**Lesson 15**

**HAL. USART. DMA**

Today we will continue the lessons of connecting the STM32 microcontroller to a PC via the USART interface.

Only in contrast to past lessons, we apply for this technology DMA.

Since we use this technology for the first time, I will briefly introduce you to it.

**DMA** ( **direct memory access** ) - direct access to memory. That is, we copy data from one area of ​​memory to another area of ​​memory, either from the periphery to the memory area or from the memory area to the periphery, but we copy not one byte, using ALU registers, but directly without using ALU. This is the way to achieve greater speed, unloading CPU time. ALU only gets information about how many specific information units need to be forwarded, as well as the address of the source and receiver memory, everything else is already happening without his participation. We either know approximately how much time is required for this, and based on this we are already building our algorithm, or we use interrupts and process the event of the end of the transfer via DMA there.

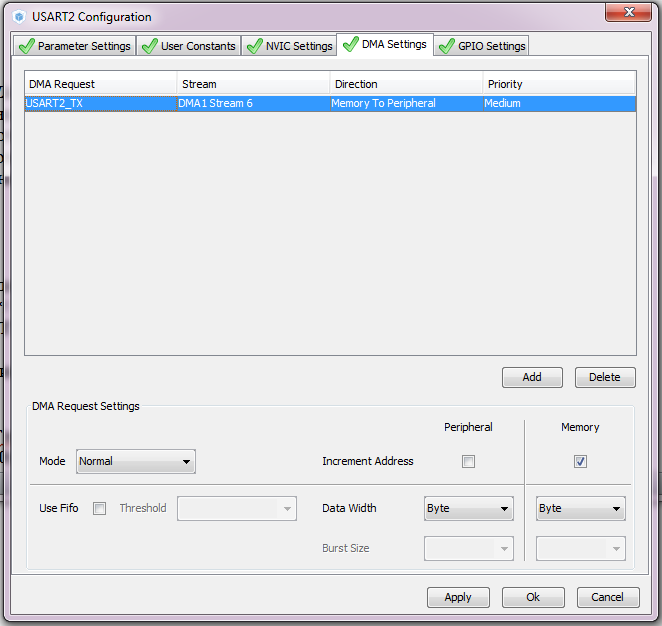
DMA technology is used, of course, not only in the USART interface, but in many other interfaces and cases, and the settings will be similar, therefore, after reading this article, and also having seen the video version of the lesson, you can already with much less difficulty understand the programming of DMA in others case.

But today we have USART.

The project was created from USART\_TRANSMIT, named USART\_TRANSMIT\_DMA, the scheme also did not change.

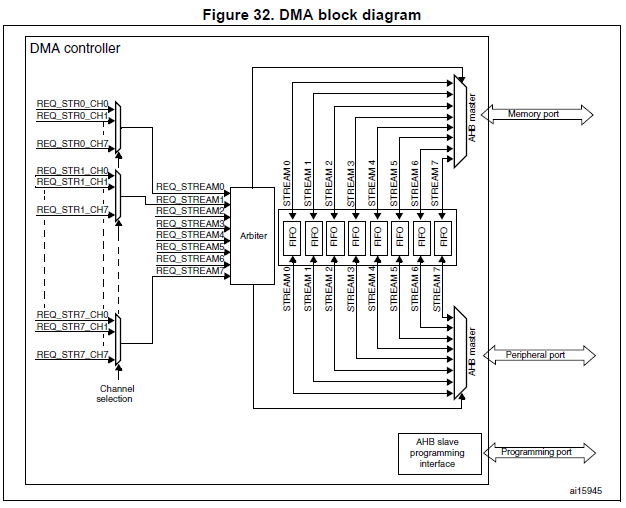
Start the cube, disable CTS and RTS in USART2, if you have them turned on.

In the settings in Configuration, we first include general interrupts. Then go to the DMA. In the DMA Settings tab, add DMA and configure it as follows

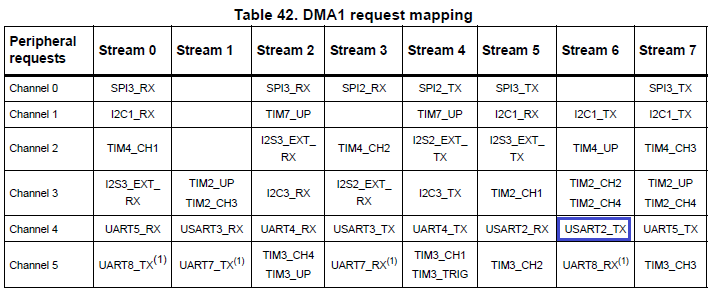


We choose USART2\_TX, as we will be engaged in the transfer and DMA technology, we will apply it in data transmission. In this case, all the streams and channels, as well as the direction of the DMA, the Cube MX will choose for us.

