8. Universal Asynchronous Serial Communications

Nowadays there is a really high number of serial communication protocols and hardware interfaces available in the electronics industry. The most of them are focused on high transmission bandwidths, like the more recent USB 2.0 and 3.0 standards, the Firewire (IEEE 1394) and so on. Some of these standards come from the past, but are still widespread especially as communication interface between modules on the same board. One of this is the *Universal Synchronous/Asynchronous Receiver/Transmitter* interface, also simply known as USART.

Almost every microcontroller provides at least one UART peripheral. Almost all STM32 MCUs provide at least two UART/USART interfaces, but the most of them provide more than two interfaces (some up to eight interfaces) according the number of I/O supported by the MCU package.

In this Chapter we will see how to program this really useful peripheral using the CubeHAL. Moreover, we will study how to develop applications using the UART both in *polling* and *interrupt* modes, leaving the third operative mode, the *DMA*, to the next chapter.

8.1 Introduction to UARTs and USARTs

Before we start diving into the analysis of the functions provided by the HAL to manipulate universal serial devices, it is best to take a brief look to the UART/USART interface and its communication protocol.

When we want two exchange data between two (or even more) devices, we have two alternatives: we can transmit it in parallel, that is using a given number of communication lines equal to the size of the each data word (e.g., eight independent lines for a word made of eight bits), or we can transmit each bit constituting our word one by one. A UART/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.

When the information flows between two devices inside a common channel, both devices (here, for simplicity, we will refer to them as *the sender* and *the receiver*) have to agree on the *timing*, that this 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 (usually the device that acts as *the master* of this interconnection system).

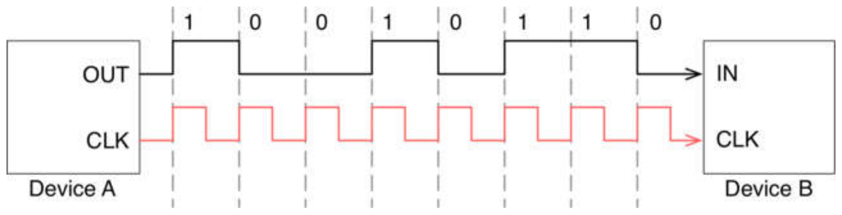


Figure 1: A serial communication between two devices using a shared clock source

In **Figure 1** we have a typical timing diagram[1] showing the *Device A* sending one byte (`0b01101001`) serially to the *Device B* using a common reference clock. The common clock is also used to agree on when to start *sampling* the sequence of bits: when the master device starts *clocking* the dedicated line, it means that it is going to send a sequence of bits.

In a **synchronous transmission** the transmission speed and duration are defined by the clock: its frequency determines how fast we can transmit a single byte on the communication channel[2]. But if both devices involved in data transmission agree on how long it takes to transmit a single bit and when to start and finish to sample transmitted bits, than we can avoid to use a dedicated clock line. In this case we have an **asynchronous transmission**.

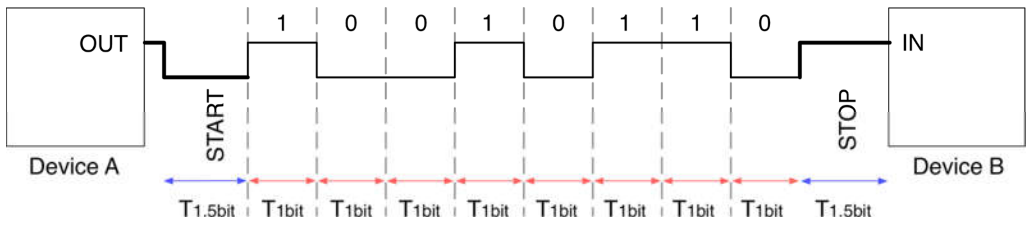


Figure 2: The timing diagram of a serial communication without a dedicated clock line

Figure 2 shows the timing diagram of an asynchronous transmission. The idle state (that is, no transmission occurring) is represented by the high signal. Transmission begins with a **START** bit, which is represented by the low level. The negative edge is detected by the receiver and 1.5 bit periods after this (indicated in Figure 1s $T_{1.5bit}$), the sampling of bits begins. Eight data bits are sampled. The least significant bit (LSB) is typically transmitted first. An optional parity bit is then transmitted (for error checking of the data bits). Often this bit is omitted if the transmission channel is assumed to be noise free or if there are error checking higher up in the protocol layers. The transmission is ended by a STOP bit, which last 1.5 bits.

[1]A Timing Diagram is a representation of a set of signals in the time domain.

[2]However, keep in mind that the maximum transmission speed is determined by a lot of other things, like the characteristics of the electrical channel, the ability of each device involved in transmission to sample fast signals, and so on.

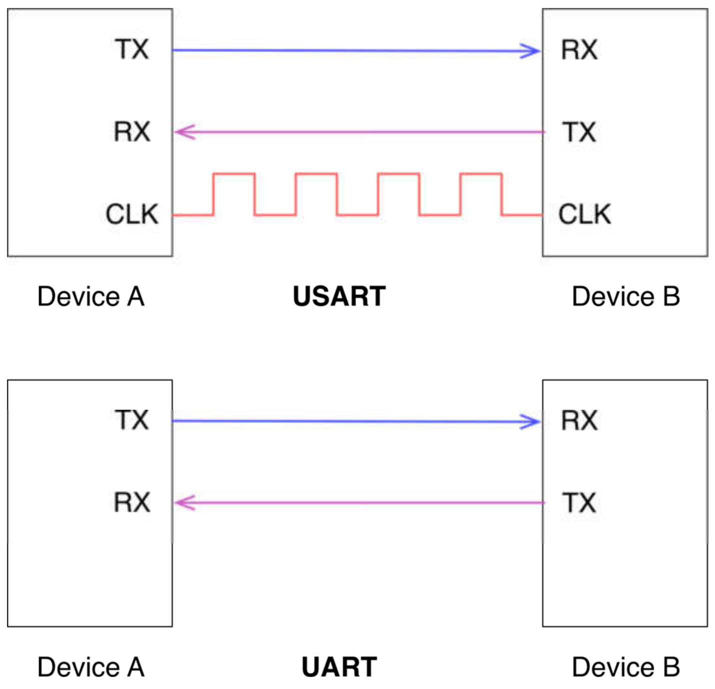


Figure 3: The signaling difference between a USART and a UART

A *Universal Synchronous Receiver/Transmitter* interface is a device able to transmit data word serially using two I/Os, one acting as transmitter (TX) and one as receiver (RX), plus one additional I/O as one clock line, while a *Universal Asynchronous Receiver/Transmitter* uses only two RX/TX I/Os (see **Figure 3**). Traditional we refer to the first interface with the term **USART** and to the second one with the term **UART**.

A UART/USART defines the signaling method, but it say nothing about the voltage levels. This means that an STM32 UART/USART will use the voltage levels of the MCU I/Os, which is almost equal to VDD (it is also common to refer to these voltage levels as *TTL voltage levels*). The way these voltage levels are translated to allow serial communication outside the board is demanded to other communication standards. For example, the EIA-RS232 or EIA-RS484 are two really popular standards that define signaling voltages, in addition to their timing and meaning, and the physical size and pinout of connectors. Moreover, UART/USART interfaces can be used to exchange data using other physical and logical serial interfaces. For example, the FT232RL is a really popular IC that allows to map a UART to a USB interface, as shown in **Figure 4**.

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.

