Direct Memory Access Management

DMA is used as an independent memory transfer between memory and peripherals that frees CPU from handling data exchange, therefore it speeds up the system performance. Understand about transfer modes, configure source and destination address. Example to copy data from memory to memory, or memory to a peripheral.

#arm #stm32 #dma

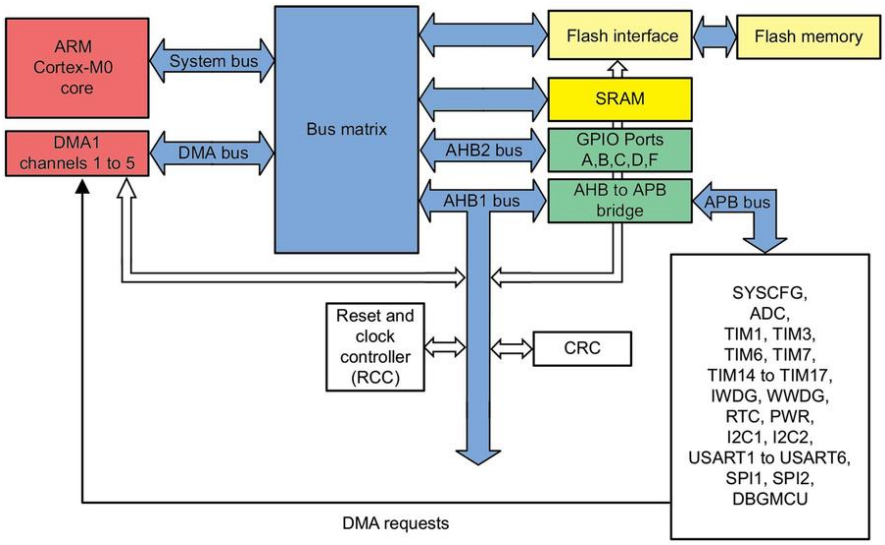
Last update: May 3, 2021

Table of Content

1. DMA Controller
2. DMA Channels
3. DMA Circular Mode
4. DMA Interrupts
5. HAL Software
6. Lab: DMA Memory to Memory
    6.1. Enable DMA
    6.2. Generated code
    6.3. User code
    6.4. Inspect interrupts
7. Lab: DMA Memory to UART TX
    7.1. Enable DMA on UART TX
    7.2. Generated code
    7.3. User code
8. Lab: DMA Peripheral to Memory

1. DMA Controller

The *Direct Memory Access* (DMA) controller is a dedicated and programmable hardware unit that allows MCU peripherals to access to internal memories without the intervention of the Cortex-M core. The CPU is completely freed from the overhead generated by the data transfer (except for the overhead related to the DMA configuration), and it can perform other activities.



Bus architecture of and STM32F0 MCU

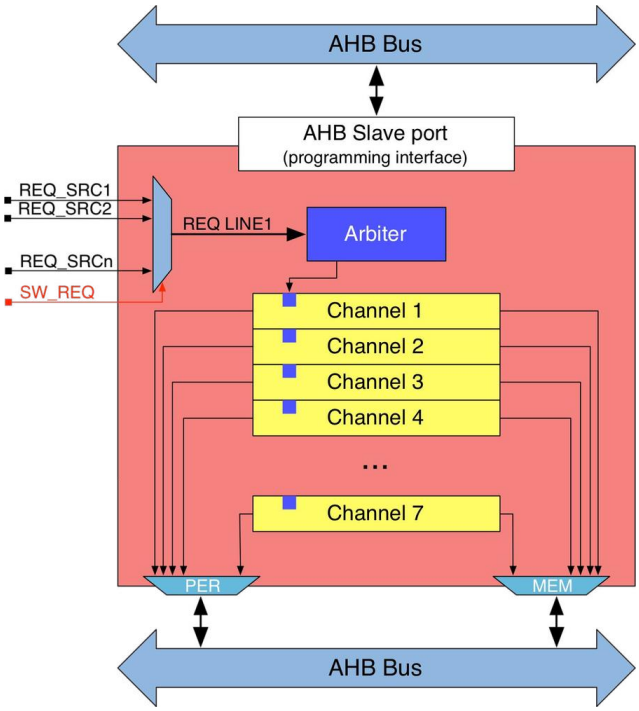
Some important things about DMA:

