CDC (Communication Device Class)

Abbreviated **CDC**, the **Communication Device Class** allows you to transfer raw data between a USB host and a USB device as if the USB connection was an old-school serial port (a.k.a. **RS232** or **COM** port). For this reason, it's sometimes referred to as a **Virtual COM Port** or **VCP**.

This is going to be a long hard one. So long in fact that I'm going to have to make it a mini-series. But if you stick with me, by the end of it you'll know how to send and receive data to your STM32 through USB using your computer's COM port API. Courage !

1. Some Context (Yes, More Theory)

You will not find the notion of "COM port" (virtual or otherwise) anywhere in the USB specifications. That's because VCP is just the practice of using the Communication Device Class to *emulate* a COM port, which is a very specific scenario. The USB specifications are much more abstract than that. Let me put everything into context before I start getting practical. I'll try to make it fun.

When you select the USB CDC in STM32CubeIDE's **Middleware** category, you'll notice that it says :

What this means is that we're actually looking at a **subclass** of the CDC, the official name of which is the **PSTN Device Subclass**. Let's unpack all that.

The Communication Device Class specification actually covers, as you should expect, any type of device which main purpose is communication. Duh. This means it covers everything from WiFi USB dongles and USB-Ethernet adapters all the way to old-school telephone line **modems**.

You might be too young to know any of this, or too old to remember, but way back before multicellular life (life with multiple cellphones) evolved on Earth, our main telecommunication network was made of telephone **landlines**. This dates back to a time before computers,

which is why connecting two ends of a call required **human middleware** in the form of sexy switchboard operators. You know, like in the John Wick movies :

Network appliances in a datacenter. Note the standardized high-heels.

Needless to say, this wasn't a sustainable approach to networking. For one thing, there just isn't a sufficient supply of sexy tattooed ladies who also enjoy the tedium of plugging and unplugging patch cables. So the process was eventually automated and became what professionals now call the **PSTN**, or **Public Switched Telephone Network**.

The PSTN allows you to dial any phone on Earth from any other phone on Earth through a set of protocols which has evolved over the ages but still uses phone numbers as its addressing scheme. It's also the first network that allowed computers to talk to each other using so-called dial-up modems. Originally (in the 1970's) a modem looked like this :

You can tell the age from the futuristic character font on the device. What you'd do is pick-up your good old phone and manually dial the number of whoever's computer you wanted to connect to. The person on the other end would answer. You'd exchange politenesses and talk about the weather for a moment as a rudimentary form of

authentication protocol, then you'd both strap your phones' handsets to your modems' acoustic couplers so *they* could talk. Which they did by modulating data into mind-altering screeching noises.

That got tiresome real fast. Plus, it wasn't exactly something you could miniaturize and fit into a MacBook Air. So we also automated all this and got rid of the handset interface (who needs humans anyway ?). This led to the dial-up modem most people my age discovered when the internet started getting popular in the 90's. Just a box with some LED's on the front and some connectors on the back :

The "line" socket goes to your phone landline, the "phone" socket goes to your regular phone : this allows the modem to cut your phone so it can use the landline in its place. There's a power socket too, of course, and then an **RS232 connector**. Yes, I think you're starting to see where I'm going with this.

Before USB, modems connected to computers using RS232 (COM) ports. The RS232 interface has no inherent formatting of data : you're just shoving bytes through the cable, one at a time. This is enough for simple applications like, well, getting your Arduino to print "Hello World" on your computer's terminal emulator program. However, it's *not* enough if you want to use a modem... because a modem needs to be commanded to do things like dialing a phone number and hanging-up. But how would it know the difference between data and commands ?

That's where a guy called Hayes comes in. His company made some of the first dial-up modems, capable of a blazing-fast 300 bits per second... and to make it possible, he came up with the Hayes command set. It was so good that everyone ended-up using it; and because this world is full of jealous people, they started calling it the **AT command**

set instead. It works like character strings in C : every byte is treated as data except for special values which act as escape sequence and allow you to send commands.

The AT command set is still very much in use today, if only because dial-up modems are still very much a thing. Except now they tend to look like this :

There's still a phone jack on one side, but the RS232 port has been replaced with a USB port, which also powers the modem.

And so, here we have it at last : the **USB Communication Class Subclass Specification for PSTN Devices**, which STM32CubeIDE implements *partially* in the form of the **Virtual COM Port**, describes how to use USB to replace the RS232 interface between a computer and a dial-up modem. This provides the tools for carrying data and AT commands issued by **legacy software** to new USB modems.

In other words : USB wasn't created to replace specific interfaces, so it was never going to have an "RS232 mode", and RS232 isn't exactly a protocol to begin with, it's just loose bytes on a wire. USB was designed to be something new and **legacy-free**. To that end, it specifies all sorts of protocols and mechanisms that can be used to do the same thing as older interfaces but in a very different way.

2. What Is Virtual COM Port, Then ?

Well it's not a standard, for one thing. It's just a concept. You won't find a specification for it.

VCP is any software (usually a **driver**) that acts like an COM port and has the same API but actually transfers data on a *different* type of physical interface than RS232. It's not specific to USB : even before, there was already VCP over Ethernet, allowing you to connect RS232 devices over a network. This is very useful in scenarios where you need to connect a lot more RS232 devices to your computer than you had

physical COM ports for. Or if your RS232 devices are too far from your computer for RS232 cables to be practical.

The concept is to make the programs on your host computer "believe" they are talking to a COM port when in fact the data goes through a USB port to firmware on your STM32 that has nothing in common with a UART. Kinda like Robert Downey Junior in Tropic Thunder :

When you're using USB VCP on an STM32, the programs on your host computer see a COM port (or a TTY if you're into that Linux lifestyle), but your firmware on the **STM32**, however, **will not look like a UART**. That's why I've been boring you with all sorts of information about the USB protocol. USB behaves very differently than RS232. Just to mention one obvious difference : RS232 talks in bytes, USB talks in packets. And USB talks a lot faster. When you open your VCP as a 9,600 baud COM port, data will still be transmitted at 12 Mbit/s on the USB cable.

> I'm putting this in a frame for people who fell asleep while I was talking : I repeat, the STM32 USB CDC library **does not** provide you with simple Arduino-style "Serial.begin", "Serial.write" and "Serial.read" functions. It's much more universal than that because it's meant for people who work for a living; not for hobbyists who need a quick and dirty way to perform "debug printfs". And the reason we're here is to fix that, which requires knowing how USB works in intimate detail.

If you so desire, you can write your own VCP on the STM32 side. This could be a Virtual UART Peripheral or VUP, if you will. This would allow you to easily reuse STM32 code written for a UART. **Bad news**, ST offers no such thing. **Good news**, you probably don't need it in the first place

anyway.

