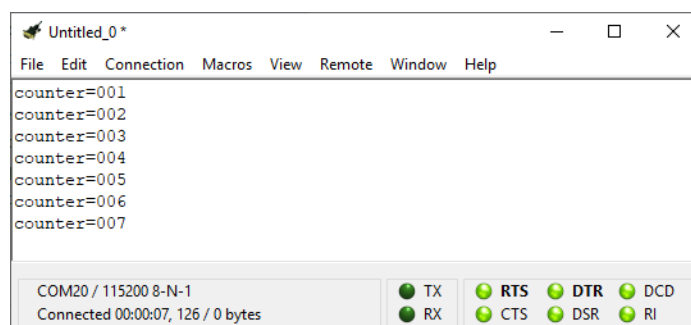
Because STM32F0 Discovery does not have a Virtual COM port on ST-LINK/V2, so use a TTL-to-USB converter go get UART data. Connect pins **PA9** and **PA10** to UART terminal on PC. It's recommend to check the voltage because MCU board is running at 3.3V while PC USB or COM port might be running at 5V.

Another option is to use an Arduino Uno board with RESET pin connected to GND, and use its TX, RX pins which are connected to the Arduino Virtual COM port.

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



UART output in a terminal

3.6. Receive data

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

Use the function `HAL_UART_Receive(&huart1, (uint8_t *)buffer, 4, 2000)` to read the input, which means:

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

Let's modify the code to get helper functions and process input in main while loop:

```
char counter = 0;
char buffer[16] = { 0 }; // counter=xxx\n\r
const char MSG_PAUSE[] = "PAUSED\n\r";
const char MSG_RESUME[] = "RESUMED\n\r";
const char MSG_OK[] = "OK\n\r";
const char MSG_BUSY[] = "BUSY\n\r";
const char MSG_ERROR[] = "ERROR\n\r";
const char MSG_TIMEOUT[] = "TIMEOUT\n\r";

HAL_StatusTypeDef Write(const char *buffer) {
    return HAL_UART_Transmit(&huart1, (uint8_t*) buffer, strlen(buffer),
                             HAL_MAX_DELAY);
}

HAL_StatusTypeDef Read(char *buffer, int n) {
    HAL_StatusTypeDef ret = HAL_TIMEOUT;

    ret = HAL_UART_Receive(&huart1, (uint8_t*) buffer, n, 2000);
    if (ret == HAL_OK) {
        Write(MSG_OK);
    } else if (ret == HAL_BUSY) {
        Write(MSG_BUSY);
    } else if (ret == HAL_ERROR) {
        Write(MSG_ERROR);
    } else if (ret == HAL_TIMEOUT) {
        Write(MSG_TIMEOUT);
    }

    return ret;
}
```

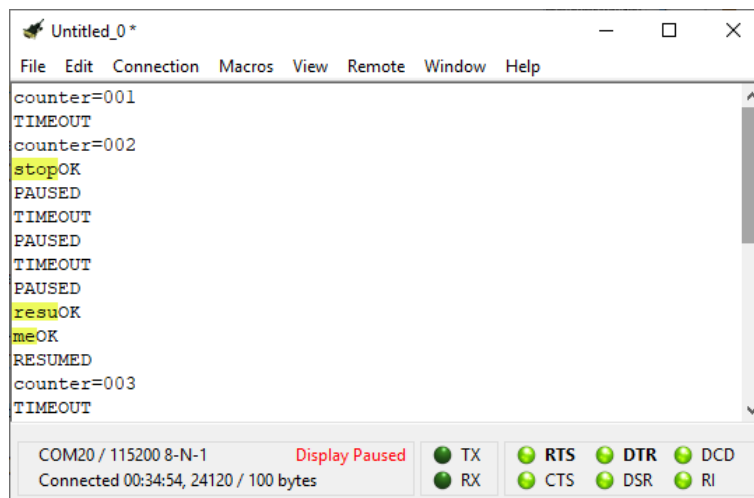
```

int main(void) {
    char pause = 0;

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

        Read(buffer, 4);
        if (strncmp(buffer, "stop", 4) == 0) {
            pause = 1;
            Write(MSG_PAUSE);
        } else if (strncmp(buffer, "resu", 4) == 0) {
            Read(buffer, 2);
            if (strncmp(buffer, "me", 2) == 0) {
                pause = 0;
                Write(MSG_RESUME);
            }
        }
    }
}

```



Receive user's input

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).

4. Lab 2: Interrupt Mode

4.1. Interruptions

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 |
| 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 that job. Then user is warned about the interrupt thanks to a series of callback functions invoked by the `HAL_UART_IRQHandler()`.