- Both the Cortex-M core and the DMA controller interact with the other MCU peripherals through a series of buses
- Both the Cortex-M core and the DMA controller are masters, This means they are the only units that can start a transaction on a bus, but they cannot access to the same slave peripheral at the same time

In every STM32 MCU, the DMA controller is a hardware unit that:

- has *two master* ports, named *peripheral* and *memory* port respectively, connected to the Advanced High-performance Bus (AHB), one able to interface a slave peripheral and the other one a memory controller (SRAM, flash, FSMC, etc.); in some DMA controllers a peripheral port is also able to interface a memory controller, allowing *memory-to-memory* transfers
- has one *slave* port, connected to the AHB bus, used to program the DMA controller from the other master, that is the CPU
- has a number of independent and programmable *channels* (request sources), each one connectable to a given peripheral request line (UART_TX, TIM_U, etc.)
- allows to assign different *priorities* to channels, in order to arbitrate the access to the memory giving higher priority to faster and important peripherals

- allows the data to flow in both directions, that is from *memory-to-peripheral* and from *peripheral-to-memory*



DMA Controller

2. DMA Channels

Each channel can handle DMA transfer between a peripheral register located at a fixed address and a memory address. The amount of data to be transferred (up to 65535) is programmable. The register which contains the amount of data items to be transferred is decremented after each transaction.

The transfer data sizes of the peripheral and memory are fully programmable through the *PSIZE* and *MSIZE* bits in the *DMA_CCRx* register.

Peripheral and memory pointers can optionally be automatically post-incremented after each transaction depending on the *PINC* and *MINC* bits in the _DMA_CCRx _register. If incremented mode is enabled, the address of the next transfer will be the address of the previous one incremented by 1, 2, or 4 depending on the chosen data size.

3. DMA Circular Mode

The circular mode is available to handle circular buffers and continuous data flows (e.g. ADC scan mode). This feature can be enabled using the *CIRC* bit in the *DMA_CCRx* register.

When the circular mode is activated, the number of data to be transferred is automatically reloaded with the initial value programmed during the channel configuration phase, and the DMA requests continue to be served.

4. DMA Interrupts

An interrupt can be produced on a Half-transfer, Transfer complete, or Transfer error for each DMA channel. Separate interrupt enable bits are available for flexibility.

5. HAL Software

The Hardware Abstract Layer (HAL) is designed so that it abstracts from the specific peripheral memory mapping. But, it also provides a general and more user-friendly way to configure the peripheral, without forcing the programmers to now how to configure its registers in detail.

excerpt from Description of STM32F0 HAL and low-layer drivers

How to use DMA HAL

1. Enable and configure the peripheral to be connected to the DMA Channel (except for internal SRAM / FLASH memories: no initialization is necessary). Please refer to Reference manual for connection between peripherals and DMA requests.
2. For a given Channel, program the required configuration through the following parameters: Transfer Direction, Source and Destination data formats, Circular or Normal mode, Channel Priority level, Source and Destination Increment mode, using `HAL_DMA_Init()` function.