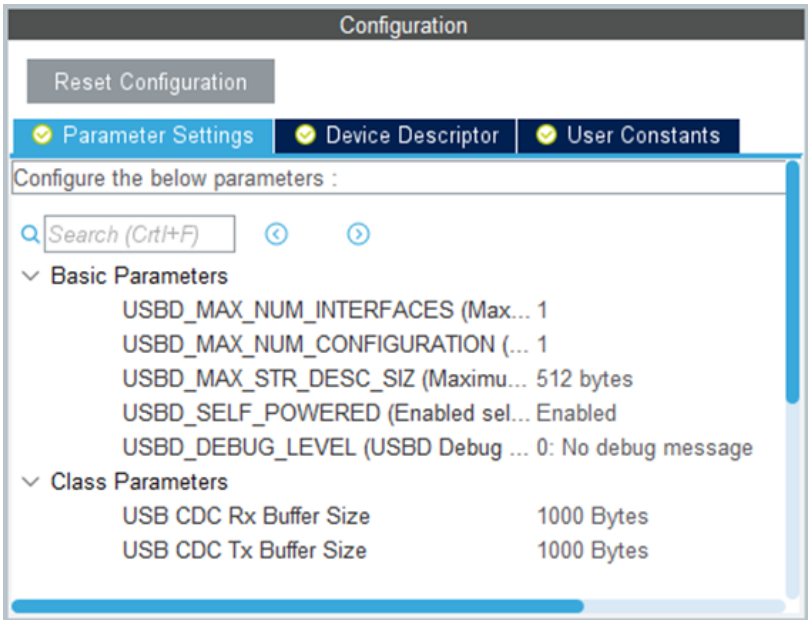
It's all about what you're really trying to achieve. Deep down, you really just want to send and receive data to and from a host computer. VCP means you can use simple COM port libraries and tools on the host side because it's much, much simpler than to code custom USB drivers and applications... but the API on the STM32 side doesn't matter. In fact, because an STM32 has limited processing power, it's best to keep it as simple as possible. It can just be limited to two functions to send and receive a buffer of bytes.

And that's **exactly** what we're going to code on top of the STM32Cube USB Device Library.

3. The Lay Of The Land

Let's assume you've already started an STM32 project in **STM32CubeIDE** and you're looking to add USB VCP support. This brings us back to that USB_DEVICE middleware configuration pane I showed you earlier :

Once you've selected Communication Device Class, the following pane will show up :



You can leave everything unchanged : Cube's code generation for middleware is, at this time, less advanced than for actual peripheral blocks. Anything you may want to change is best changed directly in the source files. Moreover, this will spare you regenerating your project's code in the future which, depending on how rigorous you are, could

delete some of your own code.

After you generate the code, quite a few files will be added to your project. They will be split into two new folders outside the usual "**Core**" and "**Drivers**" folders :

· "**Middlewares**" contains the source code for the ST USB CDC library. This needs to be left untouched. Those are low-level drivers which sit above the USB HAL drivers and handle USB transactions.

· "**USB_DEVICE**" contains the application-specific source code for your instance of the Communication Device Class library. Initially, those files are simply **templates** that you will need to modify to implement your application-specific behavior of the CDC.

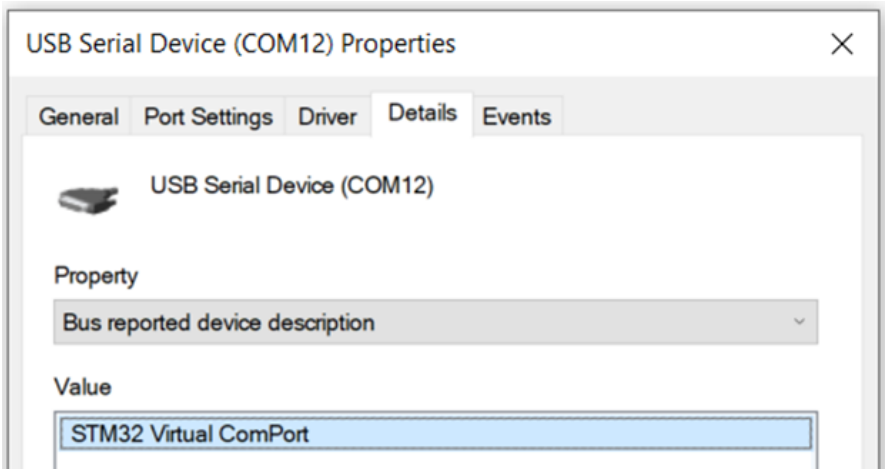
At this point, you can (and should) compile and Flash your project to your STM32 board. If your hardware is wired correctly, then you've just created a USB device. Plug the USB cable from your computer to your board's USB socket.

If you get a message like this one :

Then you may be looking at a hardware problem. It could even be a faulty cable, so make sure to always have a spare. Since this isn't a board-specific page I can't provide more help there.

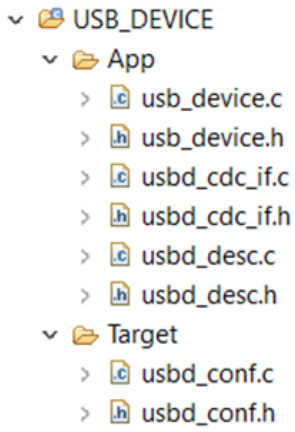
If everything works as it should, you might still need to install a **VCP driver** on your computer if you haven't already, or if it doesn't happen automatically. The official VCP driver for this STM32 middleware can be downloaded from ST website. I don't know which OS you're running so you'll have to Google it yourself.

With the proper driver installed, and assuming your machine runs Windows, you should have a new COM port in your Device Manager and it will look like this :



Now let's get back to those new "template" source files I've mentioned.

Expand your project's **USB_DEVICE** folder. You should get this :



Their exact name and location have changed slightly since the last time ST documented this middleware, which I why I'm never referring to that documentation. They may change again in the future, but once you know what to look for it doesn't take long to find where they went.

First, let me reassure you : you do not need to customize **all** those files :

· **usbd_conf** is a set of macros and callback definitions that are generated by Cube to adapt the library to your firmware. For example, depending on whether or not you're using FreeRTOS this file will contain a different macro for calling the appropriate delay function.

· **usbd_desc** focuses on the USB device descriptor your firmware will provide to the host. It contains things like the string for your device's name. Very few things to change in there.

· **usb_device** is the library's entry point. It contains the STM32Cube-style **MX_USB_DEVICE_INIT** function that will be called from **main** just like the initialization function of any other peripheral block. There's

nothing for you to change in these files.

The bulk of your work is meant to go into **usbd_cdc_if.c** and its associated header file.

4. Nefastor's Easy-VCP Library

I'm not going to sugar-coat this : customizing the CDC library into something that feels like a familiar UART library is **non-trivial**. But because I'm some kind of saint, I've done it for you and created a VCP library. And because I believe in extensive documentation as the first line of defense against bugs, the rest of this page will go into the gory details of how my implementation works. Then in the next episode I'll explain how to integrate it into your own projects.

My library is hosted on GitHub at https://github.com/Nefastor-Online/STM32_VCP

If you're unfamiliar, GitHub is a hosting service for Git repositories. Git is a version control system that you really need to learn. Even (and especially) if you're a week-end coder. It's a great way to keep track of your work's progress. Git repositories support the **tagging** of specific **versions** of your source code. This page pertains to the version tagged **v1.0**.

It is likely that my VCP library will evolve in the future in ways that could make this page obsolete. If you're here to learn, you should checkout that version.

Now, let's go back to **usbd_cdc_if.c** and how to customize it.

5. The CDC Driver Interface

The ST middleware's architecture may seem complicated (well, it **is** complicated) but that is in part because it encapsulates as much of the USB CDC functionality as possible. It only leaves exposed the smallest possible sets of "blanks" for you to fill.

It's called a middleware because it sits in the middle of other stuff :

- Your application code on one side, wants to send and receive arbitrary amounts of data, at arbitrary intervals, over USB.
- The library's USB CDC on the other side, wants to keep the USB host

happy at all times.

Remember : USB is host-centric. The host is always in control of communication. As a result, the ST middleware's priority is to deal with all **USB transactions**.

