

## Description of STM32 resources. Exemplification by test software

### 1. Test software explanation: STM32 microcontroller resources.

This paragraph explains the operation of the test software, which was written explicitly to illustrate how to access the microcontroller resources.

The software uses ST's open-source library called HAL: *Hardware Abstraction Layer*. All functions starting with HAL access a physical resource (e.g. an I/O port) through a method transparent to the programmer.

In main.c and main.h files we observe many sections clearly marked by BEGIN / END comments, for example:

```
/* Private variables-----*/
TIM_HandleTypeDef htim2;           // <- this part is deleted when
UART_HandleTypeDef huart1;         // when you generate a new project with
                                   STM32CubeMX

/* USER CODE BEGIN PV */

uint8_t uart_buf[1];               // <- this part is kept
/* USER CODE END PV */

/* Private function prototypes -----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

you will notice the comments `/* User Code Begin */` respectively `/* User Code End */` and an abbreviation such as PV= Private Variables or a number. It is important that the portions of the code that you are going to write are exclusively between these User... Begin and End comments, like for example the portion marked above (`uart_buf[1]` statement). The portions of code that are NOT in the "User" sections are those generated by STM32CubeMX, for example in the figure above are "private variables" that are related to Timer2 and UART1 because these 2 peripherals were initialized when STM32CubeMX was called.

**Rationale:** STM32CubeMX generates code for the compiler in the form of a directory structure, .C files, .h etc. But it also saves the STM32CubeMX project as Test1.ioc. You can later reopen this project in STM32CubeMX, make changes to the initialization of the peripherals, and regenerate the code. All lines you have written in between the User... Begin and End comments will be preserved, the rest will be overwritten when regenerating the code !

The test software has the following functions:

- blinks the tricolor LED using timer interrupt 2

- reads using the receive serial interrupt (Rx) the serial characters on the UART1 serial port
- transmits the next value for each serial character received, except "?" which transmits the version number (it is useful for any software to include a method to query the version, as at some point in development you may be unsure which version is running. Include this functionality in your software !).
- In the loop, it uses the principle of state automation to change the blinking speed of the LED at the press of a button. This button is read using polling, not interrupt

The following will explain the hand-written portions of code in different sections.

## Explanations main.h

This file is for definitions with the `#define` directive. It defines the version number as well as the state names of the state automaton. They start with `SM_` (for *State Machine*).

## Main.c explanations

PV (*private variables*) code area: define the required (global) user variables.

Code area 0: include support for the `printf()` function redirected to the serial port. The macro `io_putchar` macro and the `_write()` function are used internally by `printf()`. The actual transmission on the UART1 serial port is done with the `HAL_UART_Transmit()` function.

Writing/reading from the serial port: the `HAL_UART_RxCpltCallback()` interrupt is defined which means "Receive Complete" i.e. it is executed when a serial character has been received. These functions, sometimes called ISR - *Interrupt Service Routines*, are also called *callback* because they occur asynchronously with the program running, after the English expression "*I will call you back*" - it is assumed that the "call" can come at any time, which is essentially the basic idea of an interrupt. This function processes the received character in the variable `uart_buf[0]` (of size 1 as only one character is processed at a time). It handles the "?" character separately and any other character.

Notice that the `printf()` function is used to send a string, but for the single-character situation, it was chosen to write directly using the `HAL_UART_Transmit()` function - the `printf()` function would have called this function anyway.

The last line of the `HAL_UART_Receive_IT()` interrupt routine resets the interrupt to receive the next character. These functions are among the many functions in the code automatically generated as a source file by STM32CubeMX. For example, right click on `HAL_UART_Receive_IT()` and select *Go To Definition (F12)*. This will open the source file and you can study the definition, parameters, etc.