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 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:

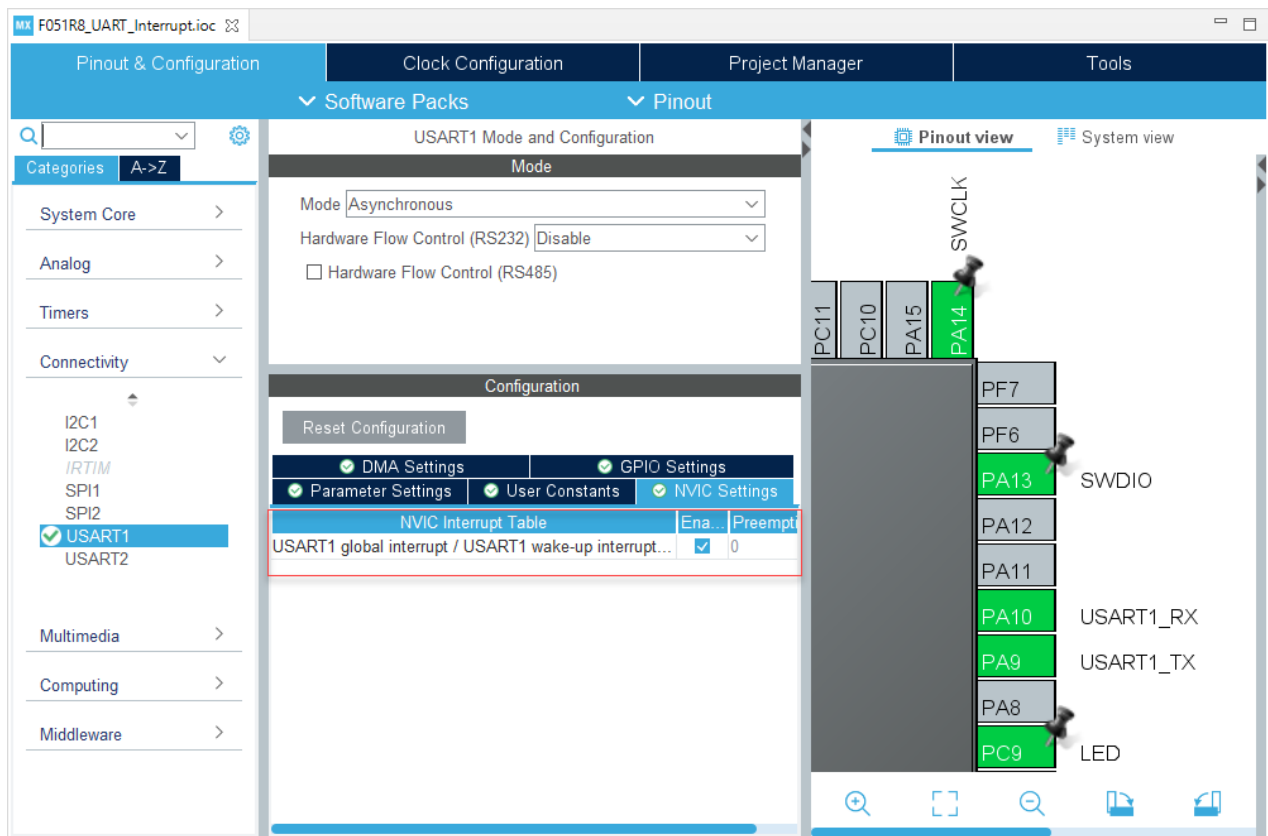
- In polling mode, the `HAL_UART_Receive()` / `HAL_UART_Transmit()` functions are designed so that it waits for the corresponding event flag to be set, for every byte of data.
- In interrupt mode, the `HAL_UART_Receive_IT()` / `HAL_UART_Transmit_IT()` functions are designed so that they do not wait for data transmission completion, but the 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.

4.2. Start a new project

Open STM32CubeIDE and create a new STM32 with the same steps in the previous lab, including configuration for clocks, debug, and UART1.