Some transactions have nothing to do with your application, for example the whole enumeration process. Those are handled internally by the middleware.

Some transactions require data from the device to be sent to the host. From your microcontroller application's point of view, that would the underlying transactions of a "send data" function. But since a device can't talk to the host until the host asks to be talked to, a buffer is necessary.

Some transactions require data from the host to be read by the device. From your microcontroller application's point of view, that would the underlying transactions of a "receive data" function. But of course you never know when that data will arrive. As with a regular UART. And so, a buffer is necessary here too.

By default, the CDC middleware has no idea what data your application wants to send and what it wants to do with incoming data from the host.

In the template source file **usbd_cdc_if.c** you will find the following elements to help you complete the middleware and adapt it to your application :

- **Two buffers** : one for transmitting to the host, one for receiving.

- **Five functions** to handle, among other things, sending and receiving data.

Let's look at them in detail.

6. The Buffers

Back in Cube's graphical configuration tool, you may remember there were two parameters to set the size of the Tx and Rx buffer. Their default size is 1,000 bytes each.

You could be forgiven for deducing that those are circular FIFO's that will let you bridge the gap between the host's timing and that of your STM32 application. But that's not the case. All this parameter does is

set the size for two arrays of bytes in **usbd_cdc_if.c** :

```
1.  /* Define size for the receive and
    transmit buffer over CDC */
2.  /* It's up to user to redefine and/or
    remove those define */
3.  #define APP_RX_DATA_SIZE  1000
4.  #define APP_TX_DATA_SIZE  1000
```

And later :

```
1.  /* Create buffer for reception and
    transmission           */
2.  /* It's up to user to redefine and/or
    remove those define */
3.  /** Received data over USB are stored
    in this buffer      */
4.  uint8_t
    UserRxBufferFS[APP_RX_DATA_SIZE];
5.
6.  /** Data to send over USB CDC are
    stored in this buffer   */
7.  uint8_t
    UserTxBufferFS[APP_TX_DATA_SIZE];
```

Notice the comments in both snippets. Those buffer declarations are just **placeholders**. If you use them as they are, each time the host sends data it'll overwrite the previous data whether you read it or not, because by default the middleware writes incoming data to the start of **UserRxBufferFS**. Things ain't much better on the transmission side.

It's be up to you to turn those two byte arrays into actual buffer mechanisms with any bells and whistles you want... such as circular FIFO management and overflow detection.

And the additional code to do that will go into the **five template functions** provided by the library.

7. The Template Functions

Their names are somewhat self-explanatory, but exactly how they work... isn't. So pay close attention.

The first four functions are actually **callbacks** that the middleware will call whenever a USB transaction requires it. The last function (transmitting data to the host) pertains to an event that only your

application knows the timing of, so it obviously can't be a callback :

· **CDC_Init_FS** : since everything else is encapsulated in other files, this particular initialization function is where you'll do what no other function does : initialize your own buffers.

· **CDC_DeInit_FS** : this function will be called when your STM32 stops USB operations. It's supported differently depending on which STM32 you're using, and that's a story for another time. You can just leave it as is. For now.

· **CDC_Control_FS** : the host may send your STM32 some commands, such as changing the baud rate of the VCP. This function is where you'll handle host commands you want to support and that aren't data transfers. It can be left as is.

· **CDC_Receive_FS** : whenever a data packet is received from the host, it is passed to this function. Therefore, this is where you implement data reception, which means storing it into buffer until your STM32 application reads it.

· **CDC_Transmit_FS** : your STM32 application will use this function to send data to the host. To do that, this function will simply store the data into the middleware's transmit buffer so that it can be sent to the host when the host requests it.

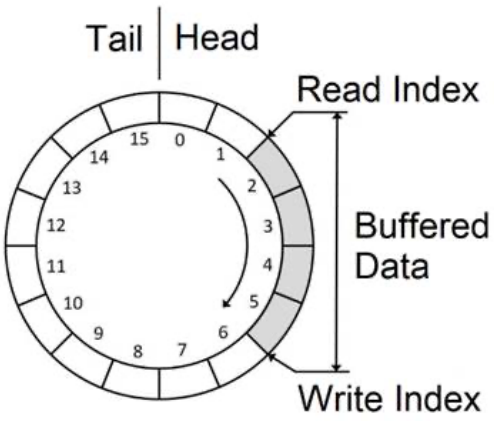
8. Buffer Design

The first aspect that needs to be considered is the design of your **Tx and Rx buffers**.

At this point, all you have is a couple of **byte arrays** graciously provided by the ST code generator. That's like getting a car without an engine. Luckily, we're engineers. That means we can design engines.

If you're unfamiliar with the programming of communications software, the problem we need to solve is one of **conflicting schedules** : software on the host computer and on the STM32 live their own separate lives and do things on their own time. If they were perfectly in sync there would be no need for buffers : they'd just pass each other bytes in real time. But this never happens. On the microcontroller, data can be received from the host or requested by the host at any time. Therefore, buffers are required.

The most efficient type of buffer to solve this problem is the **circular FIFO** (First In, First Out) :



In this scheme, data is written (and later read) all the way to the end of the buffer (the **tail**), and then wraps around back to the start of the buffer (the **head**), creating a pseudo-endless buffer. Obviously, if data is written into this buffer faster than it is read out, it will eventually overflow. Remember : the buffer is here to adapt the **timing** of the communications between two different equipments, but it can't do much about differences in the **rate** of those communications.

Programming a circular FIFO is normally very easy; unfortunately in this case we have an added difficulty : unlike an RS232 COM port, which sends individual bytes, USB sends data as **packets**. And those packets have **variable length**. The full prototype of the **CDC_Receive_FS** function is :

```
1.  static int8_t CDC_Receive_FS(uint8_t*
    Buf, uint32_t *Len);
```

Remember : this is the callback the USB CDC driver will call whenever your STM32 receives data from your host over a virtual COM port. That data will be stored in a temporary buffer (**Buf**) and have a length of **Len** bytes, a value you can't predict. All you know is it's at least 1 byte. It can be as much as the maximum size of a CDC packet, a value that is defined in the middleware's source code as **CDC_DATA_FS_MAX_PACKET_SIZE** and is set to 64.

In the device-to-host direction, we face another complication : the USB middleware's data transmission code is asynchronous. When our code sends data to the host, it will return immediately... but the data may still take a while to go out to the host. During that time, it needs to be kept in the buffer and must not be overwritten with new data.

To that end, I've written a slightly modified FIFO structure. For data storage, it uses the two byte arrays already created by Cube's code

generator. It adds separate write and read indices, and to deal with the aforementioned complications, it adds a third “special” index. The declaration goes into **usbd_cdc_if.h**, between the USER CODE EXPORTED TYPES comments :

```
typedef struct VCP_FIFO_TYPE
{
  uint8_t* data;  // Will point to the Cube-generated Tx or Rx buffer
  int  wr;    // Write index
  int  rd;    // Read index
  int  lb;    // Additional index
} VCP_FIFO;
```

Note : **data** *should always point to an array of bytes with a length that is a multiple of 4 in order to maintain 32-bit alignment.*

You need to declare two instances of this structure in **usbd_cdc_if.c**, between the USER CODE PRIVATE VARIABLES comments :

```
// Circular FIFO to store outgoing data until it can be sent over USB
VCP_FIFO vcp_tx_fifo;

// Circular FIFO to store incoming data from the host over USB
VCP_FIFO vcp_rx_fifo;
```

We’ll look at how these buffers work when we write our custom **transmission** and **reception** functions.

