How To Use STM32’s UART

Module 1: Creating convenient library for transmitting and receiving data using STM32’s UART

Introduction

Through this module, you'll learn how to use STM32's UART the right way. **Everything you've seen before is garbage!** Just trash it… We will create a serial library which is fast, reliable and can be easily reused in different projects.

Multi-port, powered by **HAL** and compatible with any STM32 microcontroller series.

Table of contents

There are many STM32 UART **myths** created by incompetent people. E.g., that DMA can’t be used with **variable-size data** packets and other nonsense.

Avoiding DMA peripheral is one of the worst things you can do. It is much simpler to make really good UART code when **DMA** is used.

Lesson A: The right way

Feature 1: Reliability

Never lose a byte! Many will say that their UART code is reliable, good, etc. Do you believe it?

We will perform two simple tests to ensure that our serial library is reliable:

* **Loopback test** to ensure that each byte is sent and received correctly.
* **Hot-plug test** to ensure that communication can be resumed after cable is re-connected.

Feature 2: Simplicity

Sending and receiving should be simple. We need only these functions:

* Function to send and receive **single byte**
* Function to send and receive **data arrays** (this includes sending/receiving of 16-bit and 32-bit words)
* Function to check if new data **available**
* Function to check if transfer is **finished**
* Some other functions will be provided in next modules

Our serial library **must** be portable across multiple STM32 series. Generally, we should consider portability across microcontrollers from different manufactures.

E.g., porting code to Microchip's **PIC32** would be complicated if our library heavily relies on STM32's "idle" interrupt. Thus, we know what functions to avoid if we want our library to be more portable.

Feature 3: Portability

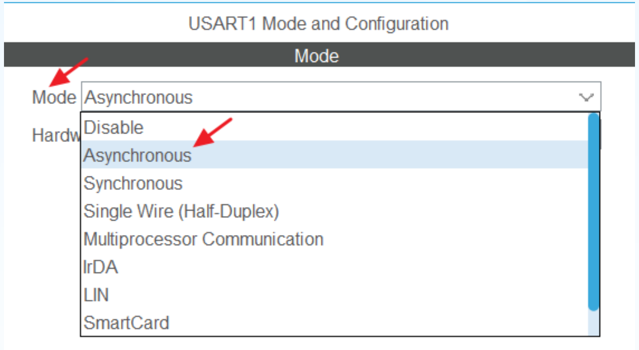
We need to be able to send or receive streams of data without **blocking** other peripherals, such as DAC, ADC, SPI, etc.. That's why data should be sent or received in **background** using **DMA**.

Feature 4: Efficiency

First, we will **enable all** available serial ports and their transmit and receive **DMA** requests. Optionally, you can set different Baud Rate under “Parameter Settings” tab (default is 115200 Bits/s).

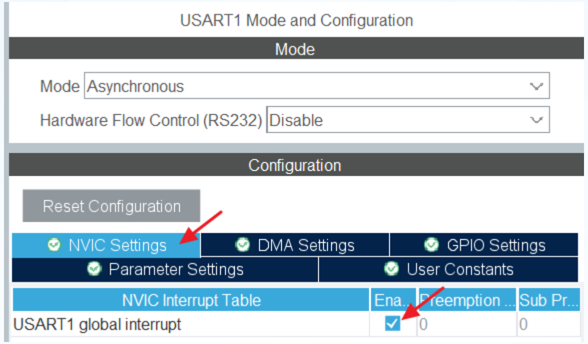
Lesson B: Configuring UART ports

Open “Pinout & Configuration” tab → Connectivity → USART1 → Set **“Mode”** to **“Asynchrous”**. Do the same for USART2 and USART3.



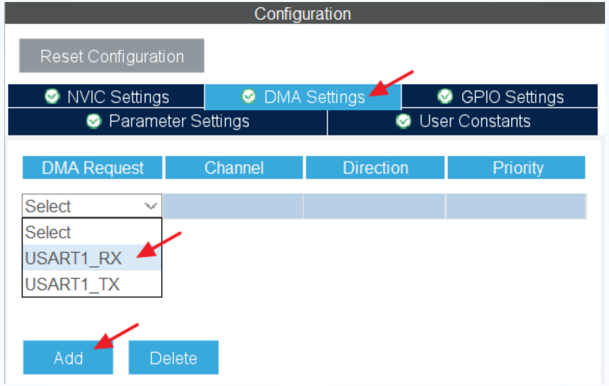
Step 1: Enable UART peripheral

Open USART1 → **“NVIC Settings”** tab → check **“USART1 global interrupt”**. Use the same way to enable “global interrupt” for USART2 and USART3.



Step 2: Enable interrupt

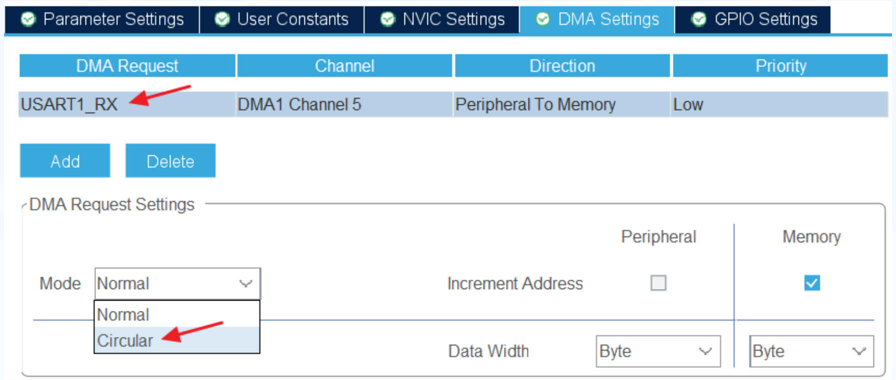
Open USART1 → **”DMA Settings”** tab → click **”Add”** to add a new line. Select **“USART1_RX”** from drop-down list. Repeat the same for USART2 and USART3.



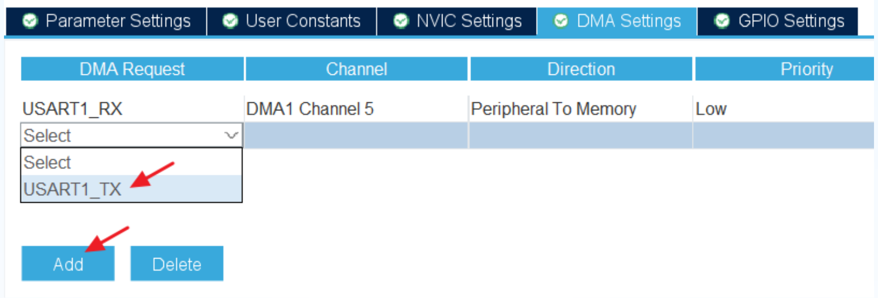
Step 3: Enable receive DMA request

Step 3 (continued)

With USART1_RX DMA Request selected, set **"Mode"** to **"Circular"** under "DMA Request Settings". Repeat the same for USART2 and USART3.