4.3. Enable interrupt

Go to USART1 module, select **NVIC Settings** tab 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);
}
```

4.4. Send data with interrupt

As said above, use `HAL_UART_Transmit_IT()` function to send data.

main.c

```
char counter = 0;
char buffer[16] = { 0 }; // counter=xxx\n\r

HAL_StatusTypeDef Write(const char *buffer) {
    return HAL_UART_Transmit_IT(&huart1, (uint8_t*) buffer, strlen(buffer));
}

int main() {
    char pause = 0;

    while(1) {
        if (pause == 0) {
            sprintf(buffer, "counter=%03d\n\r", counter++);
            Write(buffer);
        }
        HAL_Delay(1000);
    }
}
```

Race condition in Interrupt Mode

Consider below code:

```
void printWelcomeMessage(void) {
    HAL_UART_Transmit_IT(&huart1, buffer1, strlen(buffer1));
    HAL_UART_Transmit_IT(&huart1, buffer2, strlen(buffer2));
    HAL_UART_Transmit_IT(&huart1, buffer3, strlen(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 as it will see that UART is in the Busy state.

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], strlen(strings[i]));
        while (HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX ||
               HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX_RX);
    }
}
```

When all data in the buffer are sent, HAL library will call to a callback function named `HAL_UART_TxCpltCallback()` to notify about the end of the transmission. /* There is no callback when half of data is transferred */. This function can be overridden to do something after the buffer is transmitted.

main.c

```
char uart_tx_done = 0;

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
    if(huart == &huart1) {
        uart_tx_done = 1;
    }
}
```

4.5. Receive data with interrupt

Next step is to read data using interrupt with the function `HAL_UART_Receive_IT()`. 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 the `sprintf()` function, therefore, should use a new buffer to store received data, e.g. `command`.

When the receiver get enough characters, it will fire an interrupt to run the `HAL_UART_RxCpltCallback()` function. That function can be overridden to handle received data in the main:

```
char uart_rx_int = 0;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart == &huart1) {
        uart_rx_int = 1;
    }
}
```

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

```
char uart_rx_int = 0;
char command[16] = { 0 };
const char MSG_PAUSE[] = "PAUSED\n\r";
const char MSG_RESUME[] = "RESUMED\n\r";

HAL_StatusTypeDef Read(char *buffer, int n) {
    return HAL_UART_Receive_IT(&huart1, (uint8_t*) buffer, n);
}

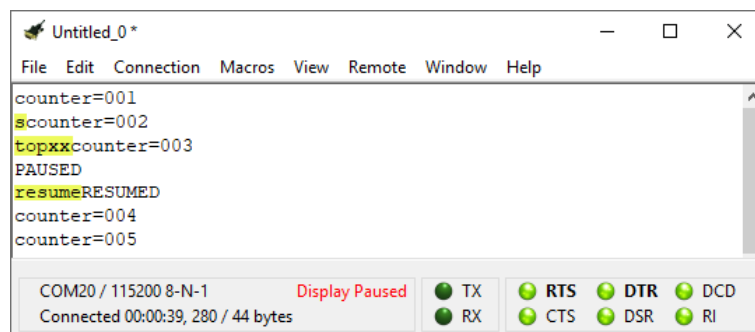
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart == &huart1) { uart_rx_int = 1; }
}

int main() {
    char pause = 0;

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

        if (uart_rx_int == 1) {
            uart_rx_int = 0;
            if (strcmp(command, "stop", 4) == 0) {
                pause = 1;
                Write(MSG_PAUSE);
            } else if (strcmp(command, "resume", 6) == 0) {
                pause = 0;
                Write(MSG_RESUME);
            }
        }

        Read(command, 6);
        HAL_Delay(1000);
    }
}
```



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 *new line / line feed* `\n` or *carriage return* `\r` character to determine input sentences. However, this will lead to run the interrupt handler many times if the incoming data rate is high.

Here is an example to handle every byte in the Interrupt mode:

- Received one byte at a time
- Check the received byte with the *new line* `\n` character to separate strings
- Call Receive function again to listen to a new character

```
char rx_buffer[16] = { 0 };
int rx_idx = 0;
char rx = 0;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (rx == '\n') {
        rx_buffer[rx_idx]='\0';
        strncpy(command, rx_buffer, 16);
        rx_idx = 0;
        uart_rx_int = 1;
    } else {
        rx_buffer[rx_idx++]=rx;
    }
    Read(&rx, 1);
}

int main(void) {
    Read(&rx, 1);
    while (1) {
        if (uart_rx_int == 1) {
            uart_rx_int = 0;
            if (strcmp(command, "stop", 4) == 0) {...}
        }
    }
}
```

This approach can be optimized more by not using the HAL function (both `HAL_UART_Receive_IT()` and `HAL_UART_IRQHandler()`), but by setting up `UART_IT_RXNE` (Received Data Not Empty) interrupt and then handle this interrupt manually in the `USART1_IRQHandler()` ISR.