9. Customizing CDC_Init_FS

This is a template function. Right after code generation it only does one thing : tell the middleware’s underlying USB device driver where the application code’s buffers are located. By default, that’s the byte arrays that were also generated by Cube. The function therefore looks like this :

```
static int8_t CDC_Init_FS(void)
{
  /* USER CODE BEGIN 3 */
  /* Set Application Buffers */
  USBD_CDC_SetTxBuffer(&hUsbDeviceFS, UserTxBufferFS, 0);
  USBD_CDC_SetRxBuffer(&hUsbDeviceFS,
```

```
    UserRxBufferFS);
7.    return (USBD_OK);
8.    /* USER CODE END 3 */
9.  }
```

Because I'm an efficient guy, I've decided to reuse Cube's byte arrays as the storage for my own circular FIFO's, so this default code still works. I just needed to add buffer index initialization somewhere in that function, and then it becomes :

```
1.  static int8_t CDC_Init_FS(void)
2.  {
3.    /* USER CODE BEGIN 3 */
4.    // Circular FIFO initializations :
5.    vcp_tx_fifo.data = UserTxBufferFS;
    // Use the buffer generated by Cube
6.    vcp_tx_fifo.wr = 0;
7.    vcp_tx_fifo.rd = 0;
8.    vcp_tx_fifo.lb = 0;
9.    vcp_rx_fifo.data = UserRxBufferFS;
    // Use the buffer generated by Cube
10.   vcp_rx_fifo.wr = 0;
11.   vcp_rx_fifo.rd = 0;
12.   vcp_rx_fifo.lb = 0;
13.
14.   /* Set Application Buffers */
15.   USBD_CDC_SetTxBuffer(&hUsbDeviceFS,
    UserTxBufferFS, 0);
16.   USBD_CDC_SetRxBuffer(&hUsbDeviceFS,
    UserRxBufferFS);
17.   return (USBD_OK);
18.   /* USER CODE END 3 */
19. }
```

Easy peasy lemon squeezy.

10. Customizing CDC_Receive_FS

This template function will be called by the middleware's underlying USB device driver every time it receives a VCP data packet from the host and stores it in the buffer you've setup. You're meant to customize this function to retrieve the fresh data from the buffer and use it.

The function's prototype is :

```
1.  static int8_t CDC_Receive_FS(uint8_t*
```

```
Buf, uint32_t *Len);
```

Buf tells you where the freshly-received data has been stored by the middleware, and **Len** tells you how many bytes that is. The intent is obviously to let you copy that data away and into your own buffer.

But we're going to be smarter and more efficient. Remember, in **CDC_Init_FS** we've told the middleware to use our own FIFO as buffers. This way, the middleware writes the incoming data directly into our **vcp_rx_fifo**, saving us the trouble. It won't automatically update our FIFO indices, however : that's on us. But the logic is simple :

We've received **Len** bytes into our FIFO, therefore its write index (**wr**) must be incremented by **Len**.

The next time this function gets called, Len may be anywhere from 1 to CDC_DATA_FS_MAX_PACKET_SIZE. And the middleware doesn't know how to use a circular buffer, therefore we need to make sure that the new value of wr leaves at least CDC_DATA_FS_MAX_PACKET_SIZE bytes until the end of our FIFO's byte array (the **tail**).

If that's not the case, then there's a risk the next packet received from the host will exceed the boundaries of the FIFO and data bytes will overwrite other variables. The only solution is to wrap-around : **wr** is set to zero (**head**) but not before we save its value to the **lb** index.

Why, you ask ? Simple : because each incoming packet may have a different length, there's no way to know where **wr** will end-up before it's necessary to wrap-around. As a result, the tail of the FIFO may end with some unused space that doesn't contain valid data. Those bytes must not be read. To make that happen, we need to keep track of where exactly the FIFO wrapped around. The **lb** index (lb for loopback) serves this purpose.

On to code.

First, let's define a constant that will tell us the maximum value **wr** can take before the FIFO wraps around. It's the size of the FIFO's byte array minus the maximum size of an incoming packet, in other words :

```
1.  #define RX_BUFFER_MAX_WRITE_INDEX
    (APP_RX_DATA_SIZE -
    CDC_DATA_FS_MAX_PACKET_SIZE)
```

And the code for the function is :

```
 1.  static int8_t CDC_Receive_FS(uint8_t*
     Buf, uint32_t *Len)
 2.  {
 3.    /* USER CODE BEGIN 6 */
 4.    // Update the write index for the
     next incoming packet
 5.    vcp_rx_fifo.wr += *Len;
 6.    // Is the new value too close to the
     end of the FIFO ?
 7.    if (vcp_rx_fifo.wr >=
     RX_BUFFER_MAX_WRITE_INDEX)
 8.    {
 9.      // Solution : wrap-around (and save
     wr as lb)
10.      vcp_rx_fifo.lb = vcp_rx_fifo.wr;
11.      vcp_rx_fifo.wr = 0;
12.    }
13.    // Tell the driver where to write the
     next incoming packet
14.    USBD_CDC_SetRxBuffer(&hUsbDeviceFS,
     vcp_rx_fifo.data + vcp_rx_fifo.wr);
15.    // Receive the next packet
16.
     USBD_CDC_ReceivePacket(&hUsbDeviceFS);
17.    return (USBD_OK);
18.    /* USER CODE END 6 */
19.  }
```

So far so good : with this code, we're now able to receive data from the host and store it in a FIFO automatically. I've made this function very short because I have no idea how often it may be called. Callbacks for middleware should be treated like interrupt handlers : make then short, make them quick.

Until now, we've modified template callback functions that the middleware itself will use. It's time to work on the functions that will be called from our **application** code to send and receive data. The API, if you will.

11. The Send Function

One of the five template functions in **usbd_cdc_if.c** is **CDC_Transmit_FS**. It's the only one that isn't a middleware callback because it's *supposed* to be the one your application calls to tell the

middleware to send data to the host.

I say “supposed” because its default code is too basic to be of any use. Here it is :

```
1.  uint8_t CDC_Transmit_FS(uint8_t* Buf, ui
    Len)
2.  {
3.    uint8_t result = USBD_OK;
4.    /* USER CODE BEGIN 7 */
5.    USBD_CDC_HandleTypeDef *hcdc =
    (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pCl
6.    if (hcdc->TxState != 0){
7.      return USBD_BUSY;
8.    }
9.    USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Bu
10.   result =
    USBD_CDC_TransmitPacket(&hUsbDeviceFS);
11.   /* USER CODE END 7 */
12.   return result;
13. }
```

You pass it a pointer to the data you want to send (**Buf**) and the number of bytes you want to send (**Len**). If the middleware’s underlying USB device buffer is ready (meaning, if it’s done sending the data from a previous call) then it passes your arguments to that driver and tells it to transmit.

This is a problematic function : **USBD_CDC_TransmitPacket** is non-blocking, and you don’t get a “transmission complete” callback or interrupt you can use at your application’s level. That means you need to call it repeatedly until the middleware is ready to transmit. This is polling, and as we all know, polling is evil. It’ll suck your CPU cycles harder than a vampire with a family to feed.

Even worse, until the transmission completes you can’t write new data to your buffer because you would overwrite previous data before it is sent to the host. Let’s be blunt : this function is crap. It’s only good for one thing : it tells us how to ask the middleware to send data to the host. We’ll need that knowledge later.

