LABORATORIO DI ARCHITETTURE E PROGRAMMAZIONE DEI SISTEMI ELETTRONICI INDUSTRIALI

Laboratory Lesson 6:

Direct Memory Access (DMA)

Prof. Luca Benini <luca.benini@unibo.it>

Filippo Casamassima <filippo.casamassima@unibo.it>
Domenico Balsamo <domenico.balsamo@unibo.it>

Course Organization

- Hands-on session LAB1 **Thursday 15.00 – 19.00**
- Prof Benini Friday **9.00 – 11.00 room 5.5**
- Lab is available **Friday 11.00 – 13.00**
- Check website for **announcements, course material:** http://www-micrel.deis.unibo.it/LABARCH
- Final Exam:
  - Homeworks (**to be checked weekly**)
  - Final project
  - Final discussion (homeworks + final project)

DMA -Characteristics

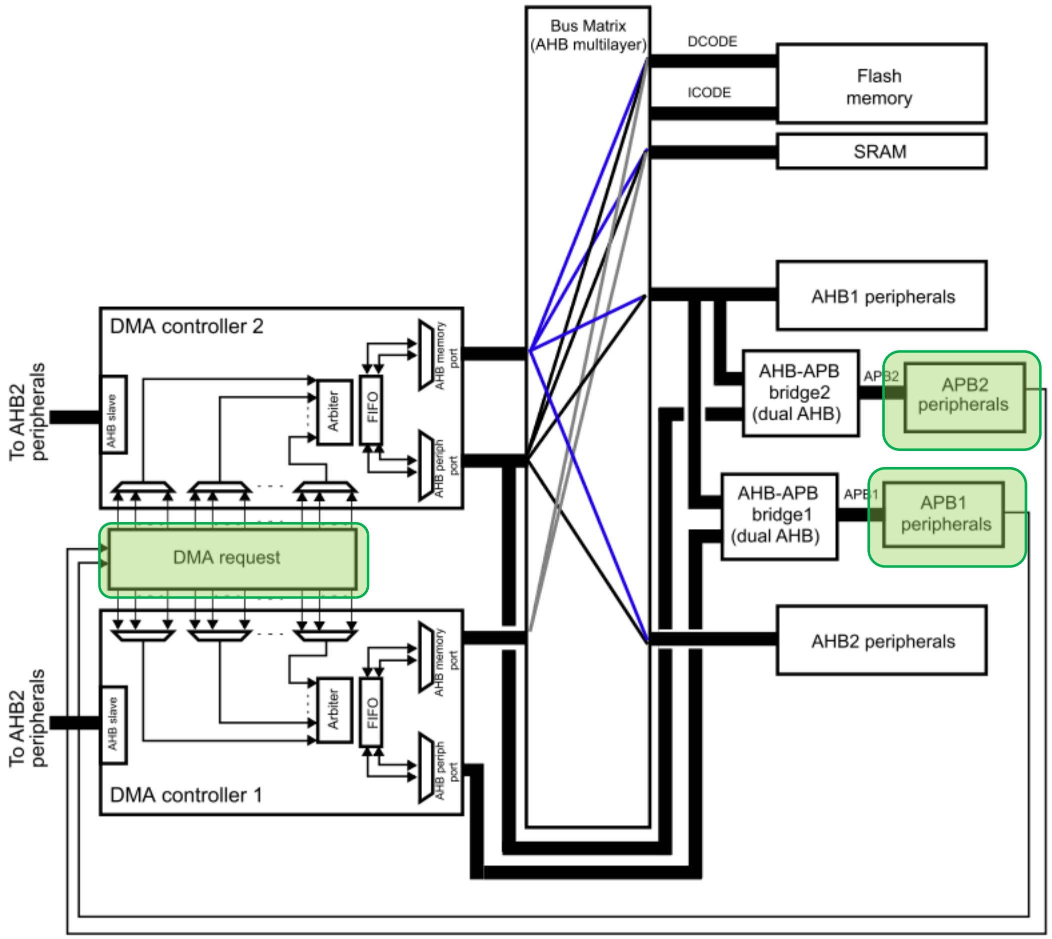
- Direct memory access (DMA) is used in order to provide **high-speed data transfer between peripherals and memory** as well as **memory to memory**. Data can be quickly moved by DMA without any CPU actions. <u>**This keeps CPU resources free for other operations.**</u>

- The two DMA controllers have 16 streams in total (8 for DMA1 and 8 for DMA2), each stream has 8 different channels **each dedicated to managing memory access requests** from one peripheral. It has an arbiter for handling the priority between DMA requests.