```
__HAL_UART_ENABLE_IT(&huart1, UART_IT_RXNE);
void USART1_IRQHandler(void) { handle_received_byte(); }
```

5. Lab 3: DMA mode

The **DMA** can be used to transfer data in or out through an UART interface. However, DMA still needs to know how many bytes of data should be exchanged. In case of transmitting, it is easy to calculate the length of data, but in case of receiving, it maybe unknown length of data.


5.1. Start a new project


Open STM32CubeIDE and create a new STM32 with the same steps in the previous lab, including configuration for clocks, debug, and UART1.

5.2. Enable DMA

Go to USART1 module, select **DMA Settings** tab and Add two DMA requests:

- **USART1_TX**: This DMA Request has direction of Memory to Peripheral, it means DMA processor will read data from Memory and write to the USART1 Transmit Data Register, therefore, only Memory Address will be increased
- **USART1_RX**: This DMA Request has direction of Peripheral to Memory, it means DMA processor will read data from USART1 Receive Data Register and write to Memory, therefore, only Memory Address will be increased

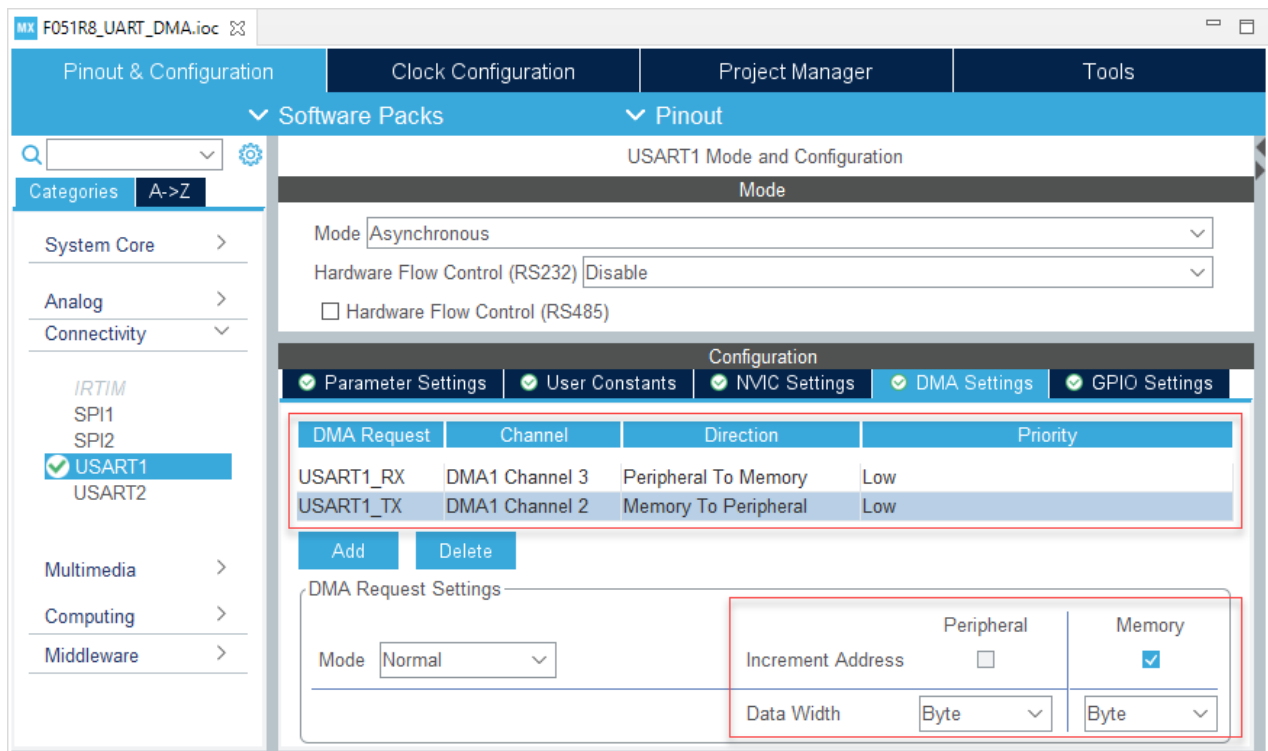
 DMA only works when the peripheral's interrupt is enabled, as it needs triggered from the peripheral. Make sure to enable USART1 global / wake-up interrupts through the external interrupt line.

 Note that CubeMX automatically enable DMA interrupts. To disable it, go to **NVIC** module under the **System Core** category.

After generating code, there is a new function **MX_DMA_Init()** added to the **main.c** file to initialize the DMA module:

```
static void MX_DMA_Init(void)
{
    /* DMA controller clock enable */
    __HAL_RCC_DMA1_CLK_ENABLE();