For now, we’re going to write a completely different function (still in **usbd_cdc_if.c**) called **vcp_send**.

vcp_send takes the same arguments, but instead of passing them to the middleware, it stores **Len** bytes from **Buf** into our Tx circular FIFO. The actual transmission will take place elsewhere (I hope you like the **suspense**).

The function is relatively simple : first, it verifies that there's enough room in our transmit FIFO to store **Len** bytes. Then it verifies if there's enough room in the FIFO's **tail**. If so, **Buf** gets copied to the tail. Otherwise, the function fills the tail with the start of Buf and copies the end of Buf to the **head** of the FIFO (wrap-around). Finally, it updates the FIFO's write index.

Here's the code :

```
1.  int vcp_send (uint8_t* buf, uint16_t
    len)
2.  {
3.    // Step 1 : calculate the occupied
    space in the Tx FIFO
4.    int cap = vcp_tx_fifo.wr -
    vcp_tx_fifo.rd;   // occupied capacity
5.    if (cap < 0)    // FIFO contents wrap
    around
6.      cap += APP_TX_DATA_SIZE;
7.    cap = APP_TX_DATA_SIZE - cap;      //
    available capacity
8.
9.    // Step 2 : compare with argument
10.   if (cap < len)
11.     return -1;   // Not enough room to
    copy "buf" into the FIFO => error
12.
13.   // Step 3 : does buf fit in the tail
```

```
    ?
14.    int tail = APP_TX_DATA_SIZE -
    vcp_tx_fifo.wr;
15.    if (tail >= len)
16.    {
17.      // Copy buf into the tail of the
    FIFO
18.      memcpy
    (&vcp_tx_fifo.data[vcp_tx_fifo.wr], buf,
    len);
19.      // Update "wr" index
20.      vcp_tx_fifo.wr += len;
21.      // In case "len" == "tail", next
    write goes to the head
22.      if (vcp_tx_fifo.wr ==
    APP_TX_DATA_SIZE)
23.        vcp_tx_fifo.wr = 0;
24.    }
25.    else
26.    {
27.      // Copy the head of "buf" to the
    tail of the FIFO
28.      memcpy
    (&vcp_tx_fifo.data[vcp_tx_fifo.wr], buf,
    tail);
29.      // Copy the tail of "buf" to the
    head of the FIFO :
30.      memcpy (vcp_tx_fifo.data,
    &buf[tail], len - tail);
31.      // Update the "wr" index
32.      vcp_tx_fifo.wr = len - tail;
33.    }
34.    return 0;  // successful completion
35.  }
```

Between my prose and the comments, I trust everything should be clear.

12. The Receive Function

Earlier, we customized the template **CDC_Receive_FS** function so that it automatically stores data coming from the host into our circular reception FIFO. Now we need a function to let application code read an arbitrary number of bytes out of that FIFO. This new function is complementary to vcp_send, so I'm going to call it **vcp_recv** and give it the same arguments. That way, it'll be very easy to code a simple echo

feature : call vcp_recv to get data from the host, then vcp_send to echo it back to the host.

As we're only reading from a FIFO, the code will be as simple as that of vcp_send. The only twist is that point where the FIFO wraps around is not necessarily the end of the FIFO because of the need to account to USB packets of different lengths. Remember we used the **lb** index in our FIFO to save the wrap-around location :

```
1.  int vcp_recv (uint8_t* buf, uint16_t
    len)
2.  {
3.    // Compute how much data is in the
    FIFO
4.    int cap = vcp_rx_fifo.wr -
    vcp_rx_fifo.rd;
5.    if (cap == 0)
6.      return 0;       // Empty FIFO, no
    data to read
7.    if (cap < 0)  // FIFO contents wrap
    around
8.      cap += vcp_rx_fifo.lb;  // Notice
    the use of lb
9.    // Limit the FIFO read to the
    available data
10.   if (len > cap)
11.     len = cap;
12.   // Save len : it'll be the return
    value
13.   int retval = len;
14.   // Read the data
15.   while (len)
16.   {
17.     len--;
18.     *buf =
    vcp_rx_fifo.data[vcp_rx_fifo.rd];
19.     buf++;
20.     vcp_rx_fifo.rd++;    // Update read
    index
21.     if (vcp_rx_fifo.rd ==
    vcp_rx_fifo.lb)  // Check for wrap-
    around
22.       vcp_rx_fifo.rd = 0;       //
    Follow wrap-around
23.   }
24.   return retval;
```

```
25.   }
```

The function returns the *actual* number of bytes that were read, which may be zero (if the FIFO is empty) or less than **len** if the FIFO didn't contain enough data.

This function can probably be optimized with the use of memcpy but experience and intuition both tell me that the gain would be negligible, if it even exists, whereas the code would be harder to read.

13. The Service Function

The **CDC_Transmit_FS** function works by calling the underlying CDC driver function **USBD_CDC_TransmitPacket**. There are two important caveats to that function :

- It is non-blocking
- It has no support for circular buffers

To use VCP in a real application requires being able to call **USBD_CDC_TransmitPacket** only when the USB peripheral is ready to transmit a packet. Of course you could poll the status of the USB peripheral... but everyone who's spent more than half a day programming anything knows that polling will just stall your application.

Moreover, using a circular FIFO means that sometimes the data inside it will wrap-around at the end of the FIFO. That's why they call it circular. When this happens, you'll need to send the data using two separate calls to **USBD_CDC_TransmitPacket** : one for the FIFO's tail, then one for the FIFO's head.

Where do you place the code to handle all that logic ? And how do you ensure it doesn't eat all your CPU cycles ?

You place that code into a **service function** and you set it up to get **called automatically at regular intervals**. A service function is essentially a function that you never call directly, but that makes sure your system keeps doing what you told it to do.

If this case, a VCP service function will check if the transmit FIFO contains any data from previous calls to **vcp_send**. If so, it'll check if the USB peripheral is ready to send data to the host. If so, it'll call **USBD_CDC_TransmitPacket** on all or part of the FIFO's contents and then update the FIFO.

Here's what my basic VCP service function looks like :

```
1.  void vcp_service ()
2.  {
3.  USBD_CDC_HandleTypeDef *hcdc =
    (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pCl
4.
5.    // Test if the USB CDC is ready to tra
6.    if (hcdc->TxState == 0)
7.    {
8.      // Update the FIFO to reflect the co
    of the last transmission
9.      vcp_tx_fifo.rd = vcp_tx_fifo.lb;
10.
11.     // Compute how much data is in the F
12.     int cap = vcp_tx_fifo.wr - vcp_tx_fi
13.     if (cap != 0)  // The FIFO is empty
    immediately
14.     {
15.       if (cap < 0)  // The FIFO contents
    around
16.       {
17.         // Send only the tail of the FIF
18.         USBD_CDC_SetTxBuffer(&hUsbDevice
    &vcp_tx_fifo.data[vcp_tx_fifo.rd],
    APP_TX_DATA_SIZE - vcp_tx_fifo.rd);
19.         USBD_CDC_TransmitPacket(&hUsbDev
20.         vcp_tx_fifo.lb = 0;    // Lock t
    tail's data
21.       }
22.       else  // No wrap-around : send the
    FIFO
23.       {
24.         USBD_CDC_SetTxBuffer(&hUsbDevice
    &vcp_tx_fifo.data[vcp_tx_fifo.rd], cap);
25.         USBD_CDC_TransmitPacket(&hUsbDev
26.         vcp_tx_fifo.lb = vcp_tx_fifo.wr;
    the data
27.       }
28.     }
29.   }
30.  }
```

I know it's a bit hairy. There are two key concepts that make this

complicated :

First, USB transmission is non-blocking and under host control. This means that when I send data through the CDC middleware I have no guarantee that it will be transmitted before the STM32 application puts more data into the FIFO. So it's necessary to **write-protect** the area of the FIFO that is passed to the middleware until the service function runs again and has checked that the middleware has indeed sent the data. To do that, I use the FIFO's **lb** index. In the *receive* FIFO **lb** could be thought of as "loop-back", here you can think of it as "lock buffer".