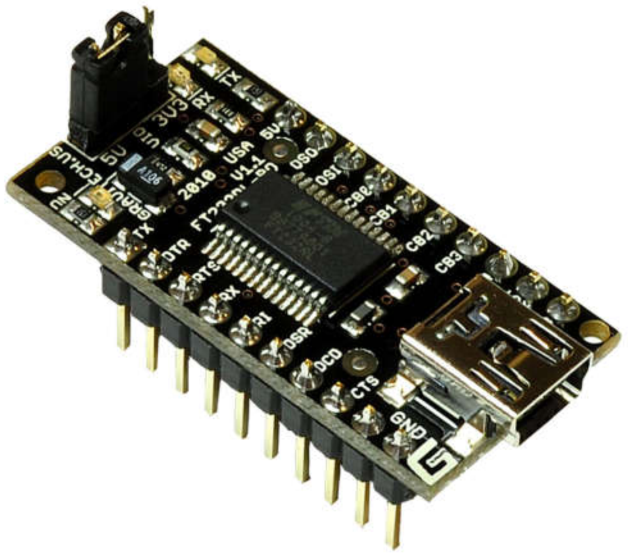


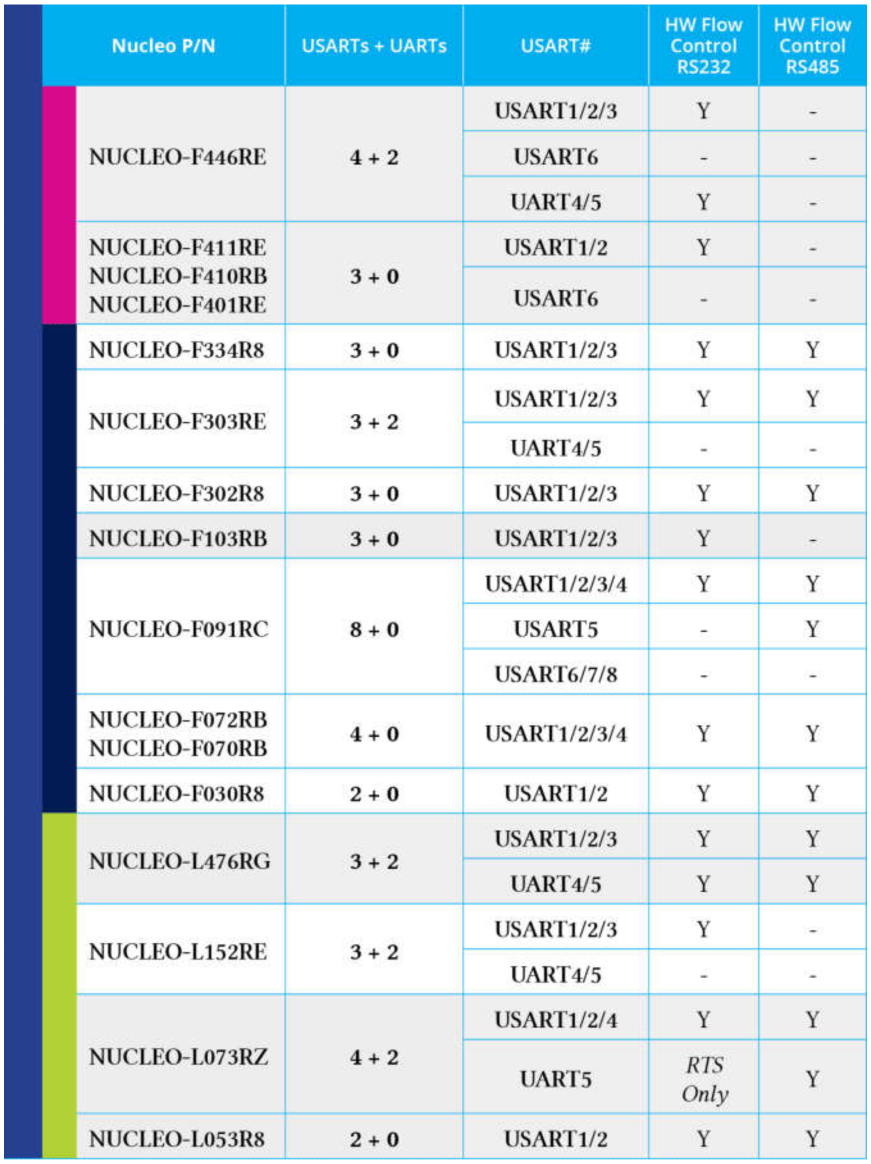
Figure 4: A typical circuit based on FT232RL used to convert a 3.3V TTL UART interface to USB

STM32 microcontrollers provide a variable number of USARTs, which can be configured to work both in *synchronous* and *asynchronous* mode. Some STM32 MCUs also provide interfaces only able to act as UART. **Table 1** lists the UART/USARTs provided by STM32 MCUs equipping all Nucleo boards. The most of USARTs are also able to automatically implement *Hardware Flow Control*, both for the RS232 and the RS485 standards.

All Nucleo-64 boards are designed so that the USART2 of the target MCU is linked to the ST-LINK interface[3]. When we install the ST-LINK drivers, an additional driver for the *Virtual COM Port*(VCP) is also installed: this allows us to access to the target MCU USART2 using the USB interface, without using a dedicated TTL/USB converter. Using a terminal emulation program we can exchange messages and data with our Nucleo.

The CubeHAL separates the API for the management of UART and USART interfaces. All functions and C type handlers used for the handling of USARTs start with the `HAL_USART` prefix and are contained inside the files `stm32xxx_hal_usart.{c,h}`, while those related to UARTs management start with the `HAL_UART` prefix and are contained inside the files `stm32xxx_hal_uart.{c,h}`. Since both the modules are conceptually identical, and since the UART is the most common form of serial interconnection between different modules, this book will only cover the features of the `HAL_UART` module.

[3]Please take note that this statement may not be true if you are using a Nucleo-32 or Nucleo-144 board. Check the ST documentation for more about this.



Nucleo P/N	USARTs + UARTs	USART#	HW Flow Control RS232	HW Flow Control RS485
NUCLEO-F446RE	4 + 2	USART1/2/3	Y	-
		USART6	-	-
		UART4/5	Y	-
NUCLEO-F411RE NUCLEO-F410RB NUCLEO-F401RE	3 + 0	USART1/2	Y	-
		USART6	-	-
NUCLEO-F334R8	3 + 0	USART1/2/3	Y	Y
NUCLEO-F303RE	3 + 2	USART1/2/3	Y	Y
		UART4/5	-	-
NUCLEO-F302R8	3 + 0	USART1/2/3	Y	Y
NUCLEO-F103RB	3 + 0	USART1/2/3	Y	-
NUCLEO-F091RC	8 + 0	USART1/2/3/4	Y	Y
		USART5	-	Y
		USART6/7/8	-	-
NUCLEO-F072RB NUCLEO-F070RB	4 + 0	USART1/2/3/4	Y	Y
NUCLEO-F030R8	2 + 0	USART1/2	Y	Y
NUCLEO-L476RG	3 + 2	USART1/2/3	Y	Y
		UART4/5	Y	Y
NUCLEO-L152RE	3 + 2	USART1/2/3	Y	-
		UART4/5	-	-
NUCLEO-L073RZ	4 + 2	USART1/2/4	Y	Y
		UART5	*RTS Only*	Y
NUCLEO-L053R8	2 + 0	USART1/2	Y	Y

Table 1: The list of available USARTs and UARTs on all Nucleo boards

8.2 UART Initialization

Like all STM32 peripherals, even the USARTs[4] are mapped in the memory mapped peripheral region, which starts from `0x4000 0000`. The CubeHAL abstracts the effective location of each USART for a given STM32 MCU thanks to the `USART_TypeDef`[5] descriptor. For example, we can simply use the `USART2` macro to refer to the second USART peripheral provided by all STM32 microcontrollers with LQFP64 package.

[4]Starting from this paragraph, the terms USART and UART are used interchangeably, unless different noticed.
[5]The analysis of the fields of this C `struct` is outside of the scope of this book.

However, all the HAL functions related to UART management are designed so that they accept as first parameter an instance of the C `struct UART_HandleTypeDef`, which is defined in the following way:

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

Let us see more in depth the most important fields of this `struct`.

- `Instance`: is the pointer to the USART descriptor we are going to use. For example, `USART2` is the descriptor of the UART associated to the ST-LINK interface of every Nucleo board.
- `Init`: is an instance of the C `struct UART_InitTypeDef`, which is used to configure the UART interface. We will study it more in depth in a while.
- `AdvancedInit`: this field is used to configure more advanced UART features like the automatic *BaudRate* detection and the TX/RX pin swapping. Some HALs do not provide this additional field. This happens because USART interfaces are not equal for all STM32 MCUs. This is an important aspect to keep in mind while choosing the right MCU for your application. The analysis of this field is outside the scope of this book.
- `pTxBuffPtr` and `pRxBuffPtr`: these fields point to the transmit and receive buffer respectively. They are used as source to transmit `TxXferSize` bytes over the UART and to receive `RxXferSize` when the UART is configured in Full Duplex Mode. The `TxXferCount` and `RxXferCount` fields are used internally by the HAL to take count of transmitted and received bytes.
- `Lock`: this field is used internally by the HAL to lock concurrent accesses to UART interfaces.