- DMA main features:
  - Each of the 16 stream is connected to dedicated hardware DMA requests, software trigger is also supported on each stream. This **configuration is done by software**.
  - Priorities between requests from channels of one DMA are software programmable (**4 levels** consisting of very high, high, medium, low) or hardware in case of equality (request 1 has priority over request 2, etc.)
  - **Independent source and destination transfer size** (byte, half word, word), emulating packing and unpacking. Source/destination addresses must be aligned on the data size.
  - **3 event flags** (<u>DMA Half Transfer, DMA Transfer complete and DMA Transfer Error</u>) logically ORed together in a single interrupt request for each channel
  - **Memory-to-memory transfer**
  - **Peripheral-to-memory and memory-to-peripheral, and peripheral-to-peripheral transfers**

DMA Block scheme

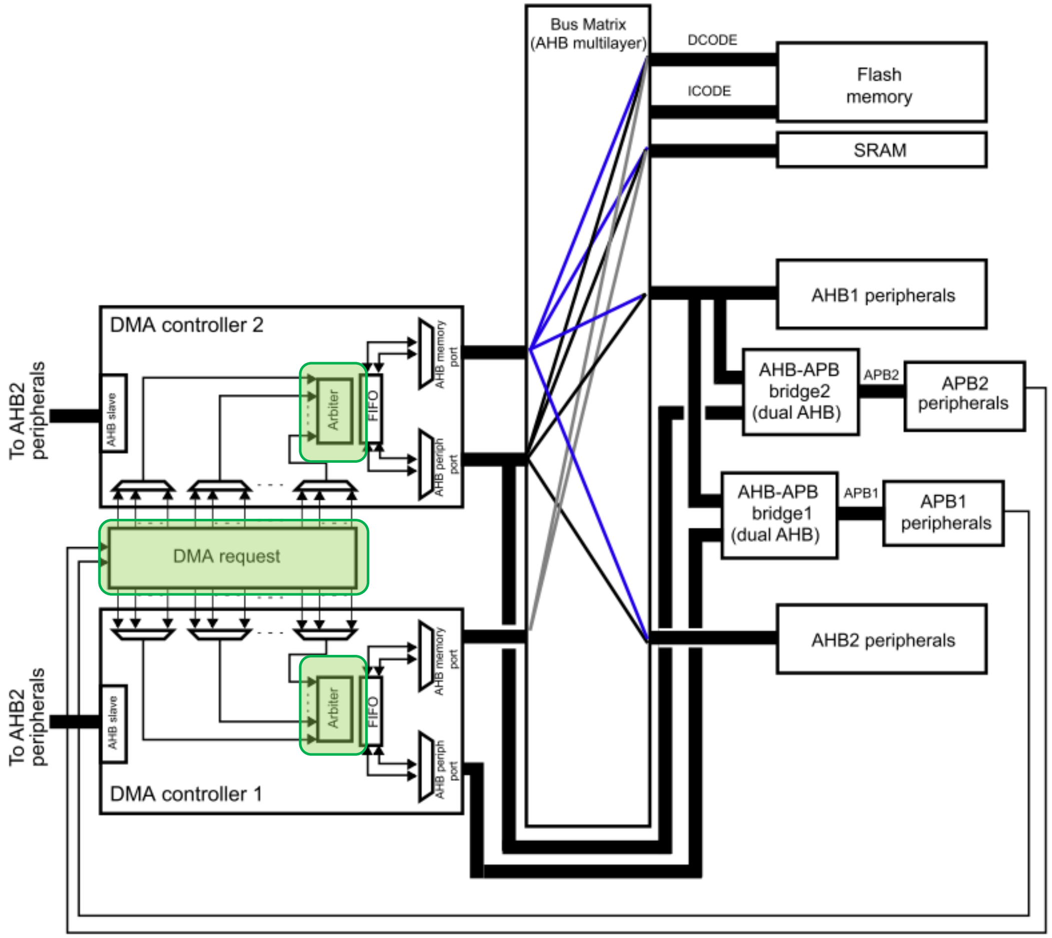
- After an event, the **peripheral sends a request signal to the DMA Controller**.
- 8 channels are multiplexed In one stream.
- The DMA controller serves the request depending on the streams priorities.
- The DMA channels can also work without being triggered by a request from a peripheral. This mode is called **Memory to Memory mode**.