In DMA and in the first and second there are how many channels and several threads are designed to distribute all the buses and interfaces. With the channels and so everything is clear. With flows a little bit different. To prevent various collisions in the FIFO buffer in the DMA, as many as 8 buffers were created and each thread was assigned its own flow. Let's see how to organize DMA in STM32 4 series in Reference Manual



Also, in the same technical documentation, it is well-defined which channel and thread for which interface it works



This table continues on this page, and it is followed by a similar table for the second DMA, but this is not so interesting for us, since our peripherals on this page and the stream with the data channel on our periphery are highlighted in a blue square. That is Cube MX was not mistaken, having exposed to us 6 flow.

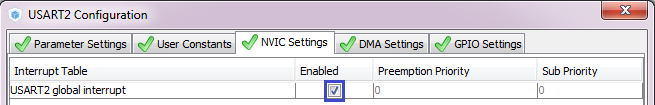
At the bottom of the DMA settings, we do not touch anything.

Leave the mode **Normal** . There is also a **Circular** mode , which forces DMA to transmit data cyclically without stopping.

We also leave the incrementing of the memory address only. We do not add peripheral addresses in each cycle, we work with only one periphery and have only one memory address.

We do not use the **FIFO** buffer , so we do not include it in our case. It is needed in case that in one transmission cycle we can transmit and receive streams in one cycle, also using a special buffer. This further speeds the data transfer rate. The data transfer unit leaves a **byte** , so it will be more convenient for us, although we can easily transmit in half-words, and even words (4 bytes each).

Also include in the NVIC tab just in case interrupts



We generate the project.

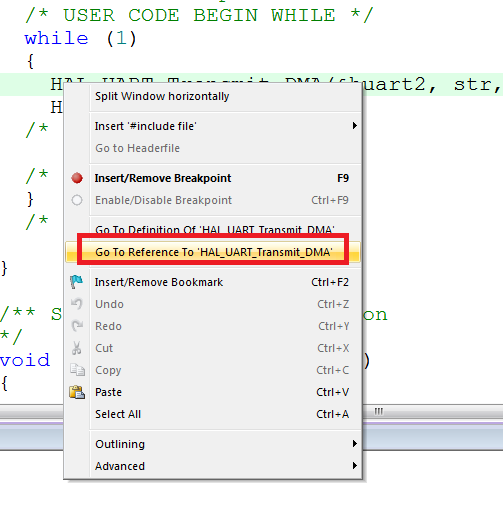
In the main function main () for beauty, we fix the variable

  / \* USER CODE BEGIN 1 \* /

        uint8\_t str [] = "USART Transmit **DMA**\ r \ n";

  / \* USER CODE END 1 \* /

We find the function for the transfer using DMA as follows:



And in an infinite loop, we also change

  while (1)

  {

        HAL\_UART\_Transmit **\_DMA**(& huart2, str, **sizeof (str) -1**);

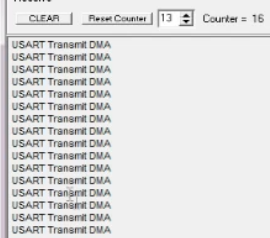
        HAL\_Delay (100);

  / \* USER CODE END WHILE \* /

How, I think, all noticed, another function is already applied, intended specifically for transmission using DMA.

The timeout is no longer needed. In this, interruptions help us at the end of the transfer.

We collect, open the terminal program, click **Connect** there, flash , watch



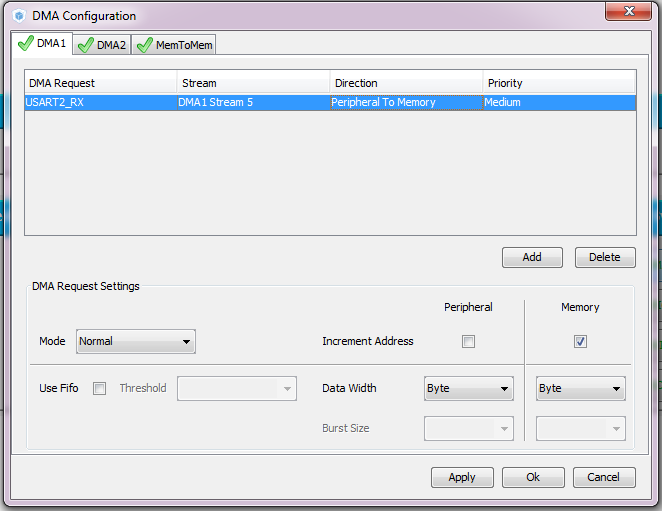
Everything is transmitted from us.