As said above, the `Lock` field is used to rule concurrent accesses in almost all HAL routines. If you take a look to the HAL code, you can see several uses of the `__HAL_LOCK()` macro, which is expanded in this way:

```
#define __HAL_LOCK(__HANDLE__)                   \
    do{                                          \
       if((__HANDLE__)->Lock == HAL_LOCKED)      \
       {                                         \
           return HAL_BUSY;                      \
       }                                         \
       else                                      \
       {                                         \
           (__HANDLE__)->Lock = HAL_LOCKED;      \
       }                                         \
    }while (0)
```

It is not clear why ST engineers decided to take care of concurrent accesses to the HAL routines. Probably they decided to have a *thread safe* approach, freeing the application developer from the responsibility of managing multiple accesses to the same hardware interface in case of multiple threads running in the same application.

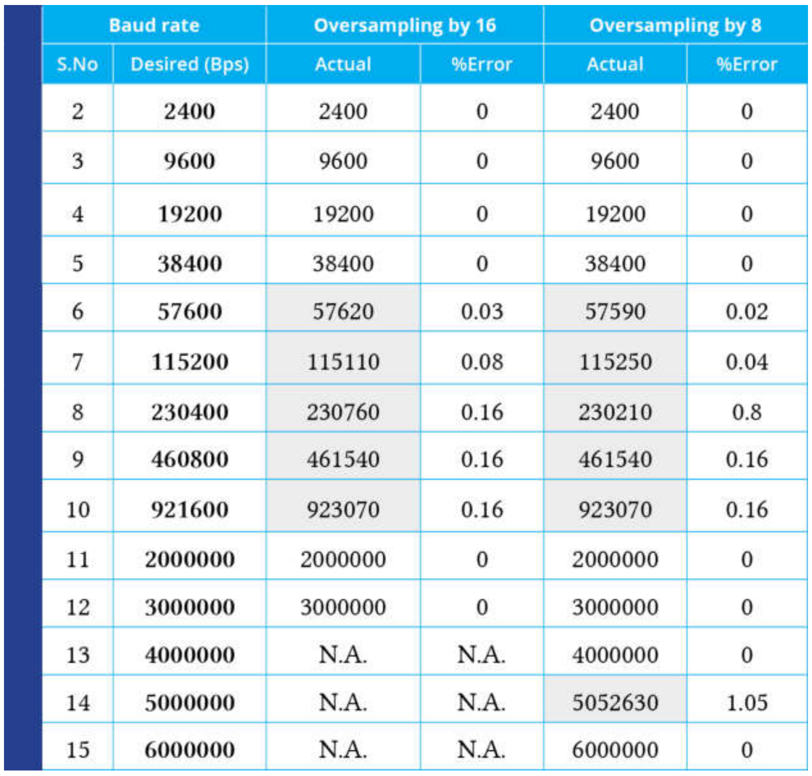
However, this has an annoying side effect for all HAL users: even if my application does not perform concurrent accesses to the same peripheral, my code will be poor optimized by a lot of checks about the state of the `Lock` field. Moreover, that way to lock is intrinsically thread unsafe, because there is no critical section used to prevent race conditions in case a more privileged ISR preempts the running code. Finally, if my application uses an RTOS, it is much better to use native OS locking primitives (like semaphores and mutexes which are not only *atomic*, but also correctly manages the task scheduling avoiding the *busy waiting*) to handle concurrent accesses, without the need to check for a particular return value (`HAL_BUSY`) of the HAL functions.

A lot of developers have disapproved this way to lock peripherals since the first release of the HAL. ST engineers have recently announced that they are actively working on a better solution.

All the UART configuration activities are performed by using an instance of the C `struct UART_-InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t BaudRate;
  uint32_t WordLength;
  uint32_t StopBits;
  uint32_t Parity;
  uint32_t Mode;
  uint32_t HwFlowCtl;
  uint32_t OverSampling;
} UART_InitTypeDef;
```

- `BaudRate`: this parameter refers to the connection speed, expressed in bits per seconds. Even if the parameter can assume an arbitrary value, usually the *BaudRate* comes from a list of well-known and standard values. This because it is a function of the peripheral clock associated to the USART (that is derived from the main HSI or HSE clock by a complex chain of PLLs and multipliers in some STM32 MCU), and not all *BaudRate*s can be easily achieved without introducing sampling errors, and hence communication errors. **Table 2** shows the list of common *BaudRate*s, and the related error calculation, for an STM32F030 MCU. Always consult the reference manual for your MCU to see which peripheral clock frequency best fits the needed *BaudRate* on the given STM32 microcontroller.



Baud rate		Oversampling by 16		Oversampling by 8	
S.No	Desired (Bps)	Actual	%Error	Actual	%Error
2	2400	2400	0	2400	0
3	9600	9600	0	9600	0
4	19200	19200	0	19200	0
5	38400	38400	0	38400	0
6	57600	57620	0.03	57590	0.02
7	115200	115110	0.08	115250	0.04
8	230400	230760	0.16	230210	0.8
9	460800	461540	0.16	461540	0.16
10	921600	923070	0.16	923070	0.16
11	2000000	2000000	0	2000000	0
12	3000000	3000000	0	3000000	0
13	4000000	N.A.	N.A.	4000000	0
14	5000000	N.A.	N.A.	5052630	1.05
15	6000000	N.A.	N.A.	6000000	0

Table 2: Error calculation for programmed baud rates at 48 MHz in both cases of oversampling by 16 or by 8

- `WordLength`: it specifies the number of data bits transmitted or received in a frame. This field can assume the value `UART_WORDLENGTH_8B` or `UART_WORDLENGTH_9B`, which means that we can transmit over a UART packets containing 8 or 9 data bits. This number does not include the overhead bits transmitted, such as the start and stop bits.
- `StopBits`: this field specifies the number of stop bits transmitted. It can assume the value `UART_STOPBITS_1` or `UART_STOPBITS_2`, which means that we can use one or two stop bits to signal the end of the frame.
- `Parity`: it indicates the parity mode. This field can assume the values from **Table 3**. Take note that, when parity is enabled, the computed parity is inserted at the MSB position of the transmitted data (9th bit when the word length is set to 9 data bits; 8th bit when the word length is set to 8 data bits). Parity is a very simple form of error checking. It comes in two flavors: *odd* or *even*. To produce the parity bit, all data bits are added up, and the evenness of the sum decides whether the bit is set or not. For example, assuming parity is set to *even* and was being added to a data byte like `0b01011101`, which has an odd number of 1's (5), the parity bit would be set to `1`. Conversely, if the parity mode was set to odd, the parity bit would be `0`. Parity is optional, and not very widely used. It can be helpful for transmitting across noisy mediums, but it will also slow down data transfer a bit and requires both sender and receiver to implement error-handling (usually, received data that fails must be re-sent). When a *parity error* occurs, all STM32 MCUs generate a specific interrupt, as we will see next.
- `Mode`: it specifies whether the RX or TX mode is enabled or disabled. This field can assume one of the values from **Table 4**.
- `HwFlowCtl`: it specifies whether the RS232[6] Hardware Flow Control mode is enabled or disabled. This parameter can assume one of the values from **Table 5**.

Table 3: Available parity modes for a UART connection

Parity Mode	Description
`UART_PARITY_NONE`	No parity check enabled
`UART_PARITY_EVEN`	The parity bit is set to `1` if the count of bits equal to `1` is odd
`UART_PARITY_ODD`	The parity bit is set to `1` if the count of bits equal to `1` is even

Table 4: Available UART modes

UART Mode	Description
`UART_MODE_RX`	The UART is configured only in receive mode
`UART_MODE_TX`	The UART is configured only in transmit mode
`UART_MODE_TX_RX`	The UART is configured to work bot in receive an transmit mode

[6]this field is only used to enable the RS232 flow control. To enable the RS485 flow control, the HAL provides a specific function, `HAL_RS485Ex_Init()`, defined inside the `stm32xxxx_hal_uart_ex.c` file.

Table 5: Available flow control mode for a UART connection

Flow Control Mode	Description
`UART_HWCONTROL_NONE`	The Hardware Flow Control is disabled
`UART_HWCONTROL_RTS`	The *Request To Send*(RTS) line is enabled
`UART_HWCONTROL_CTS`	The *Clear To Send*(CTS) line is enabled
`UART_HWCONTROL_RTS_CTS`	Both RTS and CTS lines enabled

- `OverSampling`: when the UART receives a frame from the remote peer, it samples the signals in order to compute the number of 1 and 0 constituting the message. *Oversampling* is the technique of sampling a signal with a sampling frequency significantly higher than the Nyquist rate. The receiver implements different user-configurable oversampling techniques (except in synchronous mode) for data recovery by discriminating between valid incoming data and noise. This allows a trade-off between the maximum communication speed and noise/clock inaccuracy immunity. The `OverSampling` field can assume the value `UART_OVERSAMPLING_16` to perform 16 samples for each frame bit or `UART_OVERSAMPLING_8` to perform 8 samples. **Table 2** shows the error calculation for programmed baud rates at 48 MHz in an STM32F030 MCU in both cases of oversampling by 16 or by 8.

Now it is a good time to start writing down a bit of code. Let us see how to configure the USART2 of the MCU equipping our Nucleo to exchange messages through the ST-LINK interface.