DMA Block scheme

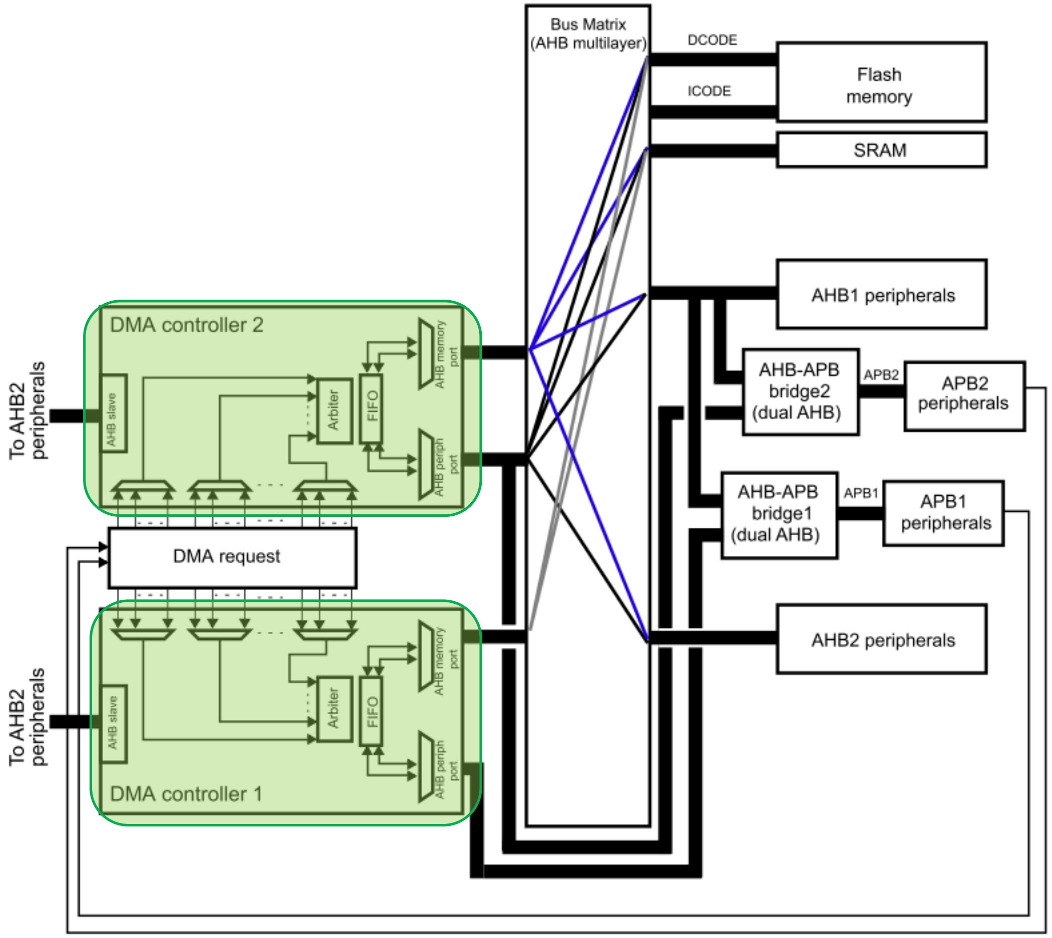
- DMA requests are collected and channel multiplexed.
- The arbiter **manages the tream requests** based on their priority and launches the peripheral/memory access sequences.

- The priorities are managed in two stages:
  - **Software**
  - **Hardware** (if 2 requests have the same software priority level, the channel with the lowest number will get priority versus the channel with the highest number.)