I did not disable the display, as it will be required for us to display the data already received.

To receive data, another project was created from the USART\_RECEIVE project.

The name of the new project is USART\_RECEIVE\_DMA.

We start the newly created project in CUBE, similarly go into Configuration, make sure that we have interrupts enabled in USART2, then go to DMA and configure everything like this:



We have already got stream 5 here, if we look at the table that was given above, then it is, As we receive this kind of flow, it is used. Also, we, respectively, changed the direction of transmission - from the periphery to the memory.

Apply the settings, generate the project. We open it.

Add the file lcd.c

Let's change the size of the variable to main ()

  / \* USER CODE BEGIN 1 \* /

        char str [ **21**] = {0};

  / \* USER CODE END 1 \* /

At us the display on 20 characters in a line, here and we will fill with characters the whole line.

Change the function

        LCD\_String (str);

        HAL\_UART\_Receive **\_DMA**(& huart2, (uint8\_t \*) str, 20);

  / \* USER CODE END 2 \* /

In an infinite loop, we also change the function and size of the received data, and also set the zero for the index 20

                LCD\_SetPos (0, 3);

                str [ **20**] = 0;

                LCD\_String (str);

                HAL\_UART\_Receive **\_DMA**(& huart2, (uint8\_t \*) str, **20**);

Let's try to assemble the display and check the operation of our code



As we can see, we can transfer from the PC not necessarily completely 20 characters at once, that is, the transmission stream already, despite the fact that we have exactly 20 bytes in the function, directly transfers bytes to the memory of the display in any number, simply when in aggregate it will be transmitted exactly 20, then the USART bus will go into a state where it will again be able to access it from the outside, and the PC will be able to transfer data further. We will transfer the same 10 bytes



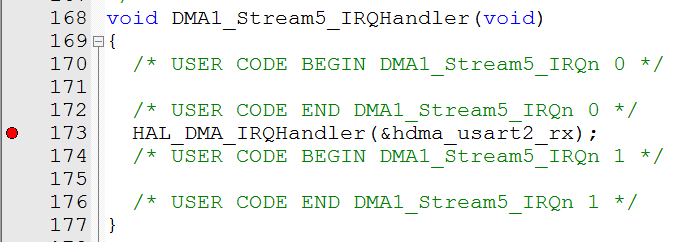
As we can see, the line was filled to the end.

Let's try to transfer some other symbols, everything is transferred.

Only now the if (huart2.RxXferCount == 0) line  loses its meaning, because the parameter to be checked will always be zero when using DMA. What should we then check to display the line when it is completely filled?

The debugger will come to our aid.

We'll collect the project, run the debugger, set the breakpoint here in the file stm32f4xx\_it.c



We start the firmware.

In the terminal, we pass a string like "123456789" - we do not get to the interrupt, and as soon as we pass the 10th character, we get into an interrupt. We move further inside the function HAL\_DMA\_IRQHandler (& hdma\_usart2\_rx)

In the window of variables for tracking we add the line hdma\_usart2\_rx - this is the parameter of our handler function, you can also add hdma, which is basically the same thing. And we wait when this state will change the State parameter. Before the change, remember the current status - it will be

0x02 HAL\_DMA\_STATE\_BUSY. When it changes, it will take the form 0x31 HAL\_DMA\_STATE\_READY\_HALF\_MEM0.

This means that the interrupt caused half the DMA memory to be full. Run the next code.

In the terminal, we first send "123456789", then we enter one character - it will be obtained already for the line and for DMA memory of 20d - we get again into the interrupt handler from DMA.

We walk again before changing the status. Now it will be 0x11 HAL\_DMA\_STATE\_READY\_MEM0.

This is exactly what we need. We will track it. In our condition, in an infinite cycle, we change the

        if ( **hdma\_usart2\_rx.State == HAL\_DMA\_STATE\_READY\_MEM0**)

        {

                HAL\_Delay (100);

Also at the end of processing the condition, we add the reset of this flag to the default view HAL\_DMA\_STATE\_BUSY

                LCD\_String (str);

                HAL\_UART\_Receive\_DMA (& huart2, (uint8\_t \*) str, 20);

**hdma\_usart2\_rx.State = HAL\_DMA\_STATE\_BUSY;**

        }

  / \* USER CODE END WHILE \* /

We collect, sew, look. We check by the terminal