```c
int main(void) {
  UART_HandleTypeDef huart2;

  /* Initialize the HAL */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Configure the USART2 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 38400;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  HAL_UART_Init(&huart2);
  ...
}
```

The first step is to configure the USART2 peripheral. Here we are using this configuration: 38400, N, 1. That is, a *BaudRate* equal to 38400 Bps, no parity check and just one stop bit. Next, we disable any form of Hardware Flow Control and we choose the highest oversampling rate, that is 16 clock ticks for each transmitted bit. The call to the `HAL_UART_Init()` function ensures that the HAL initializes the USART2 according the given options.

However, the above code is still not sufficient to exchange messages through the Nucleo Virtual COM Port. Don't forget that every peripheral designed to exchange data with the outside world must be properly bound to corresponding GPIOs, that is we have to configure the USART2 TX and RX pins. Looking to the Nucleo schematics, we can see that USART2 TX and RX pins are PA2 and PA3 respectively. Moreover, we have already seen in Chapter 4 that the HAL is designed so that `HAL_UART_Init()` function automatically calls the `HAL_UART_MspInit()` (see **Figure 19** in Chapter 4) to properly initialize the I/Os: it is our responsibility to write this function in our application code, which we will be automatically called by the HAL.

Is It Mandatory to Define This Function?

The answer is simply no. This is just a practice enforced by the HAL and by the code automatically generated by CubeMX. The `HAL_UART_MspInit()`, and the corresponding function `HAL_UART_MspDeInit()` which is called by the `HAL_UART_DeInit()` function, are declared inside the HAL in this way:

```
__weak void HAL_UART_MspInit(UART_HandleTypeDef *huart);
```

The function attribute `__weak` is a GCC way to declare a symbol (here, a function name) with a weak scope visibility, which we will be overwritten if another symbol with the same name with a global scope (that is, without the __weak attribute) is defined elsewhere in the application (that is, in another relocatable file). The linker will automatically substitute the call to the function `HAL_UART_MspInit()` defined inside the HAL if we implement it in our application code.

The code below shows how to correctly code the `HAL_UART_MspInit()` function.

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) {
  GPIO_InitTypeDef GPIO_InitStruct;
  if(huart->Instance==USART2) {
    /* Peripheral clock enable */
    __HAL_RCC_USART2_CLK_ENABLE();

    /**USART2 GPIO Configuration
    PA2     ------> USART2_TX
    PA3     ------> USART2_RX
    */
    GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_USART2; /* WARNING: this depends on
                                                    the specific STM32 MCU */
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  }
}
```

As you can see, the function is designed so that it is common for every USART used inside the application. The `if` statement disciplines the initialization code for the given USART (in our case, USART2). The remaining of code configures the PA2 and PA3 pins. **Please, take note that the alternate function may change for the MCU equipping your Nucleo**. Consult the book examples to see the right initialization code for your Nucleo.

Once we have configured the USART2 interface, we can start exchanging messages with our PC.

F334 F303

Please, take note that the code presented before could not be sufficient to correctly initialize the USART peripheral for some STM32 MCUs. Some STM32 microcontrollers, like the STM32F334R8, allow the developer to choose the clock source for a given peripheral (for example, the USART2 in an STM32F334R8 MCU can be optionally clocked from `SYSCLK`, `HSI`, `LSE` or `PCLK1`). It is strongly suggested to use CubeMX the first time you configure the peripherals for your MCU and to check carefully the generated code looking for this kind of exceptions. Otherwise, the datasheet is the only source for this information.

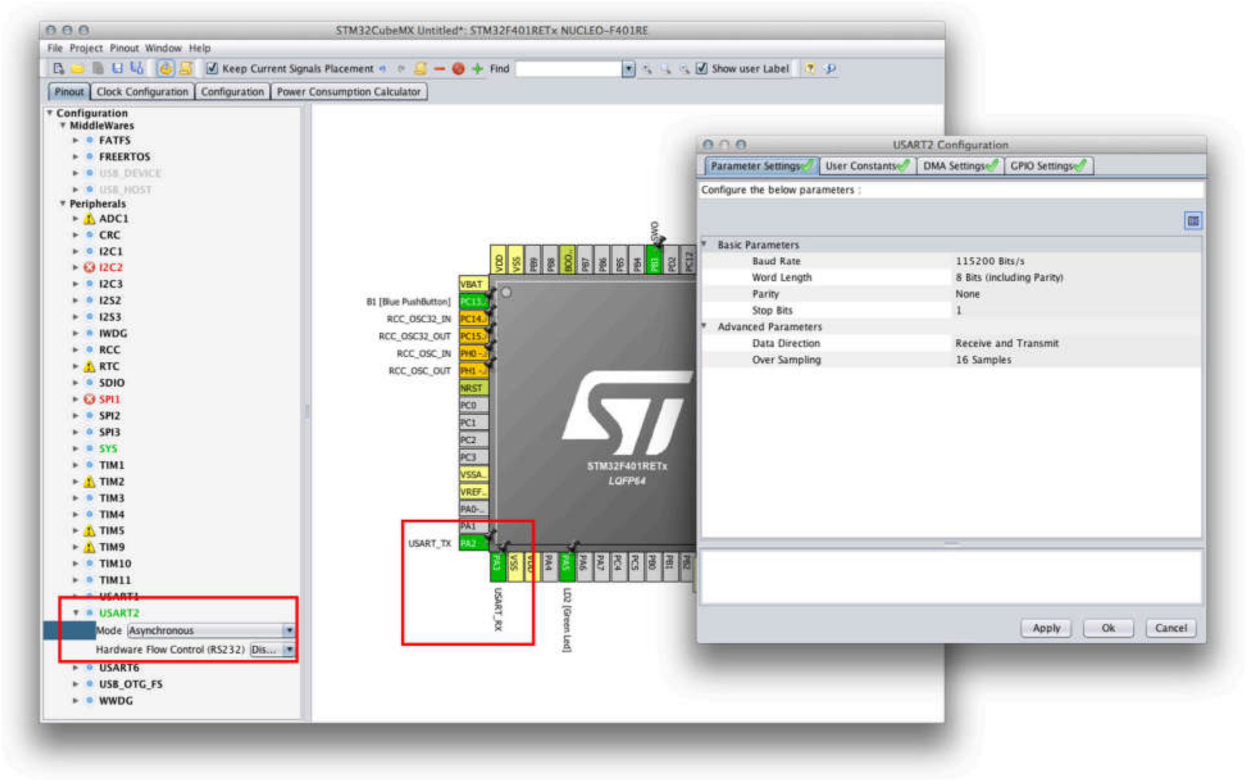
8.2.1 UART Configuration Using CubeMX

As said before, the first time we configure the USART2 for our Nucleo it is best to use CubeMX. The first step is enabling the USART2 peripheral inside the *Pinout* view, selecting the *Asynchronous* entry from the *Mode* combo box, as shown in **Figure 5**. Both PA2 and PA3 pins will be automatically highlighted in green. Then, go inside the *Configuration* section and click on the **USART2** button. The configuration dialog will appear, as shown in **Figure 5** on the right[7]. This allows us to configure the USART configuration settings, such as the *BaudRate*, word length and so on[8].

[7]Please, take note that the **Figure 5** is obtained combining two captures in one figure. It is not possible to show the USART configuration dialog from the *Pinout* view.

[8]Some of you, especially those having a Nucleo-F3, will notice that the configuration dialog is different from the one shown in **Figure 5**. Please, refer to the reference manual for your target MCU for more information.

Once we have configured the USART interface, we can generate the C code. You will notice that CubeMX places all the USART2 initialization code inside the `MX_USART2_UART_Init()` (which is contained in the `main.c` file). Instead, all the code related to GPIO configuration is placed into the `HAL_UART_MspInit()` function, which is contained inside the `stm32xxxx_hal_msp.c` file.



USART2
Mode: Asynchronous
Hardware Flow Control (RS232): Dis...