   ✏️ In Memory-to-Memory transfer mode, Circular mode is not allowed

3. Use `HAL_DMA_GetState()` function to return the DMA state and `HAL_DMA_GetError()` in case of error detection.
4. Use `HAL_DMA_Abort()` function to abort the current transfer
5. Operation modes:

   Polling mode IO operation

   - Use `HAL_DMA_Start()` to start DMA transfer after the configuration of Source address and destination address and the Length of data to be transferred
   - Use `HAL_DMA_PollForTransfer()` to poll for the end of current transfer, in this case a fixed Timeout can be configured by User depending from the application

     ⚠️ If DMA interrupt is enabled, this function may not work properly, read more in Notes - DMA Polling

Interrupt mode IO operation

- Configure the DMA interrupt priority using `HAL_NVIC_SetPriority()`
- Enable the DMA IRQ handler using `HAL_NVIC_EnableIRQ()`
- Use `HAL_DMA_Start_IT()` to start DMA transfer after the configuration of Source address and destination address and the Length of data to be transferred. In this case the DMA interrupt is configured
- Use `HAL_DMA_Channel_IRQHandler()` called under `DMA_IRQHandler()` Interrupt subroutine
- At the end of data transfer `HAL_DMA_IRQHandler()` function is executed and user can add his own function by customization of function pointer `XferCpltCallback` and `XferErrorCallback` (i.e a member of DMA handle structure).

6. Lab: DMA Memory to Memory

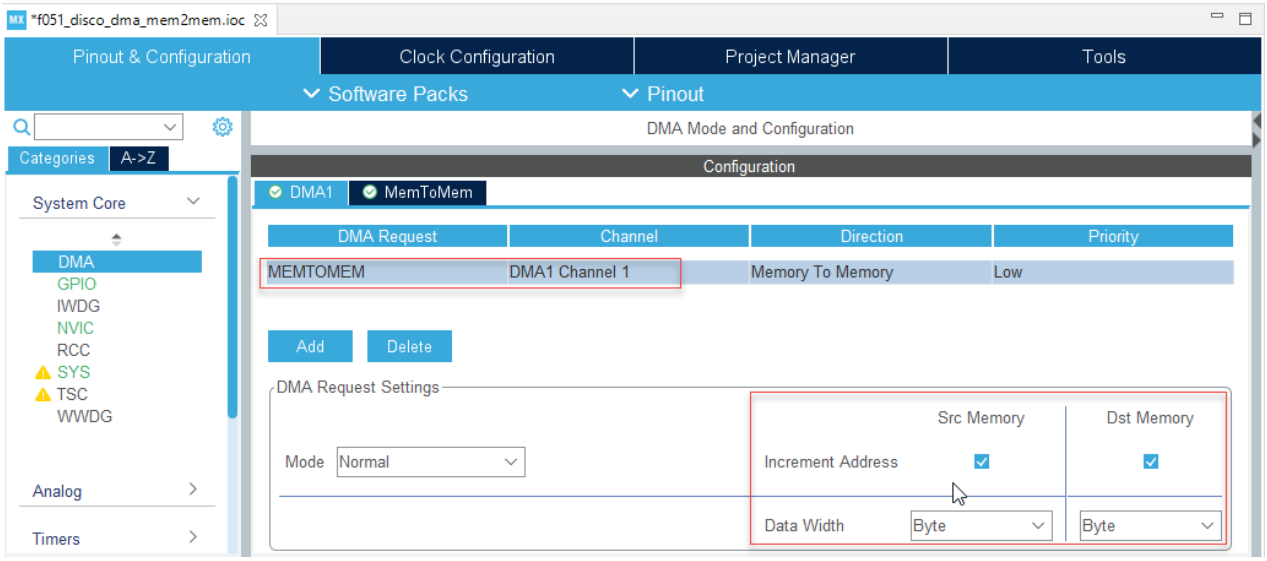
To compare the performance of DMA with CPU, this project will compare the speed of using `memcpy()` function and the DMA Memory-to-Memory transfer.

Application target: - Create a 4KB data block in flash memory (slow peripheral) - Create a 4KB buffer in SRAM memory (fast peripheral) - Run CPU `memcpy()` function to copy from flash to sram - Run DMA Mem2Mem transfer to copy from flash to sram - Each operation's duration will be measure by a pulse on GPIO to visual view on a logic analyser

Start a new project, the follow below guide.