    /* DMA interrupt init */
    /* DMA1_Channel2_3_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(DMA1_Channel2_3_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(DMA1_Channel2_3_IRQn);
}
```



Enable DMA requests on USART1

Inside the `HAL_UART_MspInit()` function, there are calls to init DMA instances for USART1_TX and USART1_RX, and finally links of DMA instances with the USART instance with the function `__HAL_LINKDMA`.

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) {
    ...
    /* DMA USART1_RX Init */
    hdma_usart1_rx.Instance = DMA1_Channel3;
    hdma_usart1_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
    hdma_usart1_rx.Init.PeriphInc = DMA_PINC_DISABLE;
    hdma_usart1_rx.Init.MemInc = DMA_MINC_ENABLE;
    hdma_usart1_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    hdma_usart1_rx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
    hdma_usart1_rx.Init.Mode = DMA_NORMAL;
    hdma_usart1_rx.Init.Priority = DMA_PRIORITY_LOW;
    if (HAL_DMA_Init(&hdma_usart1_rx) != HAL_OK) {
        Error_Handler();
    }
    __HAL_LINKDMA(huart, hdmarx, hdma_usart1_rx);

    /* DMA USART1_TX Init */
    hdma_usart1_tx.Instance = DMA1_Channel2;
    hdma_usart1_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
    hdma_usart1_tx.Init.PeriphInc = DMA_PINC_DISABLE;
    hdma_usart1_tx.Init.MemInc = DMA_MINC_ENABLE;
    hdma_usart1_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    hdma_usart1_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
```

```

    hdma_usart1_tx.Init.Mode = DMA_NORMAL;
    hdma_usart1_tx.Init.Priority = DMA_PRIORITY_LOW;
    if (HAL_DMA_Init(&hdma_usart1_tx) != HAL_OK) {
        Error_Handler();
    }
    __HAL_LINKDMA(huart, hdmatx, hdma_usart1_tx);

    /* USART1 interrupt Init */
    HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(USART1_IRQn);
}

```

There is also an implementation for DMA interrupt in the file `stm32f0xx_it.c` which calls to the `HAL_DMA_IRQHandler()` function.

There is no default callback for DMA. User has to set the callbacks manually, as described in the [DMA Interrupts](#) section. However, HAL functions will assign a callback when they need to handle an interrupt.

5.3. Send data with DMA

Sending data with DMA is quite easy, just provide the `tx_buffer` and the length of data to the function `HAL_UART_Transmit_DMA()`.

```

#define DMA_BUFFER_MAX 16
char tx_buffer[DMA_BUFFER_MAX] = { 0 };

HAL_StatusTypeDef Write(const char *buffer) {
    return HAL_UART_Transmit_DMA(&huart1, (uint8_t*) buffer, strlen(buffer));
}

int main() {
    char pause = 0;

    while(1) {
        if (pause == 0) {
            sprintf(tx_buffer, "counter=%03d\n\r", counter++);
            Write(tx_buffer);
        }
        HAL_Delay(1000);
    }
}

```

There are two interrupts will be fired for transmission, which are helpful when sending a huge amount of data: DMA notifies application to start loading new data into the first half of the buffer after the `TxCpltCallback` is fired, while the second half of the buffer is being transmitted by the DMA; then application can load the new data into the second half of the buffer after the `TxCpltCallback` is fired, while the first half is being transmitted.