Open USART1 → **"DMA Settings"** tab → click **"Add"** to add a new line. Select **"USART1_TX"** from drop-down list. Repeat the same for USART2 and USART3. Keep "Mode" normal for TX.



Step 4: Enable transmit DMA request

In previous lesson, we've configured three UART ports and their DMA requests. STM32CubeIDE can generate all required code to initialize and work with these ports using HAL functions.

In this lesson, we will learn how to **receive** and **transmit** UART data using HAL DMA functions. We'll evaluate how **circular** DMA buffer works and start to build our serial library.

Lesson C: Simple receive and transmit experiment

We start by defining very small 4-element buffer in main.c:

```
/* USER CODE BEGIN PV */
#define UART_BUFFER_SIZE 4
__IO uint8_t uartBuffer[UART_BUFFER_SIZE];
```

And start receiving process using DMA:

```
/* USER CODE BEGIN 2 */
memset((uint8_t *) &uartBuffer, 0, sizeof(uartBuffer)); // Clear buffer
HAL_UART_Receive_DMA(&huart1,                            // HAL UART handle
                     (uint8_t *) &uartBuffer,            // RX buffer pointer
                     UART_BUFFER_SIZE);                  // RX buffer size
```

Step 1: Start receiving

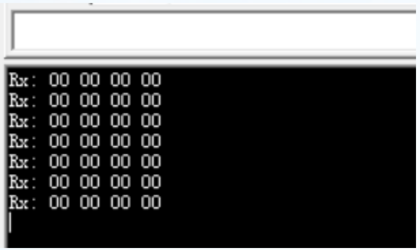
For simplicity, we will send **uartBuffer[]** back each second in main() function loop:

```
/* USER CODE BEGIN WHILE */
while (1)
{
    HAL_UART_Transmit_DMA(&huart1,                          // HAL UART handle
                          (uint8_t *)&uartBuffer,   // TX buffer pointer
                          UART_BUFFER_SIZE);        // TX buffer size
    HAL_Delay(1000); // 1000 millisecond delay
…
```

Note that we use same array for transmit and receive.

Step 2: Transmitting test

Microcontroller sends contents of 4-byte long array uartBuffer to UART1 each second. That’s how terminal window will look after 7 seconds:



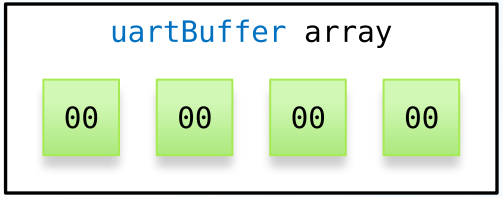
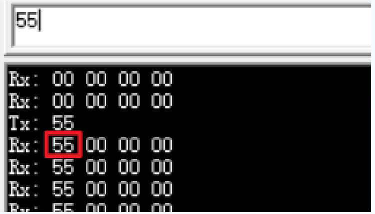


Image on the right illustrates memory contents of uartBuffer array.

Step 3: Testing in terminal program

Let's send single byte to STM32 board. In this example, we will send **0x55**. Right after we sent this byte ("Tx: 55" line), received data changed to "Rx: 55 00 00 00" instead of "Rx: 00 00 00 00".



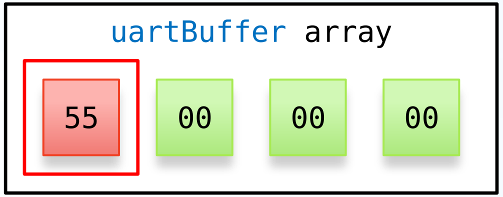
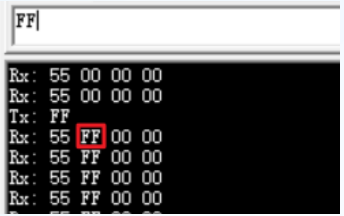


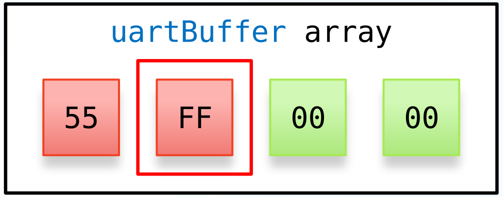
Image on the right demonstrates changes made to uartBuffer.

Step 3 (continued)

Step 3 (continued)

Let's send one more byte. This time we will send **0xFF**. Images below shows that memory still contains previously received byte and 0xFF right next to it:

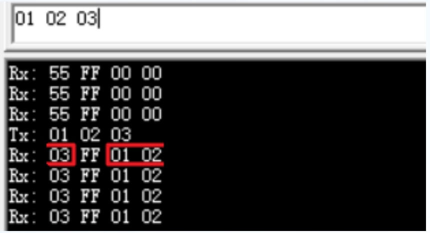


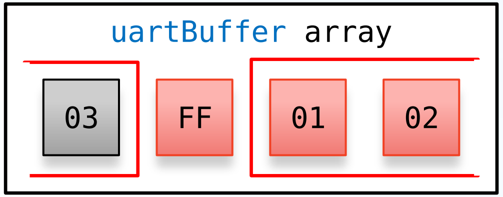


Right image show that our array still has two "free" elements.

Step 3 (continued)

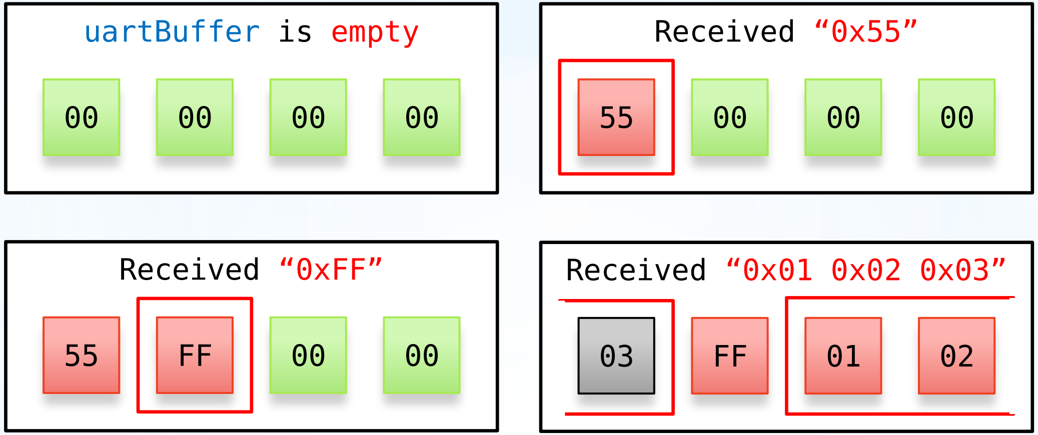
Now, let's send three bytes: **0x01, 0x02 and 0x03.** There are only two "free" elements left, but because we use Circular Mode, DMA wraps around and continues to write from the beginning:





Note how **0x55** byte was overwritten by **0x03** (gray square).