6.1. Enable DMA

1. Goto **System Core** > **DMA** peripheral
2. Add new DMA Request, then select **MEM TO MEM**.
3. Select DMA Channel, e.g. DMA1 Channel 1
4. Check the boxes that increase the Source Address and Destination Address, with Data Width as *Byte* (same as `memcpy()` function in non-optimized mode)



Enable DMA in System Core

6.2. Generated code

An instance of `DMA_HandleTypeDef hdma_memtomem_dma1_channel1` is created to hold the DMA object.

Then the function `MX_DMA_Init()` which takes care of setting up the DMA channel enabled in IDE is generated:

```
static void MX_DMA_Init(void) {
  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* Configure DMA request hdma_memtomem_dma1_channel1 on DMA1_Channel1 */
  hdma_memtomem_dma1_channel1.Instance = DMA1_Channel1;
  hdma_memtomem_dma1_channel1.Init.Direction = DMA_MEMORY_TO_MEMORY;
  hdma_memtomem_dma1_channel1.Init.PeriphInc = DMA_PINC_ENABLE;
  hdma_memtomem_dma1_channel1.Init.MemInc = DMA_MINC_ENABLE;
  hdma_memtomem_dma1_channel1.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
  hdma_memtomem_dma1_channel1.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  hdma_memtomem_dma1_channel1.Init.Mode = DMA_NORMAL;
  hdma_memtomem_dma1_channel1.Init.Priority = DMA_PRIORITY_LOW;
  if (HAL_DMA_Init(&hdma_memtomem_dma1_channel1) != HAL_OK)
  {
    Error_Handler( );
  }

}
```

6.3. User code

Declare memory block

The first step is to declare the data block in the Flash and the buffer in the SRAM.

> Using `const` modifier to put variable in to the Flash Memory by Linker. Read more here.

```
#define TRANSFER_SIZE   4096
const char flash_data[TRANSFER_SIZE] = "hello";
char sram_buffer[TRANSFER_SIZE];
```

Measure execution time

To measure the execution time in a logic analyser, just use any GPIO in to indicate it by raising it to *HIGH* logic level. For example, on the STM32F0 Disco board, LD3 on PC9 is used.

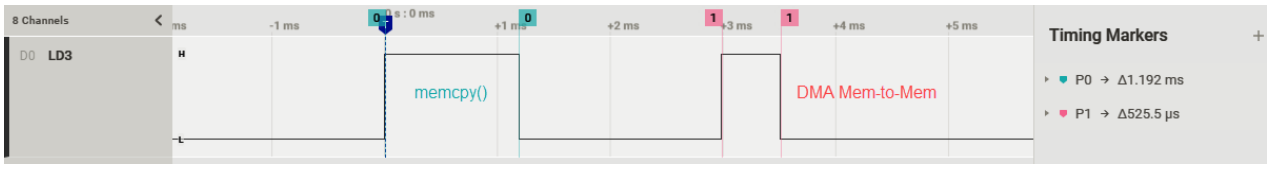
Test the CPU `memcpy()` function:

```
HAL_Delay(1);
HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
memcpy(&sram_buffer, &flash_data, TRANSFER_SIZE);
HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
```

Test the DMA Memory-to-Memory transfer, note to use the function `HAL_DMA_PollForTransfer()` to blocking the CPU execution while DMA is running:

```
HAL_Delay(1);
HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
HAL_DMA_Start(&hdma_memtomem_dma1_channel1, (uint32_t)&flash_data,
(uint32_t)&sram_buffer, TRANSFER_SIZE);
HAL_DMA_PollForTransfer(&hdma_memtomem_dma1_channel1, HAL_DMA_FULL_TRANSFER,
HAL_MAX_DELAY);
HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
```