Configure the below parameters :

Basic Parameters	
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

Figure 5: CubeMX can be used to configure the UART2 interface easily

8.3 UART Communication in *Polling Mode*

STM32 microcontrollers, and hence the CubeHAL, offer three ways to exchange data between peers over a UART communication: *polling*, *interrupt* and *DMA* mode. It is important to stress right from now that these modes are not only three different flavors to handle UART communications. They are three different programming approach to the same task, which introduce several benefits both from the design and performance point of view. Let us introduce them briefly.

- 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 in our application (the classical example is the usage of the UART as output console for debug activities).

- In *interrupt mode*, also called *non-blocking mode*, the main application is freed from waiting for the completion of data transmission and reception. The data transfer routines terminate as soon as they complete to configure the peripheral. When the data transmission ends, a subsequent interrupt will signal the main code about this. This mode is more suitable when communication speed is low (below 38400 Bps) or when it happens "rarely", compared to other activities performed by the MCU, and we do not want to stuck it waiting for data transmission.
- *DMA mode* offers the best data transmission throughput, thanks to the direct access of the UART peripheral to MCU internal RAM. This mode is best for high-speed communications and when we totally want to free the MCU from the overhead of data transmission. Without the *DMA mode*, it is almost impossible to reach the fastest transfer rates that the USART peripheral is capable to handle. In this chapter we will not see this USART communication mode, leaving it to the next chapter dedicated to DMA management.

To transmit a sequence of bytes over the USART in *polling mode* the HAL provides the function

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
                                    uint16_t Size, uint32_t Timeout);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: is the pointer to an array, with a length equal to the `Size` parameter, containing the sequence of bytes we are going to transmit;
- `Timeout`: is the maximum time, expressed in milliseconds, we are going to wait for the transmit completion. If the transmission does not complete in the specified timeout time, the function aborts and returns the `HAL_TIMEOUT` value; otherwise it returns the `HAL_OK` value if no other errors occur. Moreover, we can pass a timeout equal to `HAL_MAX_DELAY` (`0xFFFF FFFF`) to wait indefinitely for the transmit completion.

Conversely, to receive a sequence of bytes over the USART in polling mode the HAL provides the function

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;

- `pData`: is the pointer to an array, with a length at lest equal to the `Size` parameter, containing the sequence of bytes we are going to receive. The function will block until all bytes specified by the `Size` parameter are received.
- `Timeout`: is the maximum time, expressed in milliseconds, we are going to wait for the receive completion. If the transmission does not complete in the specified timeout time, the function aborts and returns the `HAL_TIMEOUT` value; otherwise it returns the `HAL_OK` value if no other errors occur. Moreover, we can pass a timeout equal to `HAL_MAX_DELAY` (`0xFFFF FFFF`) to wait indefinitely for the receive completion.

Read Carefully

It is important to remark that the timeout mechanism offered by the two functions 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).

Ok. Now it is the right time to see an example.

Filename: `src/main-ex1.c`

```c
int main(void) {
  uint8_t opt = 0;

  /* Reset of all peripherals, Initializes the Flash interface and the SysTick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();

printMessage:

  printWelcomeMessage();

  while (1)  {
    opt = readUserInput();
    processUserInput(opt);
    if(opt == 3)
      goto printMessage;
  }
}
```

```
45
46  void printWelcomeMessage(void) {
47    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"), HAL_MAX_DELAY);
48    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"), HAL_MAX_DELAY);
49    HAL_UART_Transmit(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG), HAL_MAX_DELAY);
50    HAL_UART_Transmit(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU), HAL_MAX_DELAY);
51  }
52
53  uint8_t readUserInput(void) {
54    char readBuf[1];
55
56    HAL_UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT), HAL_MAX_DELAY);
57    HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
58    return atoi(readBuf);
59  }
60
61  uint8_t processUserInput(uint8_t opt) {
62    char msg[30];
63
64    if(!opt || opt > 3)
65      return 0;
66
67    sprintf(msg, "%d", opt);
68    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
69
70    switch(opt) {
71    case 1:
72      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
73      break;
74    case 2:
75      sprintf(msg, "\r\nUSER BUTTON status: %s",
76              HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED\
77  ");
78      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
79      break;
80    case 3:
81      return 2;
82    };
83
84    return 1;
85  }
```

The example is a sort of bare-bone management console. The application starts printing a welcome message (lines 36) and then entering in a loop waiting for the user choice. The first option allows to toggle the LD2 LED, while the second to read the status of the USER button. Finally, the option 3 causes that the welcome screen is printed again.

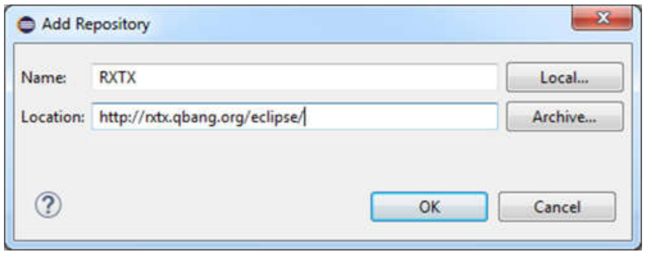
The two strings `"\033[0;0H"` and `"\033[2J"` are *escape sequences*. They are standard sequences of chars used to manipulate the terminal console. The first one places the cursor in the top-left part of the available console screen, and the second one clears the screen.

To interact with this simple management console, we need a serial communication program. There are several options available. The easy one is to use a standalone program like `putty`[9] for the Windows platform (if you have an old Windows version, you can also consider to use the classical `HyperTerminal` tool), or `kermit`[10] for Linux and MacOS. However, we will now introduce a solution to have an integrated serial communication tool inside the Eclipse IDE. As usual, the instructions differ between Windows, Linux and MacOS.

8.3.1 Installing a Serial Console in Windows

For the Windows OS we have a simple and reliable solution. This is based on two plug-ins. The first one is a wrapper plug-in around the RXTX[11] Java library. To install it, go to **Help->Install software…** menu, then click on the **Add…** button, and fill the fields in the following way: (see **Figure 6**).

Name: RXTX
Location: http://rxtx.qbang.org/eclipse/


Add Repository

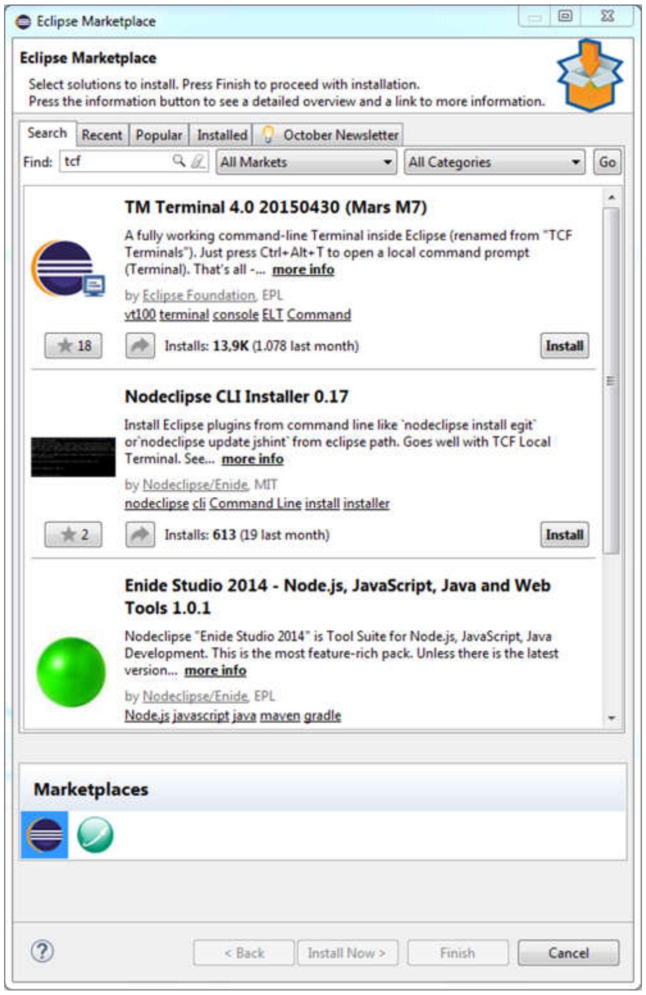
Name: RXTX

Location: http://rxtx.qbang.org/eclipse/

Figure 6: The dialog to add a new plug-in repository

Click on **OK** and install the release **RXTX 2.1-7r4** following the instructions.
Once, the installation has been completed, go to **Help->Eclipse Marketplace…**. In the **Find** text box write “tcf”. After a while, the **TM Terminal** plug-in should appear, as shown in **Figure 7**. Click on the **Install** button and follow the instructions. Restart Eclipse when requested.