Empty buffer → Received “0x55” → “0xFF” → “0x01 0x02 0x03”



Step 3 (continued)

Here is what should be noted:

* We started receiving by a single call to HAL_UART_Receive_DMA(). There is **no need to restart DMA**, because all received bytes will arrive to uartBuffer array.
* There is **no need to know arriving data size**. We only need a receiving buffer that is big enough to accept largest packet.
* We need to copy or process received data **before it's overwritten** by newly arriving bytes.
* We need to **keep track** of receiving **DMA write pointer**.

Experiment results

You've probably already noticed that receiving data using DMA's circular buffer is **much simpler** than with interrupts. It's because all arriving bytes are **automatically** placed to receiving array by DMA.

Now we need to create algorithm that extracts unprocessed bytes from circular buffer and ignores processed bytes.

Lesson D: Receiving single byte

Step 1: Getting receiving DMA current position

We can find receiving DMA current position from **NDTR** “number of data” register. This register indicates how many bytes are **not yet transferred** by DMA buffer.

Right after reset, NDTR is equal to **4** (value of UART_BUFFER_SIZE from **previous lesson**). NDTR decreases by one for each arriving byte, e.g. becomes equal to **3, 2, 1** and resets back to **4** after last element is written. Formula for calculating current position is:

```
Current DMA position = UART_BUFFER_SIZE - NDTR
```

Step 1 (continued)

If NDTR is 4, DMA will place arriving byte in uartBuffer[0] and decrease NDTR by one.

If NDTR is 3, DMA will place arriving byte in uartBuffer[1] and decrease NDTR by one.

If NDTR is 2, DMA will place arriving byte in uartBuffer[2] and decrease NDTR by one.

If NDTR is 1, DMA will place arriving byte in uartBuffer[3] and resets NDTR value back to 4 (NDTR value minimal value is 1)

Step 1 (continued)

In other words, for 4-element array NDTR changes like this:

4 → 3 → 2 → 1 → 4 → 3 → 2 → 1 → 4 → 3 → ...

Arrived bytes will be written to the following array elements:

0 → 1 → 2 → 3 → 0 → 1 → 2 → 3 → 0 → 1 → ...

Code for calculating DMA position (**dmaPos**) would be:

```
#define UART_BUFFER_SIZE 4
…
dmaPos = UART_BUFFER_SIZE - huart1.hdmarx->Instance->CNDTR;
```

We need to define two variables.

```
uint16_t dmaPos;  // DMA's current RX position (receive buffer index)
uint16_t rxPos=0; // Our's current RX position (receive buffer index)
```

* First variable is dmaPos. It's an array index at which DMA will place next arriving byte. This variable value is calculated using formula from previous step.
* Second variable is rxPos. If this variable is not equal to dmaPos, this will indicate that one or more bytes arrived since our last check and should be processed.

Step 2: Check for unprocessed bytes in receiving array

Step 2 (continued)

With these two variables, extracting separate bytes from circular array can be done using following code:

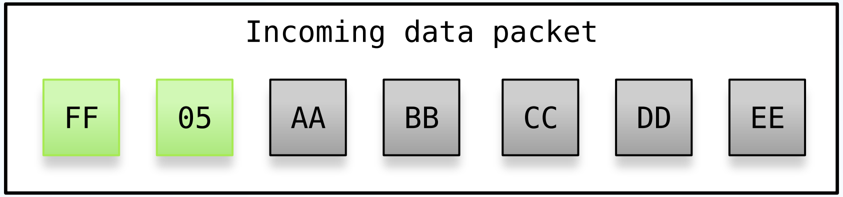
```
// Get DMA current position
dmaPos=UART_BUFFER_SIZE-huart1.hdmarx->Instance->CNDTR;

if (dmaPos!=rxPos) // DMA position differs from last processing position
{
    uint8_t rxByte=uartBuffer[rxPos];     // Get single byte
    rxPos++;                              // Increase processing position
    if (rxPos==UART_BUFFER_SIZE) rxPos=0; // Reached end of array
}
```

Note that this code should be called repeatedly in main loop.

Step 2 (continued)

Main purpose of this algorithm is to extract and process single bytes from **data headers** or **small data packets**. E.g., we can use this approach to process two first bytes of following data packet:



where **0xFF** is some command and **0x05** is data length. Following **five** bytes **0xAA, 0xBB, 0xCC, 0xDD and 0xEE** can be processed using more efficient approach with copying 5 bytes at once.

Step 3: Simple loopback test

Let's check if data is received correctly by sending it back to the computer:

```
// Get DMA current position
dmaPos=UART_BUFFER_SIZE-huart1.hdmarx->Instance->CNDTR;

if (dmaPos!=rxPos) // DMA position differs from last processing position
{
    uint8_t rxByte=uartBuffer[rxPos];      // Get single byte
    rxPos++;                               // Increase processing position
    if (rxPos==UART_BUFFER_SIZE) rxPos=0; // Reached end of array

    // Sending received byte back to computer
    HAL_UART_Transmit_DMA(&huart1, (uint8_t *)&rxByte, 1);
}
```

Step 3 (continued)

If you test this code, you will notice that some bytes are not sent back. E.g., if you send **“0x01 0x02 0x03”,** microcontroller only answers with **“0x01 0x03”,** and **“0x02”** byte is lost. That’s happens when HAL_UART_Transmit_DMA is called before previous transmitting DMA transfer is complete.

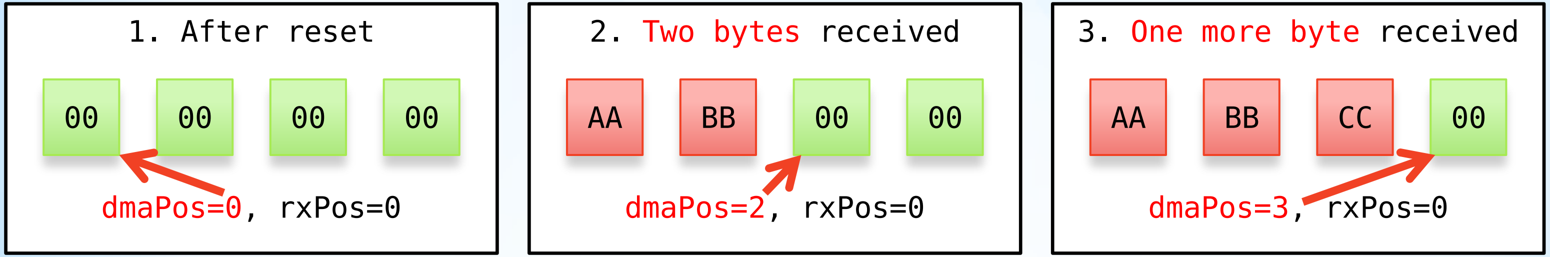
In the next lesson, we will fix this issue by adding check for DMA transfer completion.