DMA Block scheme

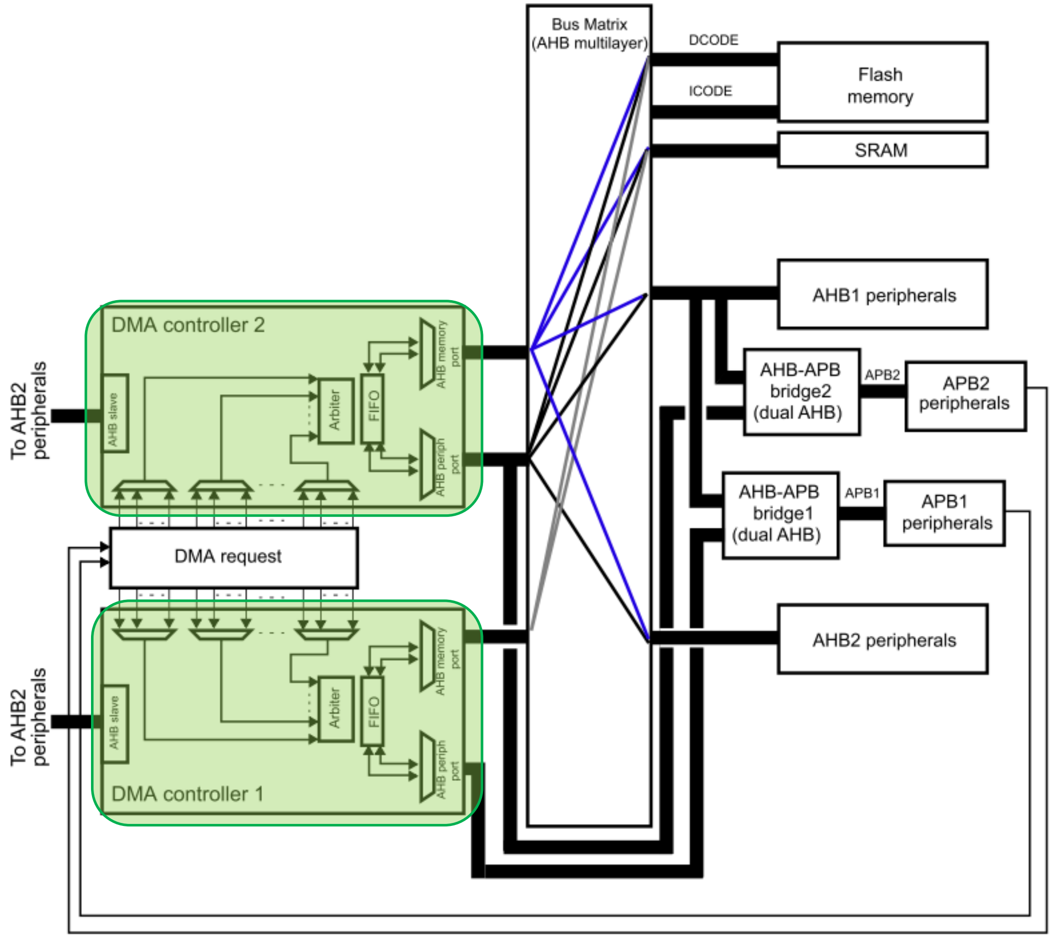
- Each channel can handle DMA transfer **between a peripheral register** located at a fixed address **and a memory address**.
- Peripheral and memory pointers can optionally be **automatically post-incremented** after each transaction.
- If incremented mode is enabled, the **address of the next transfer will be the address of the previous one incremented by 1, 2 or 4** depending on the chosen data size.



DMA Block scheme

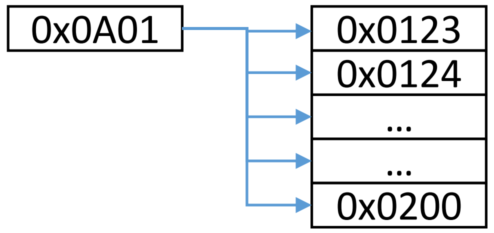
- If the channel is configured in **noncircular mode**, no DMA request is served after the last transfer (that is once the number of data items to be transferred has reached zero).

- **Circular mode** is available to handle **circular buffers and continuous data** flows (e.g. ADC scan mode). When 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.

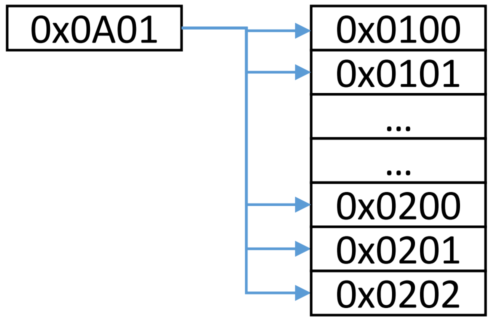


DMA Modes

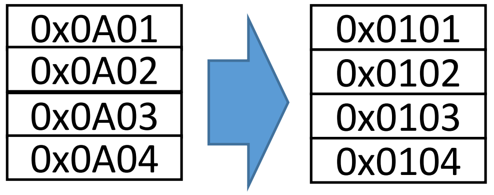
- Pointer incrementation:
  - Peripheral and memory pointers can optionally be automatically post-incremented or kept constant after each transfer



- Double buffer mode
  - This mode has two memory pointers. When the Double buffer mode is enabled, the Circular mode is automatically enabled and at each end of transaction, the memory pointers are swapped.