The locking mechanism works like this : on call "n" of **vcp_service**, data from the FIFO is passed to the middleware's USB device driver for transmission, starting at the FIFO's read index **rd**. Instead of updating **rd** immediately, which would effectively release the area of buffer I've just sent, I store the new version of rd in **lb**. Thus, after **vcp_service** returns, **rd** is unchanged and **vcp_send** will therefore be unable to write past it (and overwrite the data being sent). Meanwhile, the USB device driver does its job and eventually, on call "n+x" of **vcp_service**, it signals that it is ready. This means the data has been sent. At that point, I can finally update the **rd** index like so :

```
1.  vcp_tx_fifo.rd = vcp_tx_fifo.lb;
```

This has the effect of unlocking the area of the FIFO between the indices **rd** and **lb**.

Second, the middleware doesn't support circular FIFO's. And because transfers are non-blocking, I can only send one contiguous area of the FIFO every time the service function runs. So when the data in the FIFO wraps around, all I can do is send the **tail** end of the FIFO. The rest of the data to be sent (which obviously starts at the first byte of the FIFO) will be sent the next time the service function runs and the CDC middleware is ready to send.

Note that this service function deals only with sending data to the host : that's because **CDC_Receive_FS** *takes care of receiving data from the host.*

14. In Our Next Episode…

So that was a lot of evil-looking code… but how do you deploy it in your own STM32 application ?

That's what you'll find out in the next page. WordPress doesn't like it

when I try to post an entire novel as a single web page. You can continue reading by clicking **here**.

CDC Library Deployment

In our previous episode, we spent an unhealthy amount of time in front of a computer figuring-out some arcane USB code. Today, we finish what we started and finally get our STM32 to talk with a USB host. It's time to integrate !

Just one thing before we start : I've seized the opportunity to turn this page into a tutorial on how to clone and integrate my STM32 libraries into your projects. As a result, it's rather detailed and might cover a lot of things you already know. You may have reached this page looking for information on how to install a different library, so I've kept things mostly about Git and STM32CubeIDE.

1. Nefastor's Easy-VCP Library

You've seen in the previous page how to configure an STM32CubeIDE project to setup the USB hardware and middleware. Obviously, you have to do that in order to use USB for serial communications. As a quick reminder, this involves opening the microcontroller's configuration GUI (your project's .ioc file) and making the following changes :

- Enable the USB device peripheral you want to use.
- Under "Middleware", enable the Communication Device Class middleware.
- Make sure the correct pins are correctly setup as USB D+ and D-.
- Make sure the USB device peripheral is correctly clocked.

As for the modifications to the middleware's template source files, I've turned them into a **library** that you're now going to integrate into your project. I'm not talking about coding, I'm talking about proper software engineering.

The bottom line is that you won't have to copy and paste code from my website into your project. Instead, you'll just clone my Git repository. Faster, safer, better.

Ideally, your STM32 project is itself a Git repository. Git is good. Everyone should use it. You now have two options : the amateurish or the pro.

An amateur would just copy my library's source files into their own project and call it a day. It's amateurish because in doing so, you lose all history for my library and won't be able to pull future updates I might push to GitHub. You also won't be able to easily switch between different versions. And if you make changes of your own that you think I'd like, you won't be able to make a pull request for me to criticize. If that's how you want to do it, good luck, I don't have time to help you.

A pro would clone my library into their project as a Git **submodule**. That means your project and my library remain two separate repositories, each with their own development history and connection to their respective developers. Your own commits will simply record which version of my library they go with. Thus, you keep the option to easily pull new versions into your project. It's more comfortable and it can't break your code irreversibly.

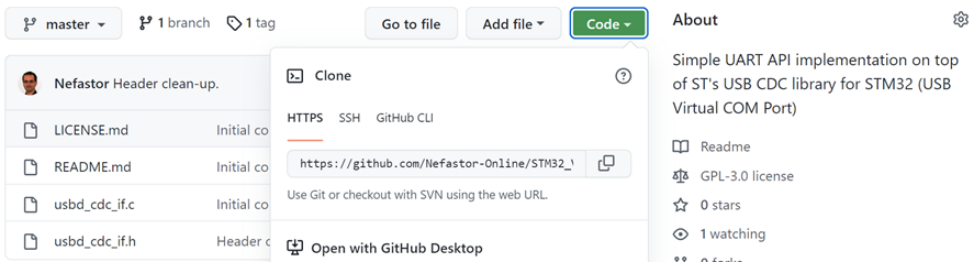
I'm going to assume you'd rather work like a pro and tell you how to do that.

2. Clone the Repository

You will find my repository at :

https://github.com/Nefastor-Online/STM32_VCP

There are different ways to clone a repo; and you may already have your preferred Git tool. The simplest method is to use the command line. Enter the "git clone" command along with the repository URL you get from GitHub :



Open a terminal, navigate to the root of your STM32 project and enter the command :

```
1.  git clone https://github.com/Nefastor-
    Online/STM32_VCP.git
```

Lo and behold, my source code is now in your project as a folder named **STM32_VCP**. Two details :

First, you can rename that folder. The exact name of the folder doesn't matter to Git or to the code. I like to call it VCP, short and sweet. You might have different taste.

Second, you must make sure you're checking out the correct version of the library. At the time of writing, this library has only one branch and one version. However, I might create more in the future :

- My library is built on top of STM32Cube libraries provided by ST. There is a risk that future versions of those libraries will break mine, requiring me to update my code.

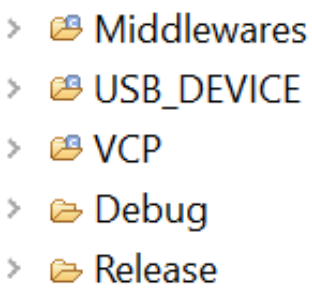
- I may also need to create different branches to support different STM32 families as the needs arise.

Therefore, after cloning, you may want to use the "git branch" and "git tag" commands to list all existing versions and checkout the one you need. The repo's README file also contains a list of hardware that has been tested and with what version.

3. Setup the Project

Cloning created a subfolder in your STM32CubeIDE project. However, at this point, it's just a normal folder like your "Release" or "Debug" output folders. Its contents won't be compiled when you build your project.

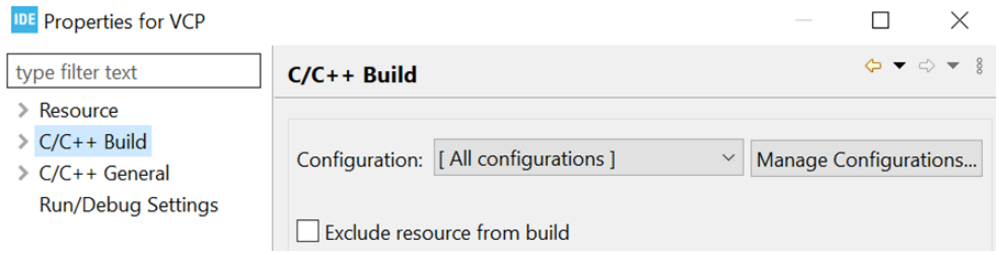
You can see that from the difference in icon :



Your freshly-cloned library needs to have a little "C" on its folder icon. Let's give it one.