All incoming bytes are automatically placed to circular buffer by DMA peripheral. Our algorithm needs to extract those bytes from circular buffer and keep track of dmaPos and rxPos values.

Receiving multiple bytes requires us to deal with **split** data packets. Splitting occures when data packet crosses boundary of **circular buffer**.

Lesson E: Calculating available unprocessed bytes count

Before we start with length algorithm, let's examine how circular buffer is filled with data and how indexes change over time:



As you can see, dmaPos points to array element that is going to be written next. rxPos will be altered later by our processing code.

Step 1: Receiving indexes

Calculation should be done for two different cases. First case is when received data packet is **not** split by circular buffer boundary. In this case available data length is calculated by subtracting our "processing" position from DMA position index:

```
uint16_t dataLen=0;

if (dmaPos>rxPos) // If DMA position is greater than processing position
{
    dataLen=dmaPos-rxPos; // difference gives unprocessed data length
}
```

Step 2: Calculating unprocessed data length

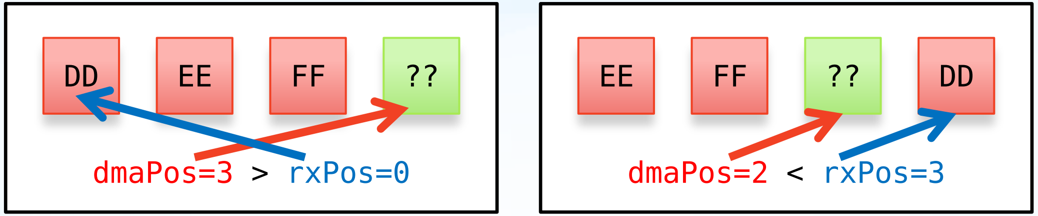
Second case is when received data packet is **split** between **ending** and **beginning** of circular buffer. In this case available data length calculated different way:

```
uint16_t dataLen=0;

if (dmaPos<rxPos) // If DMA position is less than processing position
{
    dataLen=UART_BUFFER_SIZE-rxPos; // calculating "tail" length
    dataLen+=dmaPos;                // adding "head" length
}
```

Step 2 (continued)

Following images illustrate how 3-byte data packet **"DD EE FF"** will be written to DMA receive buffer in first and second case:



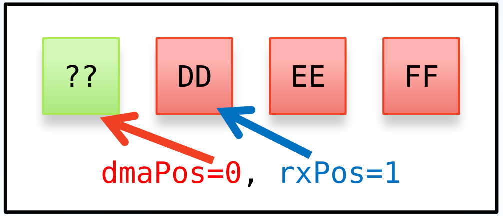
In the second case (right) 3-byte data packet is split by 4-byte circular array boundary. And data length calculation for this case:

```
// dataLen = UART_BUFFER_SIZE – rxPos + dmaPos = 4 – 3 + 2 = 3
```

Step 2 (continued)

Step 2 (continued)

There is a single case when data packet is considered **split** when **it's not**. This confusing case results from the fact that our indexes indicate elements that's **going to be** written or processed next.



In this case, **dmaPos=0** and "head" length is **zero**, so calculated data length is still correct and equals to 3:

```
// dataLen = UART_BUFFER_SIZE – 1 + dmaPos = 4 – 1 + 0 = 3
```

In previous step, we calculated amount of unprocessed data length which is available in circular buffer. Our receiving function will

* **Check** if there is enough unprocessed bytes in circular array.
* **Copy** required amount to destination array.
* **Update** rxPos variable.

First, we need to add temporary buffer for processed data and variable which tells how much bytes we need to process:

```
__IO uint8_t destBuffer[UART_BUFFER_SIZE]; // temporary buffer
uint16_t needLen=3;                        // need to receive three bytes
```

Lesson F: Receiving multiple bytes

Step 1: Receiving normal data packet

As with available data length calculation, there are two cases for receiving process. If data is not split (normal data packet):

```
if (dataLen>=needLen) // do we have enough amount of unprocessed bytes?
{
    if(rxPos+needLen-1<UART_BUFFER_SIZE) // if data packet is not split
    {
        memmove((uint8_t *)&destBuffer[0], // destination
        (uint8_t *)&uartBuffer[rxPos],     // source
        needLen);                          // length
    }
    rxPos+=needLen; // update our buffer "processing" position
    if (rxPos>=UART_BUFFER_SIZE) { rxPos-=UART_BUFFER_SIZE;}
}
```

In second case, packet is restored from two separate pieces:

```
if(rxPos+needLen-1<UART_BUFFER_SIZE) // if data packet is not split
{
    memmove((uint8_t *)&destBuffer[0],(uint8_t *)&uartBuffer[rxPos],needLen);
}
else // if data packet is split (second case)
{
    uint16_t tailLen=UART_BUFFER_SIZE-rxPos; // “tail” length calculation
    uint16_t headLen=needLen-tailLen;        // “head” length calculation
    // Copying “tail” and “head” parts separately:
    memmove((uint8_t *)&destBuffer[0],(uint8_t *)&uartBuffer[rxPos],tailLen);
    memmove((uint8_t *)&destBuffer[tailLen],(uint8_t )&uartBuffer[0],headLen);
}
```

Step 2: Receiving split data packet

Transmitting code is much simpler than receiving code. In most cases, we do not need a separate transmit buffer and can send variables and arrays directly by passing their pointers to HAL_UART_Transmit_DMA() function.

The only thing we need to ensure is this data is not altered by our program before transmission is completed.

Lesson G: Transmitting single or multiple bytes

We can check if transmission is complete by reading USART_SR_TC bit of UART status register:

```
// Sending received byte back to computer
HAL_UART_Transmit_DMA(&huart1, (uint8_t *)&rxByte, 1);

// Wait for current transmission complete bit in UART status register
while (READ_BIT(huart1.Instance->SR,USART_SR_TC)==0) {asm("nop");}
```

Note that transmission complete check while() loop can be placed before HAL_UART_Transmit_DMA() to achieve non-blocking operation. In such cases we must guarantee that rxByte remains unchanged before transfer is complete.

Step 1: Ensure that UART transfer is complete

Step 1 (continued)

Here is non-blocking example:

```
// Wait until previous transmission is complete
while (READ_BIT(huart1.Instance->SR,USART_SR_TC)==0) {asm("nop");}

// Sending received byte back to computer
HAL_UART_Transmit_DMA(&huart1, (uint8_t *)&rxByte, 1);

// We shouldn't change rxByte value below,
// because it is uncertain if DMA transfer is finished
```

We will implement both approaches in our UART library.