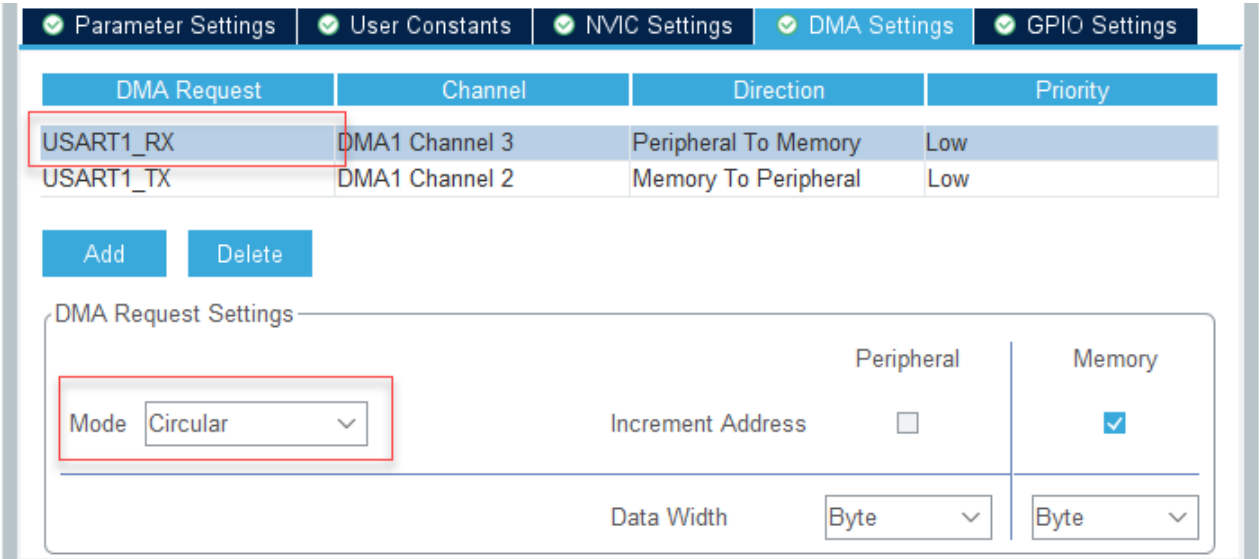
Using DMA mode can significantly reduce the number of UART interrupts, comparing to using Interrupt mode. If there is a variable to keep track of the numbers of UART interrupts, for example in the below code, this number will be increased for every bytes in the transmitting buffer in Interrupt mode (each byte sent causes an interrupt to load the next byte). However, in DMA mode, there is only one interrupt added only when the buffer is completely transferred.

```
size_t uart_irq_counter = 0;
void USART1_IRQHandler(void) {
    uart_irq_counter++;
}
```

5.4. Receive data with DMA

Normally, when calling to `HAL_UART_Receive_DMA()`, the DMA module will stop transferring data when it counts enough bytes set in the params. There is the *Circular* mode that makes DMA continuous get data and fills into memory like a ring buffer.

Go back to the **DMA Settings** tab of the USART1 module to set the mode of USART1_RX request to Circular.



| DMA Request | Channel | Direction | Priority |
|-------------|----------------|----------------------|----------|
| USART1_RX | DMA1 Channel 3 | Peripheral To Memory | Low |
| USART1_TX | DMA1 Channel 2 | Memory To Peripheral | Low |

Buttons: Add, Delete

DMA Request Settings

| | | Peripheral | Memory |
|-------------------|----------|--------------------------|-------------------------------------|
| Mode | Circular | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Increment Address | | <input type="checkbox"/> | |
| Data Width | Byte | | Byte |

Enable Circular mode for DMA on USART1_RX

Then create a buffer for receiving data, and start DMA Request in the main function:

```
#define DMA_BUFFER_MAX 16
char rx_buffer[DMA_BUFFER_MAX] = { 0 };

int main(void) {
    HAL_UART_Receive_DMA(&huart1, (uint8_t*)rx_buffer, DMA_BUFFER_MAX);
    while (1) {...}
}
```


By calling the `HAL_UART_Receive_DMA()` function, USART1_RX DMA instance will get some callback functions assigned to its Half-transfer and Full-Transfer callbacks. Refer to the function `UART_Start_Receive_DMA()` for more details.

It is possible to configure DMA to transfer one byte at a time and call to an interrupt function to handle the newly received character like in Interrupt mode, but it wastes of resource and performance. However, if application lets DMA to notify only then it receives enough the required number of characters, sometimes there is no notification sent.

” Assume that application expects to receive 20 chars, but UART only receives 14 chars:

- Application would be notified when 10 bytes received by Half-Transfer event
- Application would never be notified the rest of 4 bytes has arrived
 - If UART get more chars, application would be notified by Full-Transfer event but some chars may be left over

5.5. UART IDLE Detection

Most of STM32 series have USARTs with *IDLE Line* detection. If IDLE Line detection is not available, some of them have *Receiver Timeout* feature with programmable delay. IDLE line detection (or Receiver Timeout) can trigger USART interrupt when receive line is steady without any communication for at least 1 character for reception.

IDLE Detection

Right after enable the UART, the **IDLE** bit in Interrupt Status Register **ISR** will be set. However, the interrupt for IDLE detection only gets fired when the bit **IDLEIE** is set on the Control Register **CRx**. Therefore, the IDLE interrupt always fire up once right after the bit **IDLEIE** is set.

To clear the IDLE interrupt status, either set the **IDLECF** in the Clear Register or do a read sequence on **SR** and **DR** registers. Refer to the [Reference Manual](#) documents to get more detail.

When IDLE detection is enable, make sure the RX line is not floating to prevent the IDLE flag from being set continuously.