Define the interrupt for Timer 2 `HAL_TIM_PeriodElapsedCallback()` which is called when the defined "time expires" - see section 6 of this document for how to calculate the time. In essence, the timer parameters were calculated to generate one interrupt every 10ms (so 100 interrupts/s). One interrupt= 1 *timer tick*. So any time related action can be done with 10ms resolution (this value was chosen arbitrarily, you can program the timer with whatever values you want). Using a *switch* you do different actions - turning a color on or off - at different chosen time instants to "cycle" the colors. Because of the switch, the time is not affected by other actions of the program, for example notice that the LED still changes color even if you hold down the USER button, which in `main()` you will see keeps the CPU busy in a `while()` loop where nothing else is happening.

Writing a 0/1 value to a port pin is done using a function `HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 0)`, for example in this case port PA8 is written with value 0. You must use these definitions, for example

for example if you write 8 instead of `GPIO_PIN_8` you won't get the desired result: go to the definition and see that `GPIO_PIN_8` is defined as  $1 \ll 8$  in binary so 10000000, while 8 is 1000.

Code area 2: the `HAL_UART_Receive_IT()` and `HAL_TIM_Base_Start_IT()` functions enable the serial receive and timer2 interrupts, respectively. As seen, the serial interrupt must be reactivated after each character, while the timer remains active.

Code area 3, in the main `while()` loop (the infinite loop of the program): similar to the write function, the read state 0/1 function of the `HAL_GPIO_ReadPin()` `pin` is used to read the PC13 pin to which the USER button is connected.

Notice in the schematic that the pushbutton makes connection to ground, so to 0, so it is checking state 0 not state 1 (called according to the function definition, `GPIO_PIN_RESET`). When the pushbutton is released, the pin is not connected to anything, so the default state would be High Z (high impedance). However, state 1 is obtained because when initializing with STM32CubeMX, *the pull-up* resistor for pin PC13 was explicitly enabled. This resistor "pulls into 1" the pin in the absence of any other connection.

Note also that this reading is done by polling, not by interrupting, so it is not very time efficient.

*The button reading is done 3 times for 2 reasons:*

- first of all, the human press time is very long compared to the loop speed of the uP (which runs at 72MHz, so for a RISC processor an instruction takes about  $1/72\text{MHz} = 13\text{ns}$ ). Without precautions, a press would be read thousands, maybe millions of times.
- Secondly, *debouncing* must be done, i.e. *contact bouncing* must be prevented, which means that when closing or opening contacts, the mechanical part of the pushbutton can generate multiple, very short parasitic closures and openings due to microscopic imperfections in the touching surfaces.

These principles are implemented as follows:

- the state of the button is read.
- If it is 0 (pressed), wait 25ms and read it again. It follows that a connection shorter than 25ms, parasitic, will have no effect, thereby fulfilling the *debouncing* function
- nothing is done (waiting in a `while()` loop with no statements) until the button is *released*. This behavior is similar to the way the mouse button is programmed in Windows, the action is executed on release, not on the first click.

The Finite State Machine principle: the rest of the `while()` loop is processed using this principle which is very useful in this case, being an infinite loop from which the program never exits, so we need to know what state we are in. Three states have been defined for the example, an initial state which is entered by default and exited after initialization operations, and two stable states between which *it will alternate* on repeated button presses. A `User_B_Pressed` flag is used, which is set to 1 when the press is *detected*, and to 0 when the press *is processed* by toggling between states. In this way, a press performs only one action.