[9] http://bit.ly/1jsQjnt
[10] http://www.columbia.edu/kermit/
[11] http://rxtx.qbang.org/



Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

TM Terminal 4.0 20150430 (Mars M7)

A fully working command-line Terminal inside Eclipse (renamed from "TCF Terminals"). Just press Ctrl+Alt+T to open a local command prompt (Terminal). That's all -... more info

by Eclipse Foundation, EPL
vt100 terminal console ELT Command

18 Installs: 13,9K (1,078 last month)

Nodeclipse CLI Installer 0.17

Install Eclipse plugins from command line like `nodeclipse install egit` or `nodeclipse update jshint` from eclipse path. Goes well with TCF Local Terminal. See... more info

by Nodeclipse/Enide, MIT
nodeclipse cli Command Line install installer

2 Installs: 613 (19 last month)

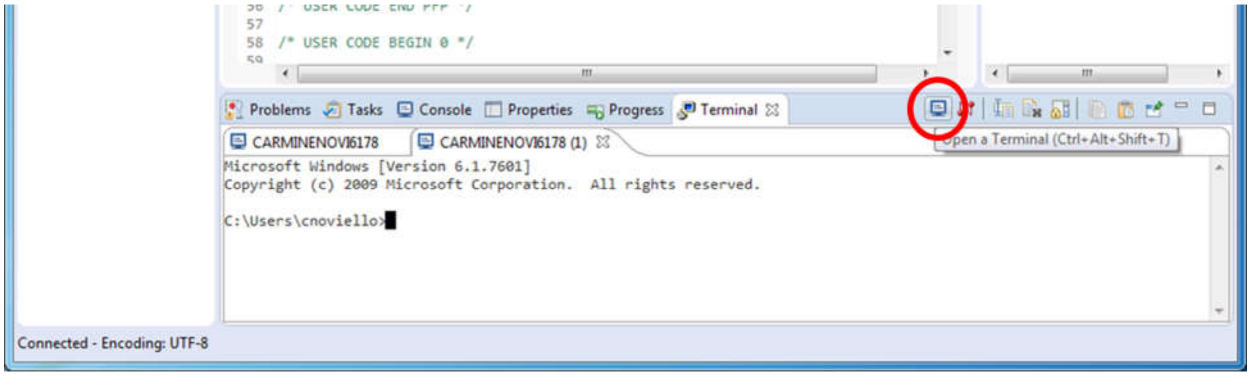
Enide Studio 2014 - Node.js, JavaScript, Java and Web Tools 1.0.1

Nodeclipse "Enide Studio 2014" is Tool Suite for Node.js, JavaScript, Java Development. This is the most feature-rich pack. Unless there is the latest version... more info

by Nodeclipse/Enide, EPL
Node.js javascript java maven gradle

Figure 7: The Eclipse Marketplace

To open the Terminal panel you can simply press **Ctrl+Alt+T**, or you can go to **Window->Show View->Other**... menu and search for **Terminal** view.

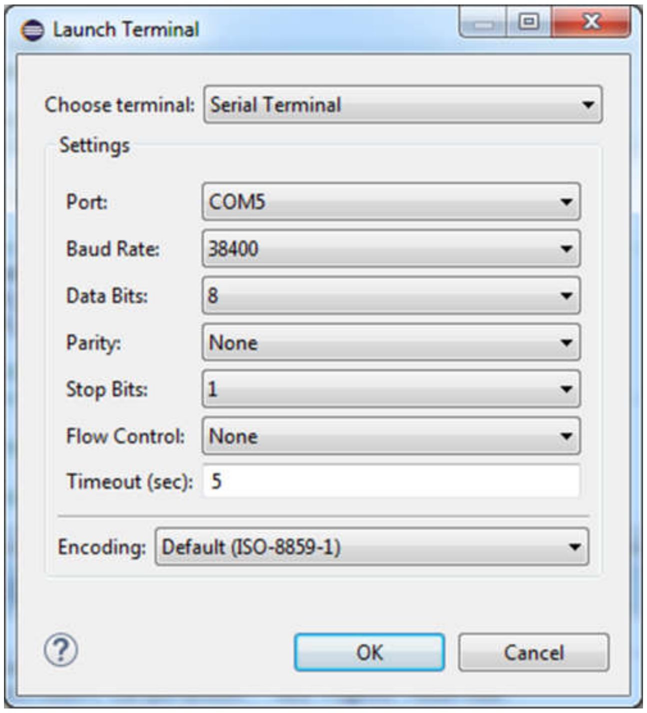


```
/* USER CODE END PFP */

/* USER CODE BEGIN 0 */
```

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\cnoviello>
```

Figure 8: How to start a new terminal

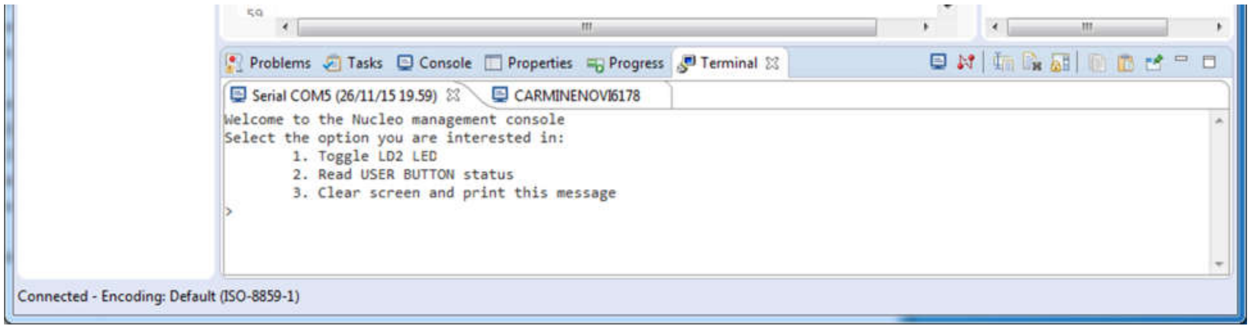
By default, the Terminal pane opens a new command line prompt. Click on the **Open a Terminal** icon (the one circled in red in **Figure 8**). In the **Launch Terminal** dialog (see **Figure 9**) select **Serial Terminal** as terminal type, and then select the COM Port corresponding to the Nucleo VCP, and 38400Bps as *Baud Rate*. Click on the **OK** button.



Choose terminal: Serial Terminal
Port: COM5
Baud Rate: 38400
Data Bits: 8
Parity: None
Stop Bits: 1
Flow Control: None
Timeout (sec): 5
Encoding: Default (ISO-8859-1)

Figure 9: Terminal type selection dialog

Now you can reset the Nucleo. The management console we have programmed using the `HAL_UART` library should appear in the serial console window, as shown in **Figure 10**.



```
Welcome to the Nucleo management console
Select the option you are interested in:
        1. Toggle LD2 LED
        2. Read USER BUTTON status
        3. Clear screen and print this message
>
```

Connected - Encoding: Default (ISO-8859-1)

Figure 10: The Nucleo management console shown in the terminal view

8.3.2 Installing a Serial Console in Linux and MacOS X

Unfortunately, installing the RXTX plug-in on Linux and MacOS X is not a trivial task. For this reason we will go another way.

The first step is installing the `kermit` tool. To install it in Linux, type at command line:

```
$ sudo apt-get install ckermit
```

while to install it in MacOS X type:

```
$ sudo port install kermit
```

Once, the installation has been completed, switch to Eclipse and go to **Help->Eclipse Marketplace…**. In the **Find** text box write “tcf”. After a while, the **TM Terminal** plug-in should appear, as shown in **Figure 7**. Click on the **Install** button and follow the instructions. Restart Eclipse when requested.

To open the Terminal panel you can simply press **Ctrl+Alt+T**, or you can go to **Window->Show View->Other…** menu and search for **Terminal** view. The command line prompt appears. Before we can connect to the Nucleo VCP, we have to identify the corresponding device under the `/dev` path. Usually, on UNIX like systems the USB serial devices are mapped with a device name similar to `/dev/tty.usbmodem1a1213`. Take a look to your `/dev` folder. Once you grab the device filename, you can launch the `kermit` tool and execute the commands shown below at the `kermit` console:

```
$ kermit
C-Kermit 9.0.302 OPEN SOURCE:, 20 Aug 2011, for Mac OS X 10.9 (64-bit)
 Copyright (C) 1985, 2011,
  Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
(/Users/cnoviello/) C-Kermit>set line /dev/tty.usbmodem1a1213
(/Users/cnoviello/) C-Kermit>set speed 38400
/dev/tty.usbmodem1a1213, 38400 bps
(/Users/cnoviello/) C-Kermit>set carrier-watch off
(/Users/cnoviello/) C-Kermit>c
Connecting to /dev/tty.usbmodem1a1213, speed 38400
 Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
```

To avoid retyping the above commands every time you launch `kermit`, you can create a file named ~/`.kermrc` inside your home directory, and put inside it the above commands. `kermit` will load those commands automatically when it is executed.

Now you can reset the Nucleo. The management console we have programmed using the `HAL_UART` library should appear in the serial console window, as shown in **Figure 10**.

8.4 UART Communication in *Interrupt Mode*

Let us consider again the first example of this chapter. What's wrong with it? Since our firmware is all committed to this simple task, there is nothing wrong by using the UART in polling mode. The MCU is essentially blocked waiting for the user input (the `HAL_MAX_DELAY` timeout value blocks the `HAL_UART_Receive()` until one char is sent over the UART). But what if our firmware has to accomplish other cpu-intensive activities in real-time?

Suppose to rearrange the `main()` from the first example in the following way:

```
  while (1)  {
    opt = readUserInput();
    processUserInput(opt);
    if(opt == 3)
      goto printMessage;

    performCriticalTasks();
  }
```

In this case we cannot block the execution of function `processUserInput()` waiting for the user choice, but we have to specify a much more short timeout value to the `HAL_UART_Receive()` function, otherwise `performCriticalTasks()` is never executed. However, this could cause the loss of important data coming from the UART peripheral (remember that the UART interface has a one byte wide buffer).

To address this issue the HAL offers another way to exchange data over a UART peripheral: the *interrupt mode*. To use this mode, we have to accomplish the following tasks:

- To enable the `USARTx_IRQn` interrupt and to implement the corresponding `USARTx_IRQHandler()` ISR.
- To call `HAL_UART_IRQHandler()` inside the `USARTx_IRQHandler()`: this will perform all activities related to management of interrupts generated by the UART peripheral[12].
- To use the functions `HAL_UART_Transmit_IT()` and `HAL_UART_Receive_IT()` to exchange data over the UART. These functions also enables the *interrupt mode* of the UART peripheral: in this way the peripheral will assert the corresponding line in the NVIC controller so that the ISR is raised when an event occurs.
- To rearrange our application code to deal with asynchronous events.

Before we rearrange the code from the first example, it is best to take a look to the available UART interrupts and to the way HAL routines are designed.

[12]If we use CubeMX to enable the `USARTx_IRQn` from the NVIC configuration section (as shown in Chapter 7), it will automatically place the call to the `HAL_UART_IRQHandler()` from the ISR.

8.4.1 UART Related Interrupts

Every STM32 USART peripheral provides the interrupts listed in **Table 6**. These interrupts include both IRQs related to data transmission and to communication errors. They can be divided in two groups:

- *IRQs generated during transmission*: Transmission Complete, Clear to Send or Transmit Data Register empty interrupt.
- *IRQs generated while receiving*: Idle Line detection, Overrun error, Receive Data register not empty, Parity error, LIN break detection, Noise Flag (only in multi buffer communication) and Framing Error (only in multi buffer communication).

Table 6: The list of USART related interrupts

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 (third column of **Table 6**). However, STM32 MCUs are designed so that all these IRQs are bound to just one ISR for every USART peripheral (see **Figure 11**[13]). For example, the USART2 defines only the `USART2_-IRQn` as IRQ for all interrupts generated by this peripheral. It is up to the user code to analyze the corresponding *Event Flag* to infer which interrupt has generated the request.

[13]The Figure 9s taken from the STM32F030 Reference Manual (RM0390).

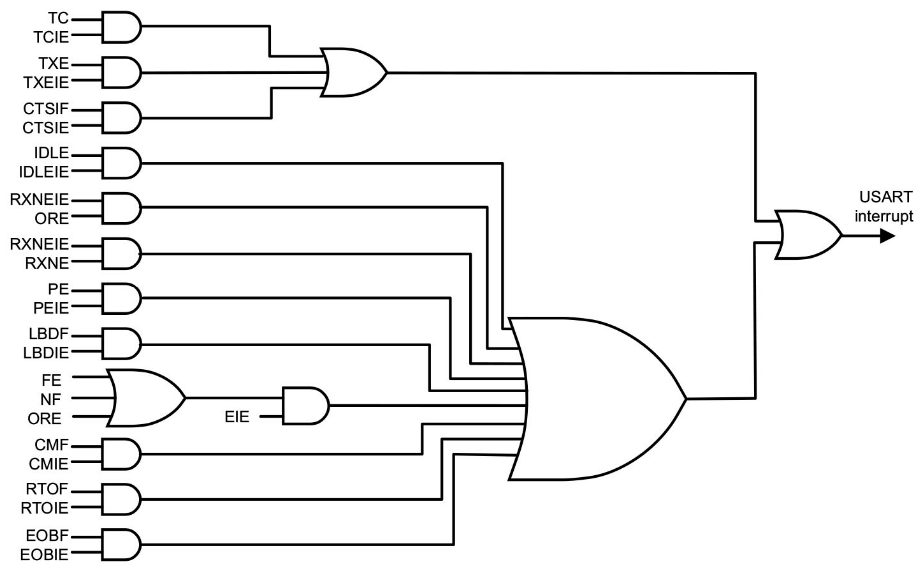


Figure 11: How the USART interrupt events are connected to the same interrupt vector

The CubeHAL is designed to automatically do this job for us. The 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 we want to transmit. 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[14].

To transmit a sequence of bytes in interrupt mode, the HAL defines the function:

[14]This is the reason why transferring a sequence of bytes in interrupt mode is not a smart thing when the communication speed is too high, or when we have to transfer a great amount of data very often. Since the transmission of each byte happens quickly, the CPU will be congested by the interrupts generated by the UART for every byte transmitted. For continuous transmission of great sequences of bytes at high speed is best to use the DMA mode, as we will see in the next chapter.

```
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart,
                                       uint8_t *pData, uint16_t Size);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: it is the pointer to an array, with a length equal to the `Size` parameter, containing the sequence of bytes we are going to transmit; the function will not block waiting for the data transmission, and it will pass the control to the main flow as soon as it completes to configure the UART.

Conversely, to receive a sequence of bytes over the USART in interrupt mode the HAL provides the function:

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData, uint16_t Size);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: it is the pointer to an array, with a length at lest equal to the `Size` parameter, containing the sequence of bytes we are going to receive. The function will not block waiting for the data reception, and it will pass the control to the main flow as soon as it completes to configure the UART.

Now we can proceed rearranging the first example.

Filename: `src/main-ex2.c`

```
  /* Enable USART2 interrupt */
  HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(USART2_IRQn);

printMessage:
  printWelcomeMessage();

  while (1)  {
    opt = readUserInput();
    if(opt > 0) {
      processUserInput(opt);
      if(opt == 3)
        goto printMessage;
    }
    performCriticalTasks();
  }
}

int8_t readUserInput(void) {
  int8_t retVal = -1;

  if(UartReady == SET) {
    UartReady = RESET;
    HAL_UART_Receive_IT(&huart2, (uint8_t*)readBuf, 1);
    retVal = atoi(readBuf);
  }
  return retVal;
}


uint8_t processUserInput(int8_t opt) {
  char msg[30];

  if(!(opt >=1 && opt <= 3))
    return 0;

  sprintf(msg, "%d", opt);
  HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
  HAL_UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT), HAL_MAX_DELAY);

  switch(opt) {
  case 1:
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
    break;
  case 2:
    sprintf(msg, "\r\nUSER BUTTON status: %s",
      HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    break;
  case 3:
    return 2;
  };

  return 1;
}