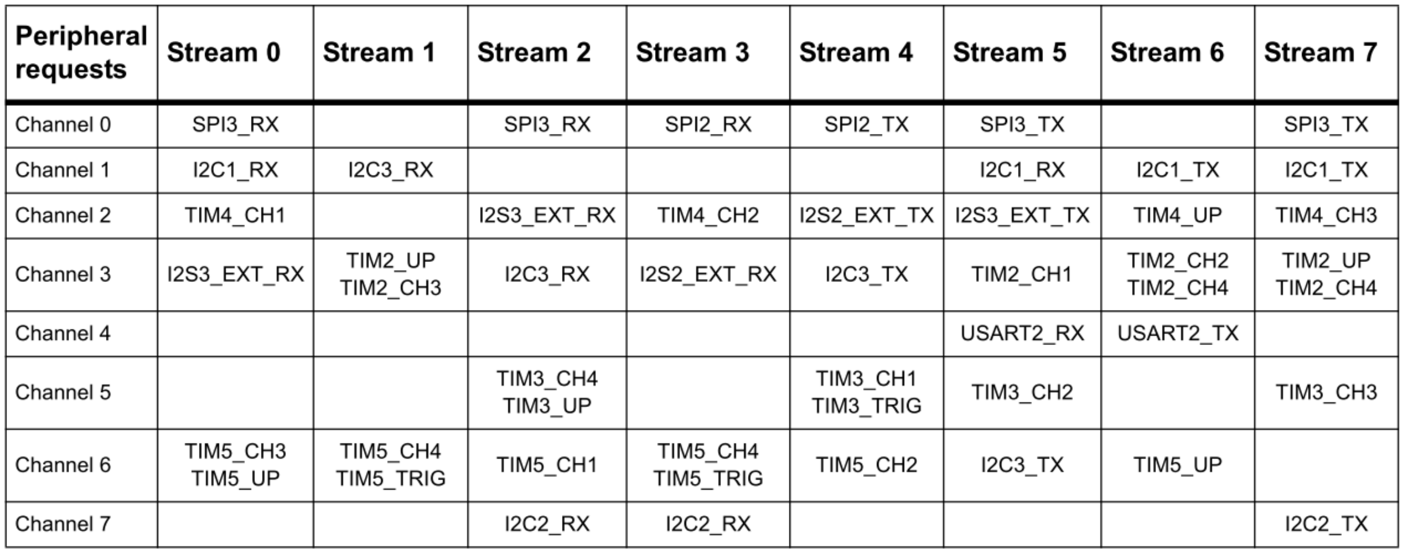
- Single and burst transfers:
  - The DMA controller can generate single transfers or incremental burst transfers of 4, 8 or 16 blocks of data.



DMA channels

- Each Stream is multiplexed up to eight channel, only one channel at a time can be active
- It is not possible to activate two channels on the same stream
- Is possible to perform transfers from all 8 streams

Table 28. DMA1 request mapping (STM32F401xB/C and STM32F401xD/E)



Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

DMA Interrupts

- Each DMA stream can generate up to 5 different interrupt. Each interrupt must be previously configured and enabled trough the NVIC.
- Interrupts are useful to detect errors or when a DMA transfer operation is completed

Interrupt event	Event flag	Enable control bit
Half-transfer	HTIF	HTIE
Transfer complete	TCIF	TCIE
Transfer error	TEIF	TEIE
FIFO overrun/underrun	FEIF	FEIE
Direct mode error	DMEIF	DMEIE

DMA (what)

- We want to use a **DMA** channel to **transfer a word data buffer from FLASH memory to embedded SRAM memory.**

  - **Enable the clock for AHB BUS**
  - **Configure NVIC**
  - **Configure and enable DMA**

    - DMA2 Channel0 Stream0 is configured to transfer the contents of a 32-word data buffer stored in Flash memory to the reception buffer declared in RAM.