Step 2: Transmitting multiple bytes

Transmitting multiple bytes is very similar:

```
HAL_UART_Transmit_DMA(&huart1,                   // HAL UART handle
                      (uint8_t *)&someArray,  // TX data pointer
                      sizeof(someArray));     // TX data size
```

If data transmission **fails**, it is usually caused by placing transmitting data (rxByte or someArray[]) in **non-DMA** memory, a memory area which can’t be accessed by DMA peripheral.

Step 3: Simple loopback test

After required amount of bytes is detected and copied to temporary destBuffer[] array, we can send this data back to the computer using following code:

```
if (dataLen>=needLen) // do we have enough data?
{
    // send temporary buffer to computer
    HAL_UART_Transmit_DMA(&huart1, (uint8_t *)&destBuffer, needLen);

    // wait for transfer completion
    while (READ_BIT(huart1.Instance->SR,USART_SR_TC)==0) {asm("nop");}
}
```

In previous lessons, we've created **proof of concept** code for transmitting and receiving data over UART.

Now we are going to transform this code into small library with user-friendly set of functions. Remember, we want to make fast and simple **multi-port** library which works on **any STM32 microcontroller** series.

Lesson H: Creating UART library and it's functions

Step 1: Port data structure

Create **uart.h** file and define COM_TPort structure:

```
#include "stm32f1xx_hal.h" // For accessing stm32 HAL data structures
#include "stdbool.h"       // For boolean type support
#include "string.h"        // For memset() function

typedef struct
{
    UART_HandleTypeDef* huartPtr; // Pointer to HAL's UART port
    volatile uint32_t* ndtrPtr;   // Pointer to NDTR register
    uint8_t* rxBufPtr;            // Pointer to Receive buffer
    uint16_t rxBufSize;           // Size of Receive buffer
    uint16_t rxPos;               // Current position for processing
} COM_TPort;
```

Add array for storing our ports and function for port selection:

```
// --- uart.h ---
#define COM_MAX_NUM 8

// --- uart.c ---
COM_TPort* myPortPtr;          // pointer to current port
COM_TPort myPort[COM_MAX_NUM]; // port array

void COM_Select(uint8_t n)
{
    // Current port pointer will help us to
    // avoid code bloating with indexes like myPort[…]
    myPortPtr=&myPort[n];
}
```

Step 2: Array of ports and port selection function

Step 3: Port initialization function

Initialization function fills structures similar to previous lesson:

```
void COM_Init(uint8_t n,                              // Port number 0 to COM_MAX_NUM-1
                  UART_HandleTypeDef *huartPtr, // Pointer to HAL's UART port
                  void *rxBufPtr,               // Pointer to receive buffer
                  uint16_t rxBufSize)           // Receive buffer size
{
    COM_Select(n);                  // Select myPort[n] as current port
    myPortPtr->huartPtr = huartPtr; // Save HAL's UART pointer and NDTR to myPort[n]
    myPortPtr->ndtrPtr= &(myPortPtr->huartPtr->hdmarx->Instance->CNDTR);

    myPortPtr->rxBufPtr= rxBufPtr;    // Save receive buffer pointer to myPort[n]
    myPortPtr->rxBufSize = rxBufSize; // Save receive buffer size to myPort[n]
    myPortPtr->rxPos = 0;             // Initialize processing position to zero

    memset((uint8_t *) rxBufPtr, (char) 0, rxBufSize); // Clear receive buffer (optional)
    HAL_UART_Receive_DMA(huartPtr, (uint8_t *) rxBufPtr, rxBufSize); // Start DMA
}
```

Step 4: Waiting for UART transfer complete function

This function waits for UART transfer completion:

```
void COM_WaitTxDone(void)
{
    while (READ_BIT(myPortPtr->huartPtr->Instance->SR,USART_SR_TC)==0)
    {
        asm("nop");
    }
}
```

It's based on code we've created in "Lesson G". It blocks execution until USART_SR_TC bit is set.

Step 5: Function for writing bytes (blocking)

This function writes arbitrary number of bytes (blocking mode):

```
void COM_Write(void *pData, uint16_t Size)
{
    COM_WaitTxDone(); // ensure previous transfer complete
    HAL_UART_Transmit_DMA(myPortPtr->huartPtr, pData,Size);
    COM_WaitTxDone(); // wait till transfer completion
}
```

Function sends Size bytes from *pData memory location and blocks execution until USART_SR_TC bit is set.

This function writes arbitrary number of bytes (non-blocking):

```
void COM_WriteFast(void *pData, uint16_t Size)
{
    COM_WaitTxDone(); // wait for previous transfer is complete
    HAL_UART_Transmit_DMA(myPortPtr->huartPtr, pData,Size);
}
```

Main difference between COM_Write() and COM_WriteFast() is that **Fast** version exits immediately after HAL_UART_Transmit_DMA() execution without waiting for transfer completion. We'll choose function type depending on application.

Step 6: Function for writing bytes (non-blocking)

This function is based on code from “Lesson D”:

```
bool COM_ComReadByte(uint8_t *b)
{
    uint16_t dmaPos=myPortPtr->rxBufSize-*myPortPtr->ndtrPtr;
    if (dmaPos!=myPortPtr->rxPos)
    {
        *b=(myPortPtr->rxBufPtr)[myPortPtr->rxPos];
        myPortPtr->rxPos++;
        if (myPortPtr->rxPos==myPortPtr->rxBufSize) {myPortPtr->rxPos=0;}
        return true; // byte was successfully read to “b” variable
    }
    return false; // byte was not read (no new data in rxBuf)
}
```

Step 7: Function for reading single byte

Step 8: Function for reading multiple bytes

This function is based on code from “Lesson F”:

```
bool COM_Read(uint8_t *destBuffer, uint16_t needLen)
{
    uint16_t UART_BUFFER_SIZE=myPortPtr->rxBufSize; // To keep code changes minimal
    uint16_t rxPos=myPortPtr->rxPos;                // compared to “Lesson F”

    // Available data length calculation for normal and split case:
    uint16_t dmaPos=UART_BUFFER_SIZE-*myPortPtr->ndtrPtr;
    uint16_t dataLen=0;
    if (dmaPos>rxPos) { dataLen=dmaPos-rxPos; }
    if (dmaPos<rxPos)
    {
        dataLen=UART_BUFFER_SIZE-rxPos;
        dataLen+=dmaPos;
    }

// …continued on next page
```

Step 8 (continued)

Two cases:

```
// …check previous page
if (dataLen>=needLen) // if we have enough data in circular array
{
    if(rxPos+needLen-1<UART_BUFFER_SIZE) // copy data for normal case
    {
        memmove((uint8_t *)&destBuffer[0],(uint8_t *)&(myPortPtr->rxBufPtr)[rxPos],needLen);
    }
    else // copy in two steps if data is split (second case)
    {
        uint16_t tailLen=UART_BUFFER_SIZE-rxPos;
        uint16_t headLen=needLen-tailLen;
        memmove((uint8_t *)&destBuffer[0],(uint8_t *)&(myPortPtr->rxBufPtr)[rxPos],tailLen);
        memmove((uint8_t *)&destBuffer[tailLen],(uint8_t *)&(myPortPtr->rxBufPtr)[0],headLen);
    }
// …continued on next page
```