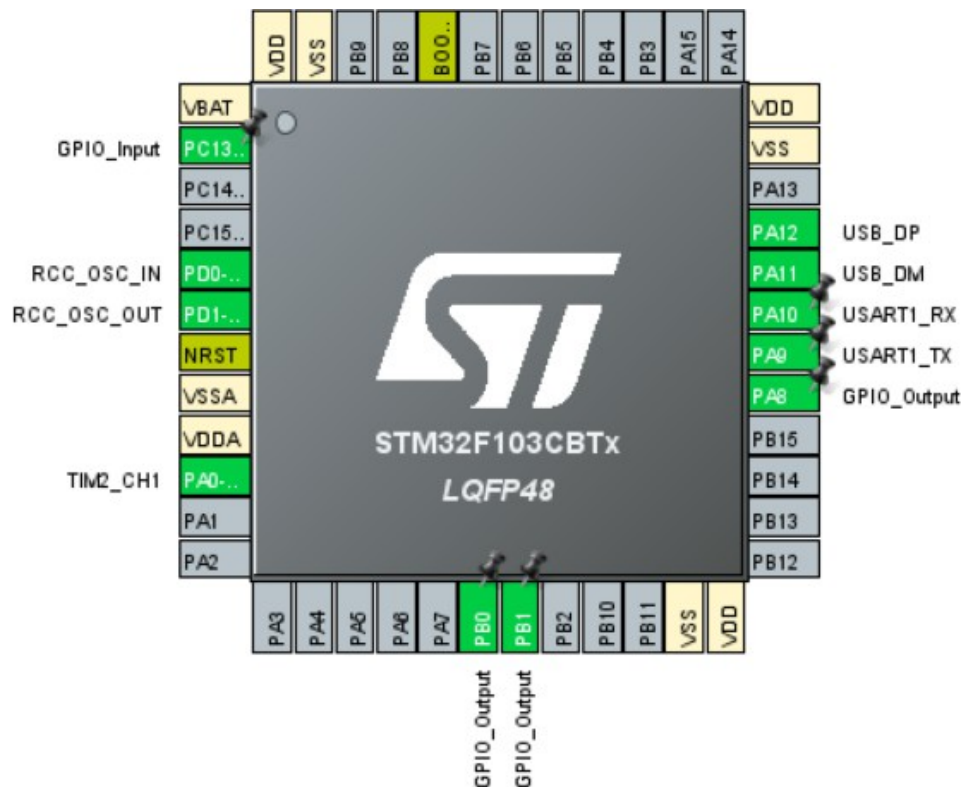
In this case, the action is to rewrite the value of the timer register by either 100 (the value calculated to have a timer period of 10ms) or 300. In the latter case, all timers dependent on timer 2 will be 3 times longer.

This action has been given as an example, to illustrate *the redefinition* of some parameters that are normally automatically written when the STM32CubeMX generates code. The variable `htim2.Init.Period` and the function `HAL_TIM_Base_Init(&htim2)` are taken from the part of the code after *the while()* which contains automatically generated functions. The initialization part of timer 2 is around line 295. This way you can manually redefine everything you have defined using STM32CubeMX, e.g. baud rate, INPUT/OUTPUT direction of a pin, pull-up resistor, etc.

## 2. Generating the software initialization part using STM32CubeMX

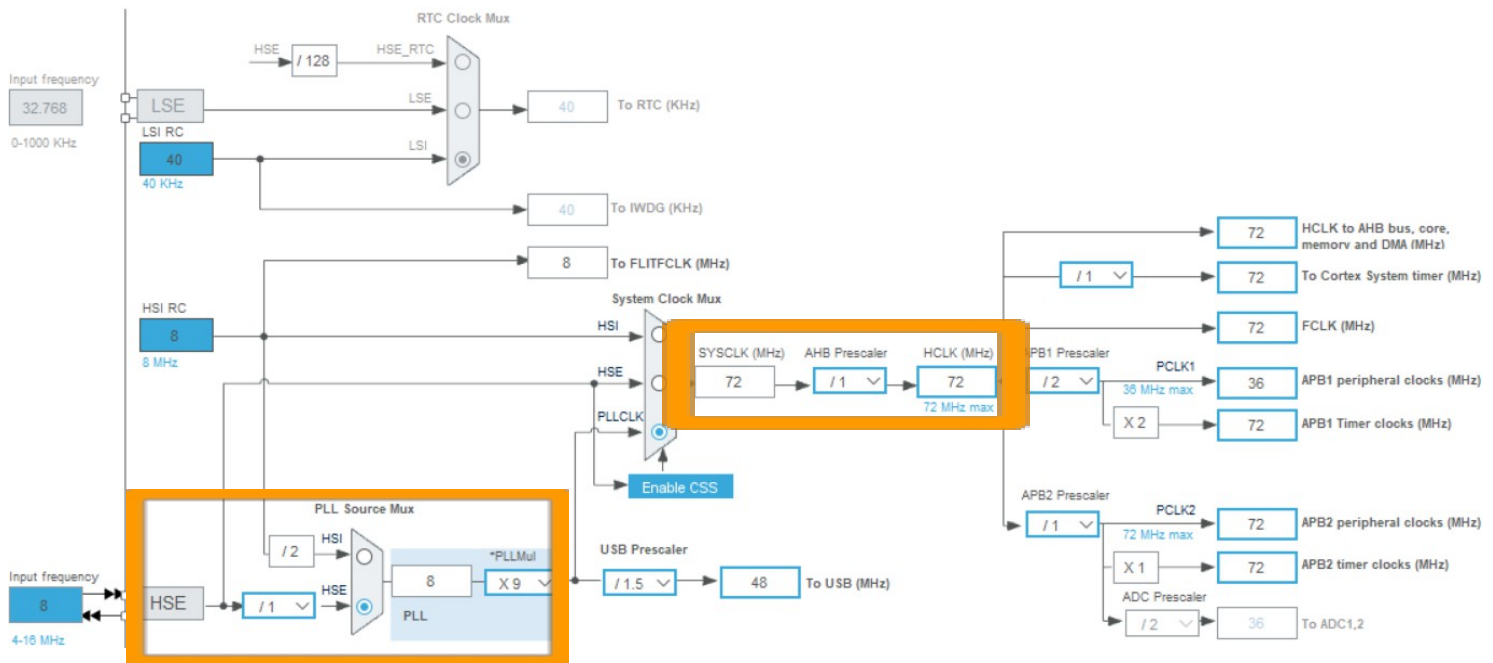
Any microcontroller software needs to initialize various registers that configure the internal devices and peripherals (interrupts, serial ports, ADC, etc). For the STM32 family we recommend the STM32CubeMX (already installed together with the toolchain) which is provided by the chip manufacturer itself. The whole initialization part of the test software, more precisely the whole directory structure and source files including libraries, was automatically generated using STM32CubeMX. The STM32 family comprises hundreds of chips, with a wide variety of peripheral combinations supported. The chip type is selected at the start.