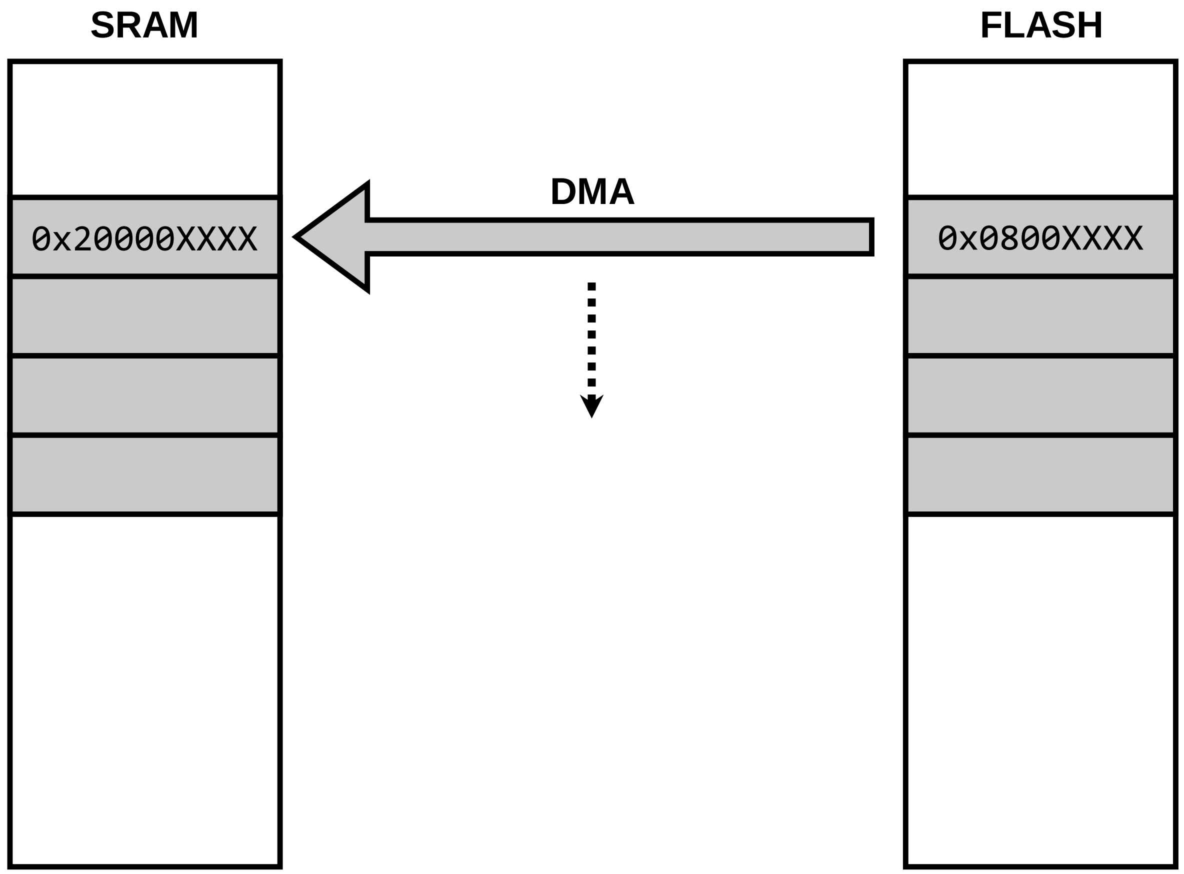
    - The start of transfer is triggered by software. DMA2 Channel0 **memory-to-memory transfer** is **enabled**. Source and destination **addresses incrementing** is also **enabled**.

    - The transfer is **started** by setting the Channel **enable bit** for DMA2 Stream0.

    - At the end of the transfer a Transfer Complete **interrupt is generated** since it is enabled. Once interrupt is generated, the **remaining data** to be transferred is read which must be equal to 0.

    - The Transfer Complete Interrupt **pending bit** is then **cleared**.

DMA (what)



DMA – code

- const uint32_t aSRC_Const_Buffer[BUFFER_SIZE]= {
-                                 0x11111111,0x22222222,0x33333333,0x44444444,
-                                 0x55555555,0x66666666,0x77777777,0x88888888,
- 0x99999999,0xAAAAAAAA,0xBBBBBBBB,0xCCCCCCCC,
-                                 0xDDDDDDDD,0xEEEEEEEE,0xFFFFFFFF,0x00000000,
- 0xCAFFE000,0xDECADECA,0x2CAFFE00,0xDECADECA,
- 0xBADBAD00,0xCADECADE,0x00FACCE0,0x12121212,
-                                 0x23232323,0x34343434,0x696A6B6C,0x6D6E6F70,
- 0x71727374,0x75767778,0x797A7B7C,0x7D7E7F80};
- uint32_t aDST_Buffer[BUFFER_SIZE];

DMA - code

```
DMA_InitTypeDef DMA_InitStructure; /* Usual Structure */

  /* Enable DMA clock */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);

  /* Reset DMA Stream registers (for debug purpose) */
  DMA_DeInit(DMA2_Stream0);

  /* Check if the DMA Stream is disabled before enabling it.
     Note that this step is useful when the same Stream is used multiple
times*/
  while (DMA_GetCmdStatus(DMA2_Stream0) != DISABLE)
  {
  }
```

Necessary only if DMA is used multiple times

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;