DMA RX and IDLE Line detection

The good combination for using DMA to get unknown length of data is to use DMA in Circular mode, with big enough memory buffer, then use DMA Half-Transfer, Full-Transfer and the IDLE line detection to notify application to process received data. HAL DMA Receiving function automatically notifies the application by calling `HAL_UART_RxCpltCallback()` and `HAL_UART_RxHalfCpltCallback()`. Therefore, it is only needed to override the `USART1_IRQHandler()` function. To more simple and still left HAL Handler processes other cases, it is better to add a small code just to check the IDLE flag and process received data before handing over the interrupt to original `HAL_UART_IRQHandler()` function.

stm32f0xx_it.c

```

void USART1_IRQHandler(void)
{
    if (((&huart1)->Instance->ISR & UART_FLAG_IDLE) != 0
        && ((&huart1)->Instance->CR1 & USART_CR1_IDLEIE) != 0) {
        __HAL_UART_CLEAR_IDLEFLAG(&huart1);
        HAL_UART_RxCpltCallback(&huart1);
    }
    // pass the work to HAL function
    HAL_UART_IRQHandler(&huart1);
}

```

main.c

```

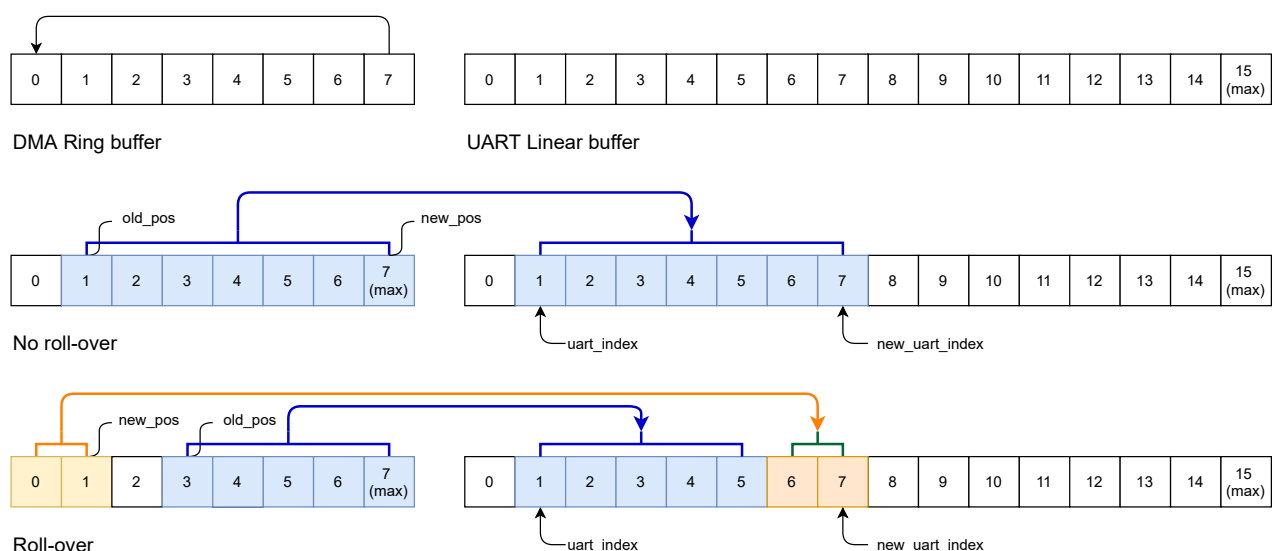
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    UART_RX_Check(&hdma_usart1_rx);
}

void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart) {
    UART_RX_Check(&hdma_usart1_rx);
}

```

5.6. Process continuous received data

Due to DMA buffer is a Ring buffer with small number of bytes, it is necessary to copy received bytes from DMA buffer to a bigger UART buffer. There are 2 indexes to mark the bytes position in DMA buffer: **new_pos** at the last received byte, and **old_pos** at last processed byte. There is one index **uart_index** to mark the last byte in the UART buffer.



Visualization of memory copy between DMA buffer and UART buffer

- In DMA buffer, if the **new_pos** is bigger than the **old_pos**, it means there is no roll-over, application can copy bytes [**old_pos** : **new_pos**] to the UART buffer.

- In DMA buffer, if the **new_pos** is less than the **old_pos**, it means there is a roll-over, application can copy bytes [**old_pos** : **max**] and [0: **new_pos**] to the UART buffer.
- In UART buffer, when buffer is full, it does not get any more character, only character *new line* **\n** will reset the index to 0
- To get the **new_pos** index, aka. number of received bytes, use the register **CNDTR** of the DMA instance. Read more about “DMA channel x number of data register (**DMA_CNDTRx** and **DMA2_CNDTRx**)” section in the [Reference Manual](#) document.