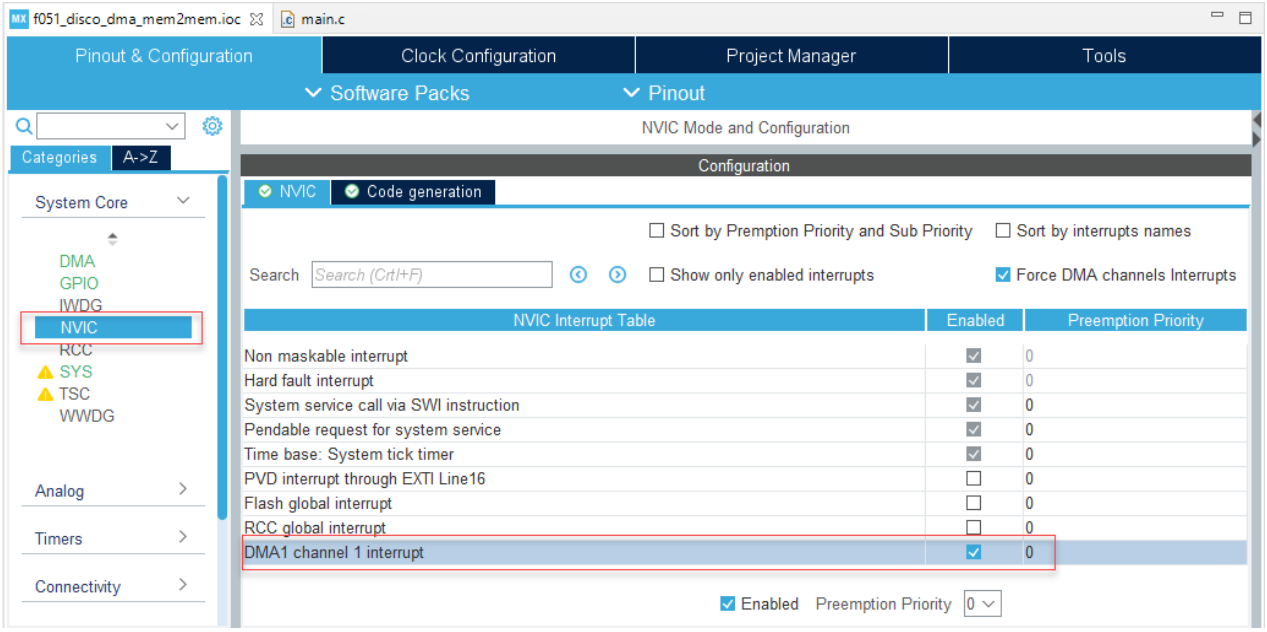
Here are the output pulses showing that `memcpy()` needs 1192 us to complete while DMA only needs 525 us.



Compare between `memcpy()` and DMA Mem-to-Mem

6.4. Inspect interrupts

Now, enable DMA interrupts using IDE by going to **NVIC** sections and check on the row saying that *"DMA1 channel 1 interrupt"*:



Enable DMA Interrupt in IDE

Then generate modified code which is newly added into the *stm32xxxx_it.c* file:

```
void DMA1_Channel1_IRQHandler(void) {
  HAL_DMA_IRQHandler(&hdma_memtomem_dma1_channel1);
}
```

The function `HAL_DMA_IRQHandler()` will call to 2 handling functions registered in DMA Handler instance:

- Half data length callback `hdma->XferHalfCpltCallback()`
- Full data length callback `hdma->XferCpltCallback()`

By default, those callback are not set, therefore, write 2 functions to handle DMA interrupts:

```
void DMA_HalfTransferCallback(DMA_HandleTypeDef * _hdma) {
  HAL_GPIO_WritePin(LD4_GPIO_Port, LD4_Pin, GPIO_PIN_SET);
}
void DMA_FullTransferCallback(DMA_HandleTypeDef * _hdma) {
  HAL_GPIO_WritePin(LD4_GPIO_Port, LD4_Pin, GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
}
```