  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

We are using Channel 0

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
```

We set the address of the data source (from where data will be read)

```
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
```

We set the address of the data destination (SRAM memory in this case)

```
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
```

Direction of data transfer

```
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
```

Size of DMA transfer (how many words will be transferred)

```
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
```

We increment the source address at each transfer

```
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
```

We increment the destination address at each transfer

```
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
```

Data size of source (word is 32 bits)

```
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
```

Data size of destination (word is 32 bits)

```
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
```

Normal data transfer
i.e. non circular buffer

```
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
```

High priority

```
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
```

FIFO is not used in this case

```
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
```

Burst transfer is not used

```
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

DMA – code

```
/* Configure DMA Structure */
  DMA_InitStructure.DMA_Channel = DMA_Channel_0;
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)aSRC_Const_Buffer;
  DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)aDST_Buffer;
  DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToMemory;
  DMA_InitStructure.DMA_BufferSize = (uint32_t)BUFFER_SIZE;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;


  DMA_Init(DMA2_Stream0, &DMA_InitStructure);
```

The DMA Peripheral is initialized on Stream 0

DMA – code

```
/* Enable DMA Stream Transfer Complete interrupt */
  DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);

  /* DMA Stream enable */
  DMA_Cmd(DMA2_Stream0, ENABLE);

  /* Check if the DMA Stream is correctly configured. */
  while ((DMA_GetCmdStatus(DMA2_Stream0) != ENABLE))
  { }


  } /* end of DMA configuration*/
```

Interrupt Enabled

DMA – code

```
/* Enable DMA Stream Transfer Complete interrupt */
  DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);

  /* DMA Stream enable */
  DMA_Cmd(DMA2_Stream0, ENABLE);

  /* Check if the DMA Stream is correctly configured. */
  while ((DMA_GetCmdStatus(DMA2_Stream0) != ENABLE))
  { }


  } /* end of DMA configuration*/
```

DMA transfer starts here

DMA – code

```
/* Enable DMA Stream Transfer Complete interrupt */
  DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);

  /* DMA Stream enable */
  DMA_Cmd(DMA2_Stream0, ENABLE);

  /* Check if the DMA Stream is correctly configured. */
  while ((DMA_GetCmdStatus(DMA2_Stream0) != ENABLE))
  { }


  } /* end of DMA
```

If there is a problem in configuration, error is catched

DMA – code , NVIC

```c
void NVIC_config(void){
	NVIC_InitTypeDef NVIC_InitStructure;
	/* Enable the DMA Stream IRQ Channel */
	NVIC_InitStructure.NVIC_IRQChannel = DMA2_Stream0_IRQn;
	NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
	NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
	NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
	NVIC_Init(&NVIC_InitStructure);
}
```

Typical NVIC initialization like in previous lessons

DMA – code, MAIN

Configure and starts DMA transfer

```
    DMA_Config();



    NVIC_config();
    while (DMA_GetCmdStatus(DMA2_Stream0) != DISABLE && (Timeout-- > 0))/* Second method */
    {
        /* Check if a timeout condition occurred */
        if (Timeout == 0)
        {
          /* Manage the error: to simplify the code enter an infinite loop */
          while (1)
          { }
        }
      /*
         Since this code present a simple example of how to use DMA, it is just
         waiting on the end of transfer.
         But, while DMA Stream is transferring data, the CPU is free to perform
         other tasks in parallel to the DMA transfer.
      */
    }

  TransferStatus = Buffercmp(aSRC_Const_Buffer, aDST_Buffer, BUFFER_SIZE);
    /* TransferStatus = PASSED, if the transmitted and received data
       are the same */
    /* TransferStatus = FAILED, if the transmitted and received data
       are different */

    if (TransferStatus != FAILED)
    {
      /* Turn LED4 on: Transfer correct */
      STM_EVAL_LEDOn(LED4);
    }
```

DMA – code, MAIN

```
DMA_Config();
NVIC_config();
```

Configure the interrupt

```
while (DMA_GetCmdStatus(DMA2_Stream0) != DISABLE && (Timeout-- > 0))/* Second method */
{
    /* Check if a timeout condition occurred */
    if (Timeout == 0)
    {
      /* Manage the error: to simplify the code enter an infinite loop */
      while (1)
      { }
    }
  /*
     Since this code present a simple example of how to use DMA, it is just
     waiting on the end of transfer.
     But, while DMA Stream is transferring data, the CPU is free to perform
     other tasks in parallel to the DMA transfer.
  */
}

TransferStatus = Buffercmp(aSRC_Const_Buffer, aDST_Buffer, BUFFER_SIZE);
  /* TransferStatus = PASSED, if the transmitted and received data
     are the same */
  /* TransferStatus = FAILED, if the transmitted and received data
     are different */

  if (TransferStatus != FAILED)
  {
    /* Turn LED4 on: Transfer correct */
    STM_EVAL_LEDOn(LED4);
  }
```