Note : if you're not seeing the folder, hit F5 to refresh the view.

In the Project Explorer pane, right-click on the library's folder and click on "Properties". Navigate the dialog box to "C/C++ Build". Unless you have a reason not to, make sure to select "[All configurations]". Finally, untick "Exclude resource from build" :



This will ensure that the IDE compiles every source file it finds in that folder. Click "Apply and Close".

We also have a header file in there. So let's also add the folder to the project's **include paths** : right-click the folder again and select "Add/remove include path..." : the rest is self-explanatory.

4. Declare the Submodule

I've mentioned Git **submodules** earlier. They are simply **nested repositories**, or "repos within repos". This feature of Git was designed with libraries in mind. It solves a fundamental problem : since a library is meant to be used in multiple projects, it has to have a development history **separate** from all those projects.

I know a lot of people who don't find the concept intuitive so I've come up with a relatable allegory.

Suppose you get hired by a company as an employee. That company has a history and you have your own, in the form of your resume. But your past doesn't become that of your new employer upon joining. Looking at the company's history; the only relevant historical information about you is the date your employment started and the skills that got you the job.

Now replace "company" with "project", "employee" with "library", "skills" with "features" and you're much closer to intuiting the concept of a Git submodule.

In practice, Git will ignore any change to your project *if it's in a submodule*. It will only tell you if the submodule contains uncommitted changes or if you've changed the version of the submodule you're using. Should you modify a submodule, you will need to commit it first (to record the modifications), then commit your project (to record that

it's using the modified submodule).

To get this behavior from Git, you just need to tell your project's repository that the nested repository is a submodule. Navigate to your project folder (same path where you cloned the repo) and tell your repo to add a submodule. If you haven't renamed it yet, the command line will be :

```
1.  git submodule add ./STM32_VCP
```

The command takes the path to the submodule, either absolute or (as in this example) relative to where you're executing the command, hence the "./".

The "git status" command will return, among other things, something like this :

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitmodules
        new file:   VCP
```

The addition of a submodule is a change to your project, and that change must be committed.

If you're still unclear about submodules, then I suggest reading the official documentation at https://git-scm.com/book/en/v2/Git-Tools-Submodules

If you came here looking for information on how to integrate one of my libraries that isn't the USB VCP library for STM32, congratulations : you're done, here. Everything that follows is specific to the aforementioned library. Happy trails ! Otherwise, you still have more work to do : read on !

5. Delete Template Files

If you've been paying attention, you've certainly noticed that cloning my library has resulted in a case of split personality. Indeed, you now have two different **usbd_cdc_if.c** and **usbd_cdc_if.h** files in your project :

- Those that STM32CubeIDE created when you generated the code for your project. Remember, they live under "USB_DEVICE/App" in your project.

- Those that you've just cloned.

Needless to say, you can only keep one set. Needless to say, you need to keep the set you just cloned.

You must therefore **delete** the two files located under "USB_DEVICE/App".

Keep in mind that you'll need to delete those two files every time you regenerate the code for your project. Don't worry if you forget : your project will fail to compile, and then you'll remember why. I haven't found an elegant way to tell ST's code generator *not* to generate them. If you know how to do that, I'd appreciate it if you told me. My contact form isn't just meant for asking questions, it's also for giving tips (of both the technical and monetary sort) and compliments.

6. Integration With Your Code

At this point, the library is ready to compile as part of your project. All that's left to do is use it, and that's the easy part. You can find the whole VCP API in **usbd_cdc_if.h**, it's just four functions :

```
1.  /* USER CODE BEGIN EXPORTED_FUNCTIONS
    */
2.  void vcp_init ();
3.  int vcp_send (uint8_t* buf, uint16_t
    len);
4.  int vcp_recv (uint8_t* buf, uint16_t
    len);
5.  void vcp_service ();
6.  /* USER CODE END EXPORTED_FUNCTIONS */
```

Initialization, transmission, reception, and a service routine. If it were any more concise it would be a haiku.

First, make sure that you're including the **usbd_cdc_if.h** header in the source files where you want to use USB serial communications. Typically, that would be "main.c".

Because you've enabled the USB device and middleware, STM32CubeIDE generated some of the relevant code, but not all of it. Here's an excerpt of a typical "main.c" file generated with USB support :

```
1.  /* Includes ------------------------
    ----------------------------------
    */
2.  #include "main.h"
3.  #include "usb_device.h"
```

```
4.
5.  /* Private includes ----------
    ----------------------------------
    */
6.  /* USER CODE BEGIN Includes */
7.  #include "usbd_cdc_if.h"
8.  /* USER CODE END Includes */
```

The IDE generated the first two include directives. The last one you may have to type in yourself. As usual, your own code must always go between **USER CODE** comments.

Next, you must call the library's initialization function, **vcp_init**. Obviously, it must be called *before* you can transfer data.

Finally, you must call the library's service function, **vcp_service**, at regular intervals. The easiest way to do this is from the System Tick (**SysTick**) interrupt service routine **SysTick_Handler**, which runs automatically every millisecond. You'll find it in your project's "Core/Src/stm32yyxx_it.c" file, along with all your ISR. It should end-up looking like this :

```
1.  void SysTick_Handler(void)
2.  {
3.    /* USER CODE BEGIN SysTick_IRQn 0 */
4.
5.    /* USER CODE END SysTick_IRQn 0 */
6.    HAL_IncTick();
7.    /* USER CODE BEGIN SysTick_IRQn 1 */
8.    // USB VCP library : background
    processing of data transmission
9.    vcp_service ();
10.   /* USER CODE END SysTick_IRQn 1 */
11. }
```

You will also need to **#include** "usbd_cdc_if.h" for the compiler to find the function.

Now that's done, congratulations : at long last, you're ready to write code that can talk to your PC through USB. We'll look at how it's done using a very simple example : a **serial echo** application.

7. Demonstration : Serial Echo

We've gone through a lot of setup and code, now it's time to check if it all works or if I just wasted your time. Spoiler alert : I haven't.

The simplest test of an RS232 port (or UART, or COM port) is the **echo**. Take every byte sent to the STM32 and send it back where it came from. This single test verifies that both **reception** and **transmission** work.

In your main function's infinite loop you just need to add :

```
    uint8_t buf[1000];
    vcp_init ();
    while (1)
    {
        /* USER CODE END WHILE */