Then register them, and call the DMA Start in Interrupt mode:

```
int main(void)
{

``` cpp
  HAL_Delay(1);
  HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
#ifdef USE_DMA_INTERRUPT
  HAL_DMA_RegisterCallback(&hdma_memtomem_dma1_channel1,
HAL_DMA_XFER_HALFCPLT_CB_ID, DMA_HalfTransferCallback);
  HAL_DMA_RegisterCallback(&hdma_memtomem_dma1_channel1, HAL_DMA_XFER_CPLT_CB_ID,
DMA_FullTransferCallback);
  HAL_DMA_Start_IT(&hdma_memtomem_dma1_channel1, (uint32_t)&flash_data,
(uint32_t)&sram_buffer, TRANSFER_SIZE);
#else
  HAL_DMA_Start(&hdma_memtomem_dma1_channel1, (uint32_t)&flash_data,
(uint32_t)&sram_buffer, TRANSFER_SIZE);
  HAL_DMA_PollForTransfer(&hdma_memtomem_dma1_channel1, HAL_DMA_FULL_TRANSFER,
HAL_MAX_DELAY);
  HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
#endif
```

In this case, pin LD3 will show a pulse during DMA a full transfer, while LD4 will show the execution time of the 2nd half transfer.

The DMA interrupt indicates time execution of the 2nd half transfer
```

7. Lab: DMA Memory to UART TX

This project demonstrates how DMA co-works with CPU to do parallel tasks:

- CPU will have a main task to blink the LD3 led every 100 ms.
- DMA will transfer a 4KB block of data on UART at 9600 bps.

7.1. Enable DMA on UART TX

Start a new project and enable USART1 with following settings:

- Mode: Asynchronous
- Mode: 9600 bps, 8-bit data, None parity, 1-bit Stop
- In **DMA Settings**, add DMA Request for USART1_TX on DMA1 Channel 2

> When enable DMA on a peripheral, DMA interrupt is automatically enabled under **NVIC** settings, and it is also locked by the option *Force DMA Channels Interrupt*.

7.2. Generated code

The function `MX_DMA_Init()` is generated but just to enable DMA Peripheral's clock and DMA Interrupt lines. The setting for DMA on peripheral is moved to the corresponding peripheral's setup functions in this case, they are `MX_USART1_UART_Init()` → `HAL_UART_Init()` → `HAL_UART_MspInit()` (override in *stm32xxxx_hal_msp.c*):

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(huart->Instance==USART1)
  {
    __HAL_RCC_USART1_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**USART1 GPIO Configuration
    PA9     ------> USART1_TX
    PA10     ------> USART1_RX
    */
    GPIO_InitStruct.Pin = GPIO_PIN_9|GPIO_PIN_10;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF1_USART1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    /* USART1 DMA Init */
    /* USART1_TX Init */
    hdma_usart1_tx.Instance = DMA1_Channel2;
    hdma_usart1_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
```

```
    hdma_usart1_tx.Init.PeriphInc = DMA_PINC_DISABLE;
    hdma_usart1_tx.Init.MemInc = DMA_MINC_ENABLE;
    hdma_usart1_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    hdma_usart1_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
    hdma_usart1_tx.Init.Mode = DMA_NORMAL;
    hdma_usart1_tx.Init.Priority = DMA_PRIORITY_LOW;
    if (HAL_DMA_Init(&hdma_usart1_tx) != HAL_OK)
    {
      Error_Handler();
    }