DMA – code, MAIN

```
    DMA_Config();
    NVIC_config();


    while (DMA_GetCmdStatus(DMA2_Stream0) != DISABLE && (Timeout-- > 0))/* Second method */
    {
        /* Check if a timeout condition occurred */
        if (Timeout == 0)
        {
          /* Manage the error: to simplify the code enter an infinite loop */
          while (1)
          { }
        }
      /*
         Since this code present a simple example of how to use D
         waiting on the end of transfer.
         But, while DMA Stream is transferring data, the CPU is fr
         other tasks in parallel to the DMA transfer.
      */
    }

  TransferStatus = Buffercmp(aSRC_Const_Buffer, aDST_Buffer, BUFFER_SIZE);
    /* TransferStatus = PASSED, if the transmitted and received data
       are the same */
    /* TransferStatus = FAILED, if the transmitted and received data
       are different */

    if (TransferStatus != FAILED)
    {
      /* Turn LED4 on: Transfer correct */
      STM_EVAL_LEDOn(LED4);
    }
```

Wait till DMA transfer is complete or timeout reaches zero

DMA – code, MAIN

```
    DMA_Config();
    NVIC_config();


    while (DMA_GetCmdStatus(DMA2_Stream0) != DISABLE && (Timeout-- > 0))/* Second method */
    {
        /* Check if a timeout condition occurred */
        if (Timeout == 0)
        {
          /* Manage the error: to simplify the code enter an infinite loop */
          while (1)
          { }
        }
      /*
         Since this code present a simple example of how to use DMA, it is just
         waiting on the end of transfer.
         But, while DMA Stream is transferring data, the CPU is free to perform
         other tasks in parallel to the DMA transfer.
      */
    }

  TransferStatus = Buffercmp(aSRC_Const_Buffer, aDST_Buffer, BUFFER_SIZE);
    /* TransferStatus = PASSED, if the transmitted and received data
       are the same */
    /* TransferStatus = FAILED, if the transmitted and received data
       are different */

    if (TransferStatus != FAILED)
    {
      /* Turn LED4 on: Transfer correct */
      STM_EVAL_LEDOn(LED4);
    }
```

Check if data in destination are equal to data in source buffer

DMA – code Interrupt

```
void DMA2_Stream0_IRQHandler(void)
{
  /* Test on DMA Stream Transfer Complete interrupt */
  if(DMA_GetITStatus(DMA2_Stream0, DMA_IT_TCIF0))
  {
    /* Clear DMA Stream Transfer Complete interrupt pending bit */
    DMA_ClearITPendingBit(DMA2_Stream0, DMA_IT_TCIF0);

    /* Turn LED3 on: End of Transfer */
    STM_EVAL_LEDOn(LED3);
  }
}
```

Interrupt Service Routine

DMA Esercizi

- 6.1 Crea un array da 100 char nella flash e usa il DMA per trasferirlo nella SRAM
- 6.2 Riscrivi il codice per effettuare la copia dei dati usando il codice (non usando il DMA)
- 6.3 Misrura il numero di cicli di clock necessario per effettuare il trasferimento dei dati, con DMA e con il codice scritto sopra inserisci nel codice dei commenti relative ai risultati di tale misura. Per farlo puoi usare il seguente snippet:

```
int count = 0;
volatile unsigned int *DWT_CYCCNT = (int *)0xE0001004; //address of the Cycle Count Register
volatile unsigned int *DWT_CONTROL = (int *)0xE0001000; //address of the Control Register
volatile unsigned int *SCB_DEMCR = (int *)0xE000EDFC; //address of the debug control Register

*SCB_DEMCR = *SCB_DEMCR | 0x01000000;
*DWT_CYCCNT = 0; // reset the counter
*DWT_CONTROL = *DWT_CONTROL | 1 ; // enable the counter

/* Functions / operations to monitor */

count= *DWT_CYCCNT; //count contains the result in number of clock cycles
```

DMA Domande

- 6.a È possibile effettuare un trasferimento dati dalla SRAM alla FLASH con il DMA ? Perchè?
- 6.b Prova a ridurre il valore di TIMEOUT_MAX fino ad 1, cosa succede, il codice si blocca nel while? Perchè?