### I/O pin initialization



With the schematic in front of you, click on the pins and initialize them according to how they are used. On the general purpose input/output pins you define the direction (*input/output*). On the PC13 input pin the pull-up resistor is activated.

## Clock initialization



In "Clock configuration" the options in the figure above are selected. Meaning:

**HSE** = High Speed External [Clock] = the external quartz oscillator we use. Set to 8 MHz as this is the value of the quartz bonded to the board.

The frequencies can be:

- *divide* (divider blocks /1, /2 etc) or
- *multiply* (**PLL** blocks noted x2 for example = multiply by 2); PLL=Phase Locked Loop, is a circuit that allows to increase a frequency by synchronizing an oscillator with a reference.

**MUX** = Multiplexer = selection of one of 2 or more clock inputs. For our board, we use an external quartz of 8MHz, configure the input multiplexer to choose **HSE**, not **HSI** (High Speed Internal, an internal oscillator without quartz so much more inaccurate), then with a PLL x9 we get to the frequency SYSCLK=8MHz x 9 = 72MHz. We choose this value according to the indication that this is the maximum supported by this processor model, in order to maximize program performance. This large value can be divided by various optional dividers (prescalers) for driving slower peripherals.

Also in the RCC section the outer-quartz HSE is selected:

RCC Mode and Configuration

Mode	
High Speed Clock (HSE)	Crystal/Ceramic Resonator
Low Speed Clock (LSE)	Disable
<input checked="" type="checkbox"/> Master Clock Output	

## Timer Configuration2

Select options as in the figure: Timers= TIM2, Clock Source=Internal Clock, Channel 1 = Output compare CH1, Prescaler = 7199, Counter Period (Autoreload) = 100.