Here is the implementation example for the above solution of processing the received data:

```
#define UART_BUFFER_MAX    64
char uart_buffer[UART_BUFFER_MAX] = { 0 };
size_t uart_buffer_idx = 0;
char uart_new_string = 0;

void UART_RX_Process(const void *data, size_t len) {
    for (int i = 0; i < len; i++) {
        char c = ((char*) data)[i];
        if (uart_buffer_idx < UART_BUFFER_MAX-2) {
            uart_buffer[uart_buffer_idx++] = c;
            uart_buffer[uart_buffer_idx] = '\0';
        }
        if (c == '\n') {
            uart_buffer_idx = 0;
            uart_new_string = 1;
        }
    }
}

void UART_RX_Check(DMA_HandleTypeDef *hdma) {
    static size_t old_pos = 0;
    size_t rx_pos = DMA_BUFFER_MAX - hdma->Instance->CNDTR;
    if (rx_pos != old_pos) { // new data
        if (rx_pos > old_pos) { // no overflow
            UART_RX_Process(&uart_buffer[old_pos], rx_pos - old_pos);
        } else { // overflow
            UART_RX_Process(&uart_buffer[old_pos], DMA_BUFFER_MAX - old_pos);
            if (rx_pos > 0) { // run up
                UART_RX_Process(&uart_buffer[old_pos], rx_pos);
            }
        }
        old_pos = rx_pos;
    }
}

int main(void) {
    while (1) {
        if (uart_new_string == 1) {
            uart_new_string = 0;
            if (strcmp(uart_buffer, "stop", 4) == 0) {...}
        }
    }
}
```

```

    }
}

```

This tutorial only show a method to process variable string length in a continuous byte stream. The function `UART_RX_Process()` should be modified to handle different streaming format.

Compile and run with variable string length to find how it work. Below captured image was in a debug section to see how many IDLE interrupts are called, how received bytes are saved into the `rx_buffer` and to monitor the `uart_buffer`.

The screenshot shows a debugger interface with the following components:

- Main Window:** Displays a list of received characters: `abcdefghijklmnopqrstuvwxyz`. Below this, a counter is shown incrementing from 120 to 133. The status bar at the bottom indicates the device is paused and shows UART control line states (TX, RX, RTS, CTS, DTR, DSR, DCD, RI).
- Live Expressions Window:** A table showing the contents of the `rx_buffer` array. The table has three columns: Expression, Type, and Value.

| Expression | Type | Value |
|---------------------------------|------------------------|--|
| <code>uart_irq_counter</code> | <code>size_t</code> | 246 |
| <code>uart_idle_counter</code> | <code>size_t</code> | 44 |
| <code>rx_buffer</code> | <code>char [16]</code> | [16] |
| <code>rx_buffer[0]</code> | <code>char</code> | 113 'q' |
| <code>rx_buffer[1]</code> | <code>char</code> | 114 'r' |
| <code>rx_buffer[2]</code> | <code>char</code> | 115 's' |
| <code>rx_buffer[3]</code> | <code>char</code> | 116 't' |
| <code>rx_buffer[4]</code> | <code>char</code> | 117 'u' |
| <code>rx_buffer[5]</code> | <code>char</code> | 118 'v' |
| <code>rx_buffer[6]</code> | <code>char</code> | 119 'w' |
| <code>rx_buffer[7]</code> | <code>char</code> | 120 'x' |
| <code>rx_buffer[8]</code> | <code>char</code> | 121 'y' |
| <code>rx_buffer[9]</code> | <code>char</code> | 122 'z' |
| <code>rx_buffer[10]</code> | <code>char</code> | 13 '\r' |
| <code>rx_buffer[11]</code> | <code>char</code> | 10 '\n' |
| <code>rx_buffer[12]</code> | <code>char</code> | 109 'm' |
| <code>rx_buffer[13]</code> | <code>char</code> | 110 'n' |
| <code>rx_buffer[14]</code> | <code>char</code> | 111 'o' |
| <code>rx_buffer[15]</code> | <code>char</code> | 112 'p' |
| <code>uart_buffer_idx</code> | <code>size_t</code> | 0 |
| <code>(char*)uart_buffer</code> | <code>char *</code> | 0x2000000b0 <uart_buffer> "abcdefghijklmnopqrstuvwxyz\r\n" |
| <code>uart_new_string</code> | <code>char</code> | 0 '\0' |

Communicate with UART in DMA mode