STM32 - Direct Memory Access Management

DMA is used as an independent memory transferring module between memory and peripherals that frees CPU from handling data exchange, therefore it speeds up the system performance. DMA is used in many cases to handling data stream and continuous transferring.

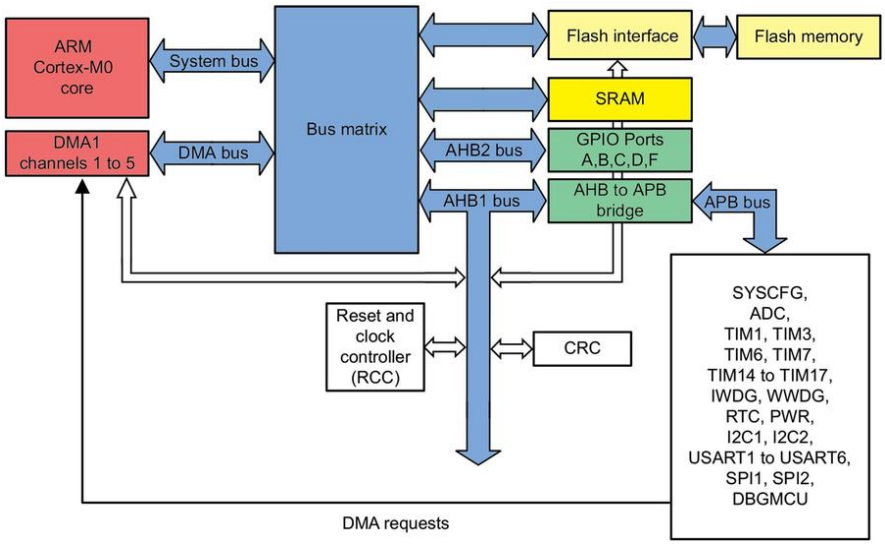
#arm #stm32 #dma

Last update: 2021-08-10 16:50:49

Table of Content

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

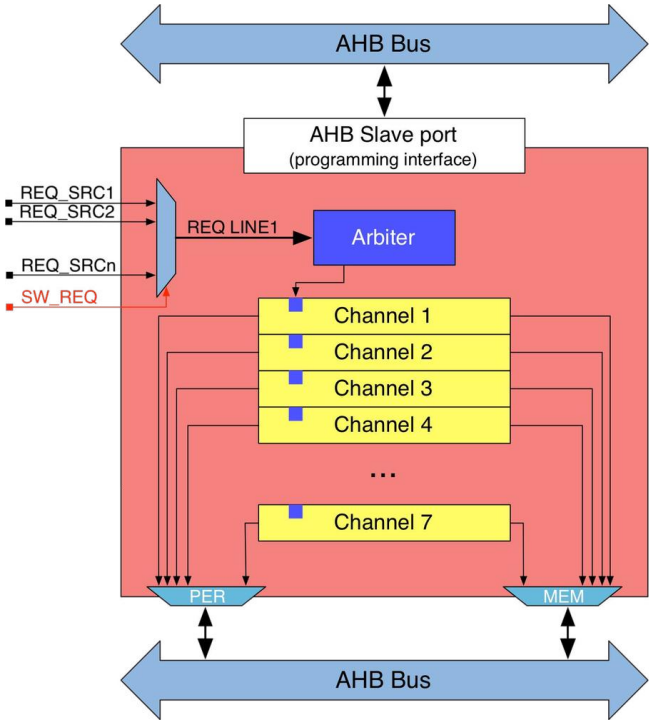
Some important things about DMA:

- 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