Step 8 (continued)

This function is based on code from "Lesson F":

```
// check previous page
        rxPos+=needLen; // increment our processing position by read amount
        if (rxPos>=UART_BUFFER_SIZE) { rxPos-=UART_BUFFER_SIZE;}
        myPortPtr->rxPos=rxPos; // save processing position to current port (one of myPort[n])
        return true;            // data copied successfully
    }
    return false; // error, not enough data in circular buffer
}
```

We kept code changes minimal, e.g. replaced #define by variable with same name and created rxPos to avoid long lines of text:

```
    uint16_t UART_BUFFER_SIZE=myPortPtr->rxBufSize; // New can reuse code from "Lesson F"
    uint16_t rxPos=myPortPtr->rxPos;                // Avoided repeating of myPortPtr->rxPos
```

Step 9: Adding function headers

Library can be used after adding function headers to uart.h file:

```
extern void COM_Select(uint8_t n);                          // Set current port

extern void COM_Init(uint8_t n,                             // Initialize port n
                     UART_HandleTypeDef *huartPtr,          // HAL’s UART handler
                     void *rxBufPtr,                        // RX buffer pointer
                     uint16_t rxBufSize);                   // RX buffer size

extern void COM_Write(void *pData, uint16_t Size);          // Blocking write
extern void COM_WriteFast(void *pData, uint16_t Size);      // Non-blocking write
extern bool COM_ReadByte(uint8_t *b);                       // Read single byte
extern bool COM_Read(uint8_t *destBuffer, uint16_t needLen); // Read bytes
```

Lesson I: Additional features

Our UART library is almost ready to be used! To make it's even **better**, we are going to add some **more features**.

In this lesson, you will learn:

* How to add support for **more** microcontroller series.
* How to add **timeout** for read operations and simple way to keep track of errors.
* How to disable **framing error** checks and why it is important.
* How to add support for **RS-485** transfers.

Step 1: Extending port data structure with new fields

Adding fields for reading timeouts/errors, and RS-485 support:

```
#include "stm32f1xx_hal.h" // For accessing stm32 HAL data structures
#include "stdbool.h"       // For boolean type support
#include "string.h"        // For memset() function

typedef struct
{
    UART_HandleTypeDef* huartPtr; // Pointer to HAL's UART port
    volatile uint32_t* ndtrPtr;   // Pointer to NDTR register
    uint8_t* rxBufPtr;            // Pointer to Receive buffer
    uint16_t rxBufSize;           // Size of Receive buffer
    uint16_t rxPos;               // Current position for processing
    uint16_t timeOut;             // New: Timeout for read operations
    bool rxFail;                  // New: Receive failure flag
    void (*_rs485tx) (void);      // New: Switch to TX mode callback (for RS-485)
    void (*_rs485rx) (void);      // New: Switch to RX mode callback (for RS-485)
} COM_TPort;
```

Step 2: Adding support for other STM32 microcontroller series

STM32 microcontroller series have minor differences in some structure **names**. Correct names can be substituted using defines:

```
// Select STM32 microcontroller family from F1, F4 and H7 by uncommenting on of lines.
#define STM32F1
//#define STM32F4
//#define STM32H7

#ifdef STM32F1
#include "stm32f1xx_hal.h"
#define MY_ISR(p) p->huartPtr->Instance->SR
#define MY_ISR_TC USART_SR_TC
#define NDTR_PTR(p) &(p->huartPtr->hdmarx->Instance->CNDTR) // No “C” letter on H7 series
#endif

// …continued on next page
```

STM32F4 is the same as STM32F1, but STM32H7 is different:

```
#ifdef STM32F4 // …see previous page for STM32F1 defines
#include "stm32f4xx_hal.h"
#define MY_ISR(p) p->huartPtr->Instance->SR
#define MY_ISR_TC USART_SR_TC
#define NDTR_PTR(p) &(p->huartPtr->hdmarx->Instance->CNDTR) // No “C” letter on H7 series
#endif

#ifdef STM32H7
#include "stm32h7xx_hal.h"
#define MY_ISR(p) p->huartPtr->Instance->ISR // No “I” letter on F1 and F4 series
#define MY_ISR_TC USART_ISR_TC
#define NDTR_PTR(p) &(((DMA_Stream_TypeDef*)p->huartPtr->hdmarx->Instance)->NDTR)
#endif
```

Step 2 (continued)

Step 2 (continued)

Modify COM_Init() function to add support for newly added defines for F1, F4 and H7 series:

```
// replace
myPortPtr->ndtrPtr= &(myPortPtr->huartPtr->hdmarx->Instance->CNDTR);
// with
myPortPtr->ndtrPtr=NDTR_PTR(myPortPtr);
```

and COM_WaitTxDone() function:

```
void COM_WaitTxDone(void)
{
    while (READ_BIT(MY_ISR(myPortPtr),MY_ISR_TC)==0){asm("nop");}
}
```

Step 3: Add reading timeout support to port initialization

Add timeOut to COM_Init() function arguments and save it to relevant field of myPort[] array:

```
void COM_Init(uint8_t n,
                  UART_HandleTypeDef *huartPtr, // HAL's UART pointer
                  void *rxBufPtr,      // Receive buffer pointer
                  uint16_t rxBufSize, // Receive buffer size
                  uint16_t timeOut)    // New: timeout in milliseconds
{
    // … some existing code above
    myPortPtr->timeOut = timeOut; // New: set myPort[]->timeOut=…
    myPortPtr->rxFail = false;    // New: clear timeout error flag
    // … some existing code below
}
```

Step 4: Adding reading timeout support to single byte reads

Rename COM_ReadByte() to COM_ReadByteTimeout() and modify it as shown below:

```
bool COM_ReadByteTimeout(uint8_t *b, uint16_t timeOut)
{
    uint16_t dmaPos=myPortPtr->rxBufSize-*myPortPtr->ndtrPtr;
    uint32_t tickStart=HAL_GetTick(); // New: Timout loop
    while ( (dmaPos==myPortPtr->rxPos) && ((HAL_GetTick() - tickStart) < timeOut) )
    {
        dmaPos=myPortPtr->rxBufSize-*myPortPtr->ndtrPtr;
    }

    if (dmaPos!=myPortPtr->rxPos)
    {
        *b=(myPortPtr->rxBufPtr)[myPortPtr->rxPos];
        myPortPtr->rxPos++;
        if (myPortPtr->rxPos==myPortPtr->rxBufSize) {myPortPtr->rxPos=0;}
        return true;
    }
    myPortPtr->rxFail=true; // New: RX failure flag is set
    return false;
}
```