    __HAL_LINKDMA(huart,hdmatx,hdma_usart1_tx);
  }
}
```

Notice the HAL macro `__HAL_LINKDMA(huart,hdmatx,hdma_usart1_tx)` that is used to link the DMA handler to the peripheral. In fact, it sets the field `huart.hdmatx = hdma_usart1_tx`, and assign `hdma_usart1_tx.Parent = huart`.

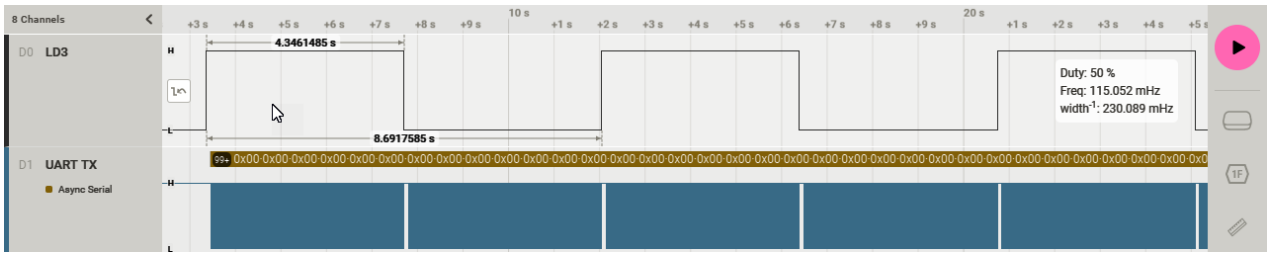
That linking function helps to link the peripheral's callbacks to the DMA callbacks of Half data transfer `hdma->XferHalfCpltCallback()` and Full data transfer `hdma->XferCpltCallback()`.

7.3. User code

For testing, start with the case that CPU should toggle the LD3 pin and then transfer 4KB data on UART1.

```
while (1)
{
  HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
  HAL_Delay(100);
  HAL_UART_Transmit(&huart1, (uint8_t *)&data, TRANSFER_SIZE, HAL_MAX_DELAY);
}
```

Apparently, the LD3 cannot be toggled every 100ms. The transmission on UART takes a long time (~ 4.3s), which causes the execution time of one loop to 4.5 seconds, so it slows down the led blink too much:

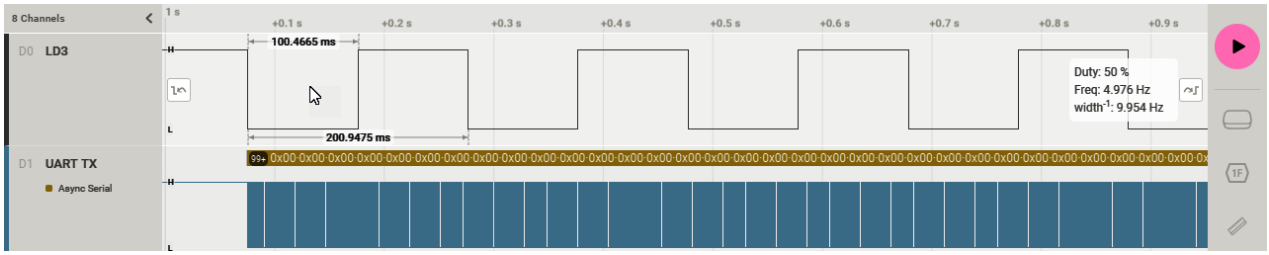


LED blinking rate is affected by UART transmission

However, when use DMA, the toggling time of the LED is remained at 100 ms. While DMA is executing, new DMA start will be ignored.

```
while (1)
{
  HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
  HAL_Delay(100);
#if defined(USE_DMA_FOR_UART_TX)
  HAL_UART_Transmit_DMA(&huart1, (uint8_t *)&data, TRANSFER_SIZE);
#else
  HAL_UART_Transmit(&huart1, (uint8_t *)&data, TRANSFER_SIZE, HAL_MAX_DELAY);
#endif
}
```

Here is the result of using DMA, the LD3 is toggled every 100 ms while UART is transmitting data.



LED blinking is running at desired speed, while UART transmission is handled by DMA

8. Lab: DMA Peripheral to Memory

update soon