Frequency calculation: the internal frequency of 72MHz divided by 7200 via the prescaler means the timer clock of  $72000000000\text{Hz} / 7200 = 10000\text{ Hz}$ .

This timer clock, divided by 100, gives us  $10000\text{Hz} / 100 = 100\text{Hz} = 100\text{ ticks/second}$



Same calculation but in terms of time: timer clock of  $f = 10\text{KHz} \rightarrow T = 0.1\text{ms}$ . It follows that a counter period lasts  $100 * 0.1\text{ms} = 10\text{ms}$ . So in one second there will be  $1000\text{ms} / 10\text{ms} = 100 \text{ ticks / second}$  (a tick is a term used for a timer event, similar to a "ticking" of a clock).

There will therefore be 100 timer interrupts per second (this is a value chosen as an example configuration of Timer 2; we have no strict physical requirement to choose 100 ticks/second).

For the prescaler, since the value 0 (which corresponds to no division) can also be set, it follows that to divide by  $K$  the value  $K-1 = 7199$  must be set.

Note that we could also get 100 ticks/second from other combinations, for example prescaler of 3600 and Counter Period =200.

The screenshot shows the STM32CubeMX Pinout & Configuration window. On the left, the 'Timers' category is expanded, and TIM2 is selected. The main panel shows the 'TIM2 Mode and Configuration' settings. Under 'Mode', 'Slave Mode' is set to 'Disable', 'Trigger Source' is 'Disable', 'Clock Source' is 'Internal Clock', and 'Channel1' is 'Output Compare CH1'. Under 'Configuration', 'Reset Configuration' is a button, and 'NVIC Settings', 'DMA Settings', 'GPIO Settings', 'Parameter Settings', and 'User Constants' are all checked. The 'Parameter Settings' section is expanded, showing 'Counter Settings' with 'Prescaler (PSC - 16 bits value)' set to 7199, 'Counter Mode' set to 'Up', 'Counter Period (AutoReload ...)' set to 100, 'Internal Clock Division (CKD)' set to 'No Division', and 'auto-reload preload' set to 'Disable'. The 'Trigger Output (TRGO) Parameters' section is also visible.

The timer interrupt "TIM2 Global Interrupt" must also be enabled:

NVIC Settings			
Parameter Settings		User Constants	
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	<input checked="" type="checkbox"/>	0	0

## Serial Port Configuration

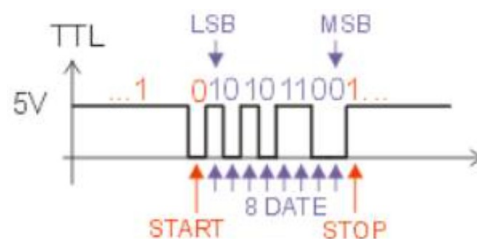
Explanations about the serial port in general can be found in the paragraph "Serial port". Initialize Connectivity→ USART1, Mode Asynchronous, Hardware Flow Control Disable, USART1 Global Interrupt Enabled, 9600 bits/s, 8 Bits, Parity None, Stop Bits=None, Receive and Transmit. I chose USART1 and not another one, because it is the one connected to the USB-TTL connector on the board.

## Code generation

Finally, in the Project Manager section, select Toolchain/IDE= *Makefile*, enter a name for the project (Test1), in Code Generator select "Copy all used libraries into project folder". Then generate using the *Generate Code* button. This will generate the *Makefile* that is used by the *make all* command.

### 3. Serial port

*Universal Asynchronous Receiver Transmitter* (UART) serial ports are used for low-speed communication between microprocessors and various peripherals. The word 'asynchronous' means that the 8 data bits are framed by a start bit, which will always be '0', and a stop bit, which will always be '1' - in synchronous transmissions these do not exist, instead there is a clock signal for synchronization. Basically, on the figure it can be seen that the stop bit "1" lasts much longer than the other bits, but this is because in the absence of data the line is held in logical "1", so after this "1" bit is transmitted the state of the line does not change until a new byte is transmitted, which will start with a new start bit "0". So the stop bit "1" remains active as long as no new character appears.