        /* USER CODE BEGIN 3 */
 // VCP demonstration - Echo all data received over VCP back to the host
      int len = vcp_recv (buf, 1000);  // Read up to 1000 bytes
      if (len > 0)    // If some data was read, send it back :
        len = vcp_send (buf, len);
    }
    /* USER CODE END 3 */
```

Build and Flash your project to your STM32, then connect its USB port to your PC.

Warning : before plugging the USB cable and depending on how your board is designed, you may need to make adjustments to the hardware. This is particularly true for the USB OTG port on all Nucleo boards !

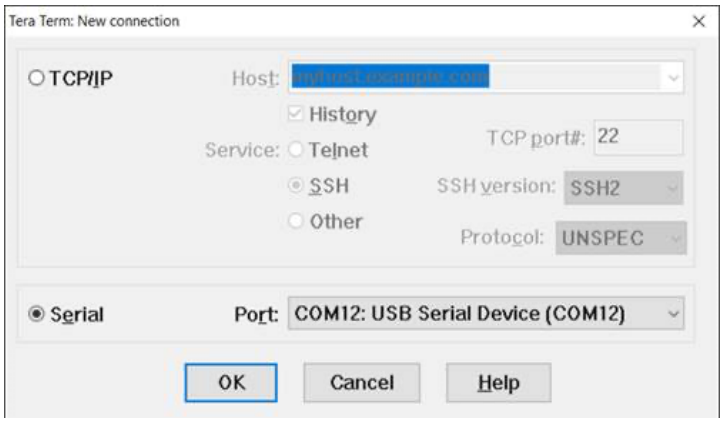
Much earlier, we have checked that our PC detected the STM32 as a COM port, but it doesn't hurt to check again. Besides, we're going to need its COM port number (or tty device name, if you're using Linux) in order to connect to it.

Next, we need software on our PC to talk to a serial device. We'll use a simple **terminal emulation program**. I recommend **TeraTerm** : it's simple, no-frills, and unlike PuTTY it actually respects the baud rate you set. Also, unlike PuTTY, it doesn't lose its marbles when its serial connection is lost. That's very important if you're going to play with VCP : any time you recompile and Flash, PuTTY will force you to click a dialog box, close it and restart it. **Tera Term** will just sit there and wait for your STM32 to return to active duty.

You can download **Tera Term** for free from many different places, I'll

let you Google it. At the time of writing, I'm using version 4.91.

Once you've installed it, and made sure your STM32 is connected to a USB port, launch it. You should be greeted with this dialog :



Make sure you pick the right COM port and click **OK**. You should now be staring at a black window with a blinking cursor. Unless you're stuck in the Matrix, don't expect to suddenly get this :

If you *do*, don't worry. Just give me five minutes to get into my S&M leather costume and find my sunglasses and I'll be right with you with a bunch of colorful pills so we can discuss USB, plugging, unplugging and the nature of reality.

But seriously... back to work :

By default, terminal programs do *not* echo your keystrokes : that feature, known as "local echo", interferes with any real-world application of serial communications. What you type goes out the COM port, and only what comes *into* the COM port will be displayed. Our STM32 running the echo code will, in effect, take your keystrokes and send them back so the terminal program can display them.

Don't be shy, type a few things on the keyboard. If they end-up on the display, congratulations, you've got a working virtual COM port STM32 project. And the knowledge to add this feature to any or your future

projects.

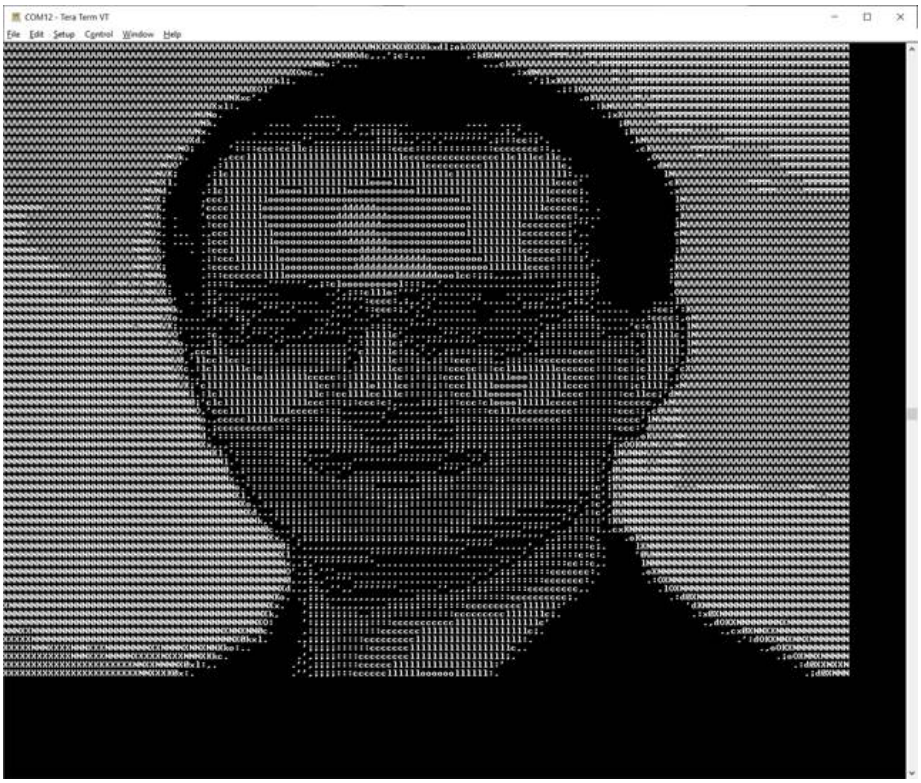
If you *don't* get an echo, well, this has been a very long and tedious process. Maybe you missed a step. Maybe you fell asleep at the keyboard just like Neo. Drink some water, remind your significant other(s) that you're still alive, and then take it from the top.

8. Fun With Testing

So. You've got some code that lets your STM32 echo **keystrokes**. But unless you've been bitten by a **radioactive secretary** we both know you're not going to stress-test your STM32's VCP just by typing. You may want to test sending a lot of data in one go and then verify that it all went through and back without errors.

The simplest and quickest way to do that is to send some ASCII art down the pipe. Because it's an image (of sorts) you will be able to easily spot missing bytes from image distortion even as you send thousands of bytes in a row.

You can use this website : https://www.ascii-art-generator.org/ to turn any picture into ASCII art that you can then copy and paste into Tera Term. This will send the whole thing to your STM32 and display what the microcontroller received and sent back. I'm sexy so I usually use my own portrait for this :



If this works, it means both your circular FIFO's work as they should. If it doesn't, maybe your baud rate is too high, or maybe your Tera Term

has a problem handling carriage returns (depends on your **operating system**) or maybe one or both of your FIFO's doesn't work quite right and you need to look at your code again. Too many possibilities for me to go through, but at this point you have all the knowledge you need to debug your way out.

9. A Quick Summary

Just to recap everything so far and get a high-orbit view of what's going on :

First, you've configured our STM32 in **STM32CubeIDE** to enable its **USB Full Speed device** peripheral, and enabled the Communication Device Class (CDC) **middleware**. You've generated the code for your project, thereby adding all sorts of source files (some of them templates) to your project.

Because the CDC middleware is very generic, you've added a UART-like API by cloning this website's VCP library into your project. This library added support of circular FIFO's and functions to synchronize data transfer to (**vcp_service**) and from (**CDC_Receive_FS**) a USB host, using those FIFO's. Those functions are essentially interrupt handlers and work in the background of your application, to keep CPU usage as low as possible.

The library also gave you functions to send (**vcp_send**) and receive (**vcp_recv**) data to and from the host at the **application level**.

Having built and flashed your project to your STM32 board, you have verified that you can actually connect to your STM32 through a **virtual COM port**.

This was quite a lot of work, so I think I'll stop here for now... but there is still a lot to explore on the topic of ST's CDC middleware. Expect another episode in the future, perhaps to cover different scenarios like FreeRTOS deployment and porting to specific STM32 families and boards.