```

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle) {
 /* Set transmission flag: transfer complete*/
 UartReady = SET;
}
```

As you can see in the above code, the first step is to enable the `USART2_IRQn` and to assign it a priority[15]. Next, we define the corresponding ISR inside the `stm32xxxx_it.c` file (not shown here) and we add the call to the `HAL_UART_IRQHandler()` function inside it. The remaining part of the example file is all about restructuring the `readUserInput()` and `processUserInput()` functions to deal with asynchronous events.

The function `readUserInput()` now checks for the value of the global variable `UartReady`. If it is equal to `SET`, it means that the user has sent a char to the management console. This character is contained inside the global array `readBuf`. The function then calls the `HAL_UART_Receive_IT()` to receive another character in interrupt mode. When `readUserInput()` returns a value greater than `0`, the function `processUserInput()` is called. Finally, the function `HAL_UART_RxCpltCallback()`, which is automatically called by the HAL when one byte is received, is defined: it simply sets the global `UartReady` variable, which in turn is used by the `readUserInput()` as seen before.

It is important to clarify that the function `HAL_UART_RxCpltCallback()` is called only when all the bytes specified with the `Size` parameter, passed to the `HAL_UART_Receive_IT()` function, are received.

What about the `HAL_UART_Transmit_IT()` function? It works in a way similar to the `HAL_UART_-Receive_IT()`: it transfers the next byte in the array every time the *Transmit Data Register Empty*(TXE) interrupt is generated. However, special care must be taken when calling it multiple times. Since the function returns the control to the caller as soon as it finishes to setup the UART, a subsequent call of the same function will fail and it will return the `HAL_BUSY` value.

Suppose to rearrange the function printWelcomeMessage() from the previous example in the following way:

```
void printWelcomeMessage(void) {
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)PROMPT, strlen(PROMPT));
}
```

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.

[15]The example is designed for an STM32F4. Please, refer to the book examples for your specific Nucleo.

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

```
void printWelcomeMessage(void) {
        char *strings[] = {"\033[0;0H", "\033[2J", WELCOME_MSG, MAIN_MENU, PROMPT};

        for (uint8_t i = 0; i < 5; i++) {
          HAL_UART_Transmit_IT(&huart2, (uint8_t*)strings[i], strlen(strings[i]));
          while (HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX ||
                 HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX_RX);
        }
}
```

Here we transfer each string using the `HAL_UART_Transmit_IT()` but, before we transfer the next string, we wait to the transmission completion. However, this is just a variant of the `HAL_UART_-Transmit()`, since we have a busy wait for every UART transfer.

A more elegant and performing solution is to use a temporary memory area where to store the byte sequences and to let the ISR to execute the transfer. A queue is the best options to handle FIFO events. There are several ways to implement a queue, both using static and dynamic data structure. If we decide to implement a queue with a predefined area of memory, a circular buffer is the data structure suitable for this kind of applications.

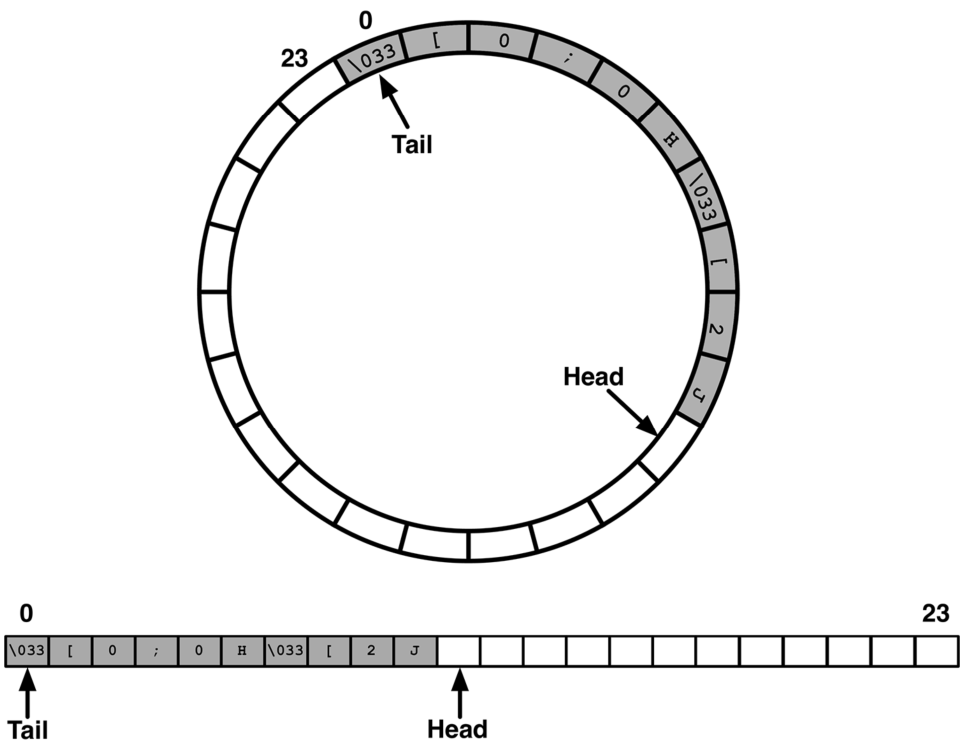


Figure 12: A circular buffer implemented using an array and two pointers

A circular buffer is nothing more than an array with a fixed size where two pointers are used to keep track of the *head* and the *tail* of data that still needs to be processed. In a circular buffer, the first and the last position of the array are seen "contiguous" (see **Figure 12**). This is the reason why this data structure is called *circular*. Circular buffers have an important feature too: unless our application has up to two concurrent execution streams (in our case, the main flow that places chars inside the buffer and the ISR routine that sends these chars over the UART), they are intrinsically thread safe, since the "consumer" thread (the ISR in our case) will update only the *tail* pointer and the producer (the main flow) will update only the *head* one.

Circular buffers can be implemented in several ways. Some of them are faster, others are more safe (that is, they add an extra overhead ensuring that we handle the buffer content correctly). You will find a simple and quite fast implementation in the book examples. Explaining how it is coded is outside the scope of this book.

Using a circular buffer, we can define a new UART transmit function in the following way:

```
uint8_t UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t len) {
        if(HAL_UART_Transmit_IT(huart, pData, len) != HAL_OK) {
                if(RingBuffer_Write(&txBuf, pData, len) != RING_BUFFER_OK)
                        return 0;
        }
        return 1;
}
```

The function does just two things: it tries to send the buffer over the UART in interrupt mode; if the `HAL_UART_Transmit_IT()` function fails (which means that the UART is already transmitting another message), then the byte sequence is placed inside a circular buffer.
It is up to the `HAL_UART_TxCpltCallback()` to check for pending bytes inside the circular buffer:

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
        if(RingBuffer_GetDataLength(&txBuf) > 0) {
                RingBuffer_Read(&txBuf, &txData, 1);
                HAL_UART_Transmit_IT(huart, &txData, 1);
        }
}
```

The `RingBuffer_Read()` it is not really fast as it could be with a more performant implementation. For some real world situations, the whole overhead of the `HAL_UART_-TxCpltCallback()` routine (that is called from the ISR routine) could be too high. If this is your case, you can consider to create a function like the following one:

```
void processPendingTXTransfers(UART_HandleTypeDef *huart) {
   if(RingBuffer_GetDataLength(&txBuf) > 0) {
       RingBuffer_Read(&txBuf, &txData, 1);
       HAL_UART_Transmit_IT(huart, &txData, 1);
    }
}
```

Then, you could call this function from the main application code or in a lower privileged task if you are using an RTOS.

8.5 Error Management

When dealing with external communications, the error management is an aspect that we must strongly take in consideration. An STM32 UART peripheral offers some error flags related to communication errors. Moreover, it is possible to enable a corresponding interrupt to be noticed when the error occurs.

The CubeHAL is designed to automatically detect error conditions, and to warn us about them. We only need to implement the `HAL_UART_ErrorCallback()` function inside our application code. The `HAL_UART_IRQHandler()` will automatically invoke it in case an error occurs. To understand which error has been occurred, we can check the value of the `UART_HandleTypeDef->ErrorCode` field. The list of error codes is reported in **Table 7**.

Table 7: List of `UART_HandleTypeDef->ErrorCode` possible values

UART Error Code	Description
`HAL_UART_ERROR_NONE`	No error occurred
`HAL_UART_ERROR_PE`	Parity check error
`HAL_UART_ERROR_NE`	Noise error
`HAL_UART_ERROR_FE`	Framing error
`HAL_UART_ERROR_ORE`	Overrun error
`HAL_UART_ERROR_DMA`	DMA Transfer error

The `HAL_UART_IRQHandler()` is designed so that we should not care with the implementation details of UART error management. The HAL code will automatically perform all needed steps to handle the error (like clearing event flags, pending bit and so on), leaving to us the responsibility to handle the error at application level (for example, we may ask to the other peer to resend a corrupted frame).