**Attention!** Using the usual binary notation b7b6b5b4b3b2b1b0, where b7 is MSB and b0 is LSB, bits are transmitted *LSB-first*, not *MSB-first* as would be more intuitive. So after the start bit (0, left) follows the LSB (b0), and the MSB (b7) is adjacent to the stop bit (1, right). The figure corresponds to the data byte 00110101.

The stop and start bits guarantee that a "1→ 0" transition occurs at the beginning of each byte, even if the bytes transmitted represent long strings of "1" or "0". The disadvantage is that for every 8 bits of data 2 bits have to be added, i.e. the transmission efficiency is only 80%.

#### Waveform visualization; serial port troubleshooting

Use an oscilloscope to visualize the waveforms at several points. The black alligator will be connected to the ground of the assembly (GND) and the probe tip to the Rx or Tx pin (PA9 or PA10). When nothing is being transmitted, the oscilloscope should show us a logic "1" (corresponding to the stop bit) i.e. a continuous 5V signal. For 9600bps a character has 10 bits so it takes:

$$10 \text{ bits} - 1/9600 \text{ sec/bit} \approx 1 \text{ ms}$$

In order for a character (10 bits) to occupy all 10 divisions on the screen, we determine  $C_x$ :

$$T_x = N_x C_x \rightarrow C_x = T_x / N_x = 1 \text{ ms} / 10 \text{ div} = 0.1 \text{ ms/div}$$

Pay attention to the oscilloscope synchronization setting (*trigger*) ! The start bit is like in the picture above: a negative edge from 5V to 0V on TTL levels so we set the *trigger*→ *slope*= *falling*

With the processor running the test program, hold down a key on the PC keyboard (after starting the terminal program). Using the test software, the layout should send back the following character: if it receives "a" send "b", etc.

Watch the order of events, and the waveform should be similar to the figure above:

1. from the PC, using the terminal, a character is emitted (RS232) which



2. reaches the processor (uP pin PA10\_RX1 ) - visualize the ASCII code on the oscilloscope and check that it corresponds to the key pressed ! note that the bits of the ASCII code are read inverted, from LSB to MSB.
3. it sends the character with the immediately following code (uP pin PA9\_TX1 ) - visualize this code too !
4. character+1 goes back to the PC.

#### Using a TTL-USB converter; powering from USB

Connect the TTL-USB converter to the pins of connector J7. Pay attention to the meaning of the pins RxD, TxD, GND, usually the TTL-USB converter has pins named according to the DTE convention, so RxD, TxD from USB must be connected to Tx, Rx from uP (*Cross connection*) respectively. By coupling the CH340 to the PC's USB, it detects this and creates a *virtual serial port* numbered COM3 or above. The COM1, COM2 ports are usually reserved for *physical* (not virtual) serial ports which are connected to special 9-pin connectors according to the RS232 standard, but which are no longer provided on recent generation PCs. The CH340 converter driver must also be installed on the PC.