Create two new functions for reading **single** byte:

```
// Fast function exits immediately if circular buffer have no new bytes
bool COM_ReadByteFast(uint8_t *b)
{
    return COM_ReadByteTimeout(b,0); // zero timeout
}

// Normal function waits up to “timeOut” milliseconds
bool COM_ReadByte(uint8_t *b)
{
    return COM_ReadByteTimeout(b,myPortPtr->timeOut);
}
```

Step 4 (continued)

Rename COM_Read() to COM_ReadTimeout() and modify it's beginning as shown below:

```
// …some code above
uint16_t dmaPos=UART_BUFFER_SIZE-*myPortPtr->ndtrPtr;
uint16_t dmaPosOld=dmaPos;
uint32_t tickStart=HAL_GetTick();

// Timeout loop:
do {
    dmaPosOld=dmaPos;
    dmaPos=UART_BUFFER_SIZE-*myPortPtr->ndtrPtr;
    if (dmaPos!=dmaPosOld) {tickStart=HAL_GetTick();} // Reset timeout counter if new byte arrived
    if (dmaPos>rxPos) { dataLen=dmaPos-rxPos; }       // Case 1: simple case
    if (dmaPos<rxPos)                                 // Case 2: data packet is split by array boundary
    {
        dataLen=UART_BUFFER_SIZE-rxPos; // Calculating "tail" length
        dataLen+=dmaPos;                // Adding "head" length
    }
} while ( (dataLen<needLen) && ((HAL_GetTick() - tickStart) < timeOut) ); // Check timeout loop exit requirements
// …some code below
```

Step 5: Adding reading timeout support to multi-byte reads

Update COM_ReadTimeout() ending part as shown below:

```
// …some code above
if (timeOut>0) myPortPtr->rxPos=dmaPos; // New: abandon unread bytes
myPortPtr->rxFail=true;                 // New: update RX fail flag
return false;
// …function ends here
```

Step 5 (continued)

Create two new functions for reading **multiple** bytes:

```
// Fast function exits immediately if circular buffer have no new bytes
bool COM_ReadFast(uint8_t *destBuffer, uint16_t needLen)
{
    return COM_ReadTimeout(destBuffer, needLen,0); // zero timeout
}

// Normal function waits up to “timeOut” milliseconds
bool COM_Read(uint8_t *b)
{
    return COM_ReadTimeout(destBuffer, needLen, myPortPtr->timeOut);
}
```

Step 5 (continued)

Add two functions to clear and check data receiving failures:

```
void COM_CleanRxFail(void) // Clean failure flag
{
    myPortPtr->rxFail=false;
}

bool COM_RxFail(void)      // Get failure flag
{
    return myPortPtr->rxFail;
}
```

Example on rxFail flag usage will be provided in next lessons.

Step 5 (continued)

```
extern void COM_Select(uint8_t n);                          // Set current port

extern void COM_Init(uint8_t n,                             // Initialize port n
                        UART_HandleTypeDef *huartPtr,       // HAL’s UART handler
                        void *rxBufPtr,                     // RX buffer pointer
                        uint16_t rxBufSize,                 // RX buffer size
                        uint16_t timeOut);                  // Read timeout (ms)

extern void COM_Write(void *pData, uint16_t Size);          // Blocking write
extern void COM_WriteFast(void *pData, uint16_t Size);      // Non-blocking write

extern bool COM_ReadByteFast(uint8_t *b);                   // Read byte
extern bool COM_ReadByte(uint8_t *b);                       // Read byte (+timeout)
extern bool COM_ReadFast(uint8_t *destBuffer, uint16_t needLen); // Read bytes
extern bool COM_Read(uint8_t *destBuffer, uint16_t needLen);     // Read bytes (+timeout)
```

Step 6: Update function headers

STM32 UART framing error check is an interesting feature for detecting inconsistences in UART waveform. **But it should be only used when relevant error flags are taken care of!** In other words, there must be some code that handles it. Otherwise, our device sooner or later will stop responding to USART commands.

In next modules, we are going to use CRC32 **checksums**, AES **encryption** and transmission **retry algorithms**. Thus, disabling framing error checking is best choice. We need UART peripheral with predictable behavior.

Step 7: Disabling framing error

To **disable** framing error checking, add this code after UART initialization in COM_Init() function:

```
CLEAR_BIT(huartPtr->Instance->CR3, USART_CR3_EIE);
CLEAR_BIT(huartPtr->Instance->CR1, USART_CR1_PEIE);
```

In most cases, our device will not pass hot-plug test if framing error bits are set and there is no framing errors handling implemented. We do not want our device to mysteriously freeze because of minor EMI interference, static surge or simple cable re-connection.

Step 7 (continued)

Step 8: Adding RS-485 support

Modify COM_Write() function by add RS-485 TX/RX callbacks:

```
void COM_Write(void *pData, uint16_t Size)
{
    COM_WaitTxDone();      // Wait for previous TX completion (if any)
    myPortPtr->_rs485tx(); // New: Set transceiver to TX mode
    HAL_UART_Transmit_DMA(myPortPtr->huartPtr, pData,Size);
    COM_WaitTxDone();
    myPortPtr->_rs485rx(); // New: Set transceiver to RX mode
}
```

When COM_WriteFast() is used, TX/RX state should be controlled in user’s code.

Add dummy callback function:

```
void dummy485() { } // Dummy callback function
```

This function will be called if RS-485 callbacks are not set.

Add callbacks initialization in COM_Init() function:

```
myPortPtr->_rs485tx=dummy485; // RS-485 transmit mode (tx)
myPortPtr->_rs485rx=dummy485; // RS-485 receive mode (rx)
```

Step 8 (continued)

Step 8 (continued)

Add function to set user callbacks for RS-485 mode selection:

```
void UART_SetCallbacks485(void (*f485tx)(void), void (*f485rx)(void))
{
    myPortPtr->_rs485tx=f485tx; // transmit mode callback
    myPortPtr->_rs485rx=f485rx; // receive mode callback
    myPortPtr->_rs485rx();      // set receive mode on start
}
```

Step 8 (continued)

Example of setting RS-485 callbacks from main() function:

```
void rs485tx(void) // Callback #1 to control RS-485 transceiver chip
{
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12,GPIO_PIN_SET); // TX Mode
    HAL_Delay(1);
}

void rs485rx(void) // Callback #2 to control RS-485 transceiver chip
{
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12,GPIO_PIN_RESET); // RX Mode
}

// Place this line right after COM_Init():
UART_SetCallbacks485(rs485tx,rs485rx);
```

Lesson J: UART library usage examples

Adding our UART library to any project is pretty straightforward. First, we use STM32CubeIDE's "Device Configuration Tool" and

* **Enable** all required UART ports
* **Enable** "USART global interrupt" for all ports
* **Enable** DMA TX and RX requests for all ports
* **Set** RX DMA mode to **"Circular mode"** for all ports
* **Generate code**, add uart.h to includes, define buffer arrays
* **Initialize** each port in main.c using COM_Init()
* **Select** current port using COM_Select()
* **Send/receive** data using COM_Read(), COM_Write()

```
// This example answers [0x55] to single byte command [0x01]
// … Variables:
#define UART_BUFFER_SIZE 128                    // circular buffer size
__IO uint8_t uartBuffer[UART_BUFFER_SIZE]; // circular buffer array

// … Initialization COM_Init(number,&handle,&buffer,size,timeout):
COM_Init(0, &huart1, (uint8_t *) &uartBuffer, UART_BUFFER_SIZE, 100);

// … Main() loop:
uint8_t header;
if (COM_ReadFast((uint8_t *)&header,1) && (header==0x01))
{
    uint8_t answer=0x55;
    COM_Write((uint8_t *)&answer,1);
}
```

Step 1: Reading and writing single byte

```
// This example outputs 4-byte temperature value to [0x02] command
// Receiving side can convert temperature back using float floatTemp=intTemp/1000.0;
float floatTemp=-60.3; // -60.3 Deg C
uint8_t header;        // single byte command
if (COM_ReadFast((uint8_t *)&header,1)) switch(header)
    {
        case 0x01: // command [0x01], answer [0x55]. Can be used for device detection
        {
            uint8_t answer=0x55;
            COM_Write((uint8_t *)&answer,1);
            break; // command 0x01
        }
        case 0x02: // command [0x02], answer is [b1] [b2] [b3] [b4]
        {
            int32_t intTemp=round(floatTemp*1000); // convert float to int32
            COM_Write((uint8_t *)&intTemp,4);      // send int32
            break; // command 0x02
        }
    }
```

Step 2: Reading temperature

```
// Adding third command [0x03] [length N] [N x bytes]
// Sends [N x bytes] back to serial port
uint8_t header;
uint8_t length;

if (COM_ReadFast((uint8_t *)&header,1)) switch(header)
{
    case 0x01: //… code from previous step for command [0x01]
    case 0x02: //… code from previous step for command [0x02]
    case 0x03: // Warning! Command [0x03] doesn’t check data length for simplicity!
    {
        COM_Read((uint8_t *)&length,1);           // read packet “length” (1-byte)
        COM_Read((uint8_t *)&tempBuffer,length);  // read [N x bytes]
        COM_Write((uint8_t *)&tempBuffer,length); // send [N x bytes]
        break; // command 0x03
    }
}
```

Step 3: Reading and writing multiple bytes

```c
// Main difference with single port example is that we must add two more receive
// buffers uartBuffer, and run COM_Init() function for each UART handle.
#define UART_BUFFER_SIZE 128
#define TEMP_BUFFER_SIZE 100
__IO uint8_t uartBuffer[3][UART_BUFFER_SIZE]; // three receive buffers
__IO uint8_t tempBuffer[3][TEMP_BUFFER_SIZE]; // three temporary buffers
// Note that we can use single temporary buffer tempBuffer if COM_Write() is used,
// but for non-blocking COM_WriteFast() we should use separate buffer.

//… Initializing three ports
COM_Init(0, &huart1, (uint8_t *) &uartBuffer[0], UART_BUFFER_SIZE, 100);
COM_Init(1, &huart2, (uint8_t *) &uartBuffer[1], UART_BUFFER_SIZE, 100);
COM_Init(2, &huart3, (uint8_t *) &uartBuffer[2], UART_BUFFER_SIZE, 100);
// Select port number by using COM_Select() before calling COM_Read() and COM_Write()

//… Three temperature values
float floatTemp[3]={-60.3,18.0, 121.5 }; // -60.3, 18, 121.5 Deg C

// Continued on next page
```

Step 4: Using three serial ports

```
// Modified example from “Step 3”, loop through three ports:
for (int portNum=0;portNum<3;portNum++)
{
    COM_Select(portNum); // Select current port for COM_Read()/COM_Write() operations
    if (COM_ReadFast((uint8_t *)&header,1)) switch(header)
    {
        case 0x01:{ uint8_t answer=0x55; COM_Write((uint8_t *)&answer,1); break;} //[0x01]
        case 0x02:{ int32_t intTemp=round(floatTemp[portNum]*1000);
                    COM_Write((uint8_t *)&intTemp,4); break; } // Command [0x02]
        case 0x03: // Command [0x03] [length] [length x data bytes]
        {
         COM_Read((uint8_t *)&length,1);                     // read [length]
         COM_Read((uint8_t *)&tempBuffer[portNum],length);  // read [length x data bytes]
         COM_Write((uint8_t *)&tempBuffer[portNum],length); // write [length x data bytes]
         break; // command 0x03
        }
    }
}
```

Step 4 (continued)

In above example, we use COM_ReadFast() to read first byte of packet. We want to **avoid** blocking of main loop execution with reading timeout if circular buffer is empty.

Consecutive bytes are read using COM_Read(), because we do not want to interrupt data packet reception before it's complete.

* COM_ReadFast() is used when we do not want to use reading timeouts, e.g. to check if first byte of data packet is arrived
* COM_Read() is used when we want to use reading timeouts

Step 5: How to choose between COM_Read() and COM_ReadFast()

In most cases, COM_Write() is best choice, because it guarantees that UART transfer is complete, and write buffer can be modified freely. COM_WriteFast() can be convenient to run heavy calculations and DMA data transfers in parallel.

* COM_WriteFast() doesn’t wait for current transfer completion, but will wait if any previous transfer is not complete. Thus, we still should avoid using this function inside any interrupts.
* COM_Write() waits for current transfer completion before function execution is complete.

Step 6: Difference between COM_Write() and COM_WriteFast()

Single-byte and multi-byte read/write functions compared:

```
// single-byte read function
if (COM_ReadByteFast((uint8_t *)&header)) { … }

// replaced by multi-byte read with length argument 1
if (COM_ReadFast((uint8_t *)&header,1)) { … }

// we can implement single-byte read function using multi-byte read function
bool COM_ReadByteFast(uint8_t *b)
{
    COM_ReadFast(b,1);
}
```

Step 7: COM_ReadByte and COM_ReadByteFast functions

More modules on STM32's UART are being prepared!

* Checksum calculation and retry algorithms
* Communication protocols and how to work with them
* Custom bootloader, encryption and firmware update
* UART to UART bridging over DMA
* Data pumping for DIY oscilloscopes, audio, etc.
* Porting STM32 library to PIC32MK or other MCUs

Introduction to next modules

Where to download

Download accompanying examples and PDF version of this STM32 course:

* At website www.thundertronics.com
* Check links under videos on this YouTube channel: https://www.youtube.com/@ThundertronicsOfficial