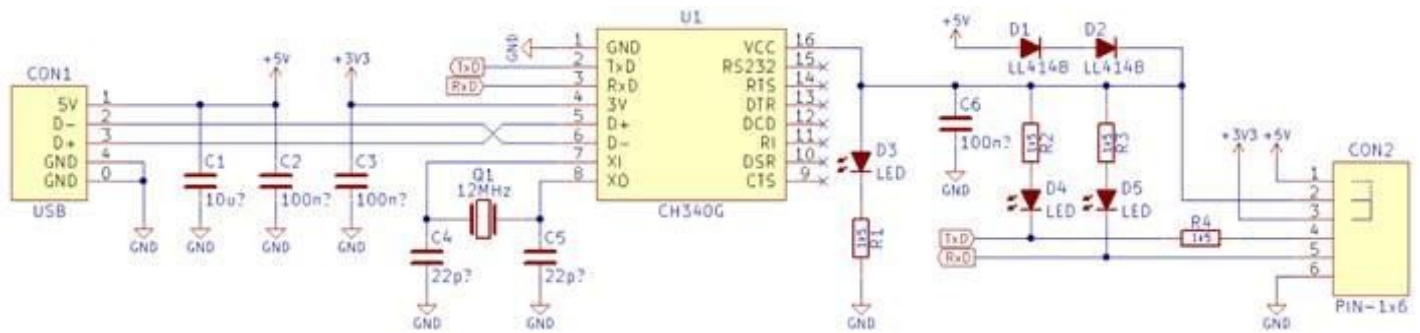
<i>uP</i>	<i>USB</i>
RXD	TXD
TXD	RXD
GND	GND



USB-TTL converters

When the USB-TTL converter is used, the board is powered through it. All 6 pins of the USB-TTL are connected to the J7 connector on the test board (see board schematic). In this case, there is no place for the voltage selection jumper, so note that on the test board schematic the connection between "Vcc" and "3V3" (pins 2,3 of J7 are shorted), instead of the yellow jumper on the converter, as in the figure

above. The converter schematic (based on the CH340 chip) is given in the figure below:



USB-TTL converter schematic with CH340

It can be seen that the reduction of the voltage from 5V to 3.3V is done on the converter by serializing 2 diodes (D1, D2), on each of them falling approx. 0.7V. If the jumper is between pins 1-2 these 2 diodes are short-circuited and the entire 5V voltage from the USB reaches the Vcc pin of the CH340 chip, otherwise a voltage of  $5 - 2 \times 0.7 = 3.6V$  reaches the Vcc pin of the CH340 chip, which approximates the 3.3V voltage.


On the test board, the 5V voltage on pin 1 of J7 (taken directly from USB) is not reduced using diodes, but the U2 stabilizer (LM1117-3.3). Therefore the diodes on the converter do not serve to reduce the supply voltage of the entire STM32 processor board, but only to ensure the logic levels on the converter.

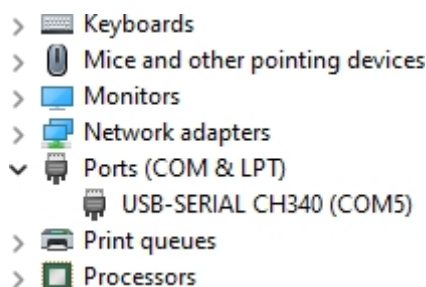
### Visualizing the waveform; troubleshooting serial communication with USB-TTL adapter

UART waveforms will only be tracked on TTL levels. Waveforms on USB output are more complex and are generated/decoded by the CH340 chip.

To see that the CH340 virtual serial port works, proceed as follows:

1) connect the USB-TTL converter to the PC (without also connecting it to the test board), check in Windows → Device Manager under "Ports - COM & LPT" that a serial port, e.g. COM5, is present. If it does not appear, or if it appears under "Other Devices" and/or appears next to a triangle symbol with an exclamation

mark , it may be manual installation of the CH340G driver may be required



2) turn on the terminal and set the appropriate COM port and the "Sent Message Echo" or "Local echo" option or to see what is being transmitted. Without this option the terminal will only display the characters received.

3) Press any key on the terminal; the Rx LED on the USB-TTL adapter should blink, indicating that it is receiving the characters from the terminal. The character should also appear on the display.

4) Place the jumper on the converter directly between the Rx, Tx pins of the converter (4 and 5) a.i. the received characters are turned back; at this point both LEDs (Tx and Rx) should blink and the transmitted characters should appear doubled (the sent and received characters, which are identical, are displayed). Note: this is why

why, in the test software, we chose to transmit *character+1* instead of *character*; this way, we know for sure that it is a new character generated by the uP, not an "echo" of the character received from the PC.

5) remove the jumper from the converter and connect it to the board running the test software. The characters sent should return as *char+1*, as they no longer go directly through the jumper, but through the test board software and the STM32 processor.

To check which ports of the uP are connected to the test board, examine the wiring diagram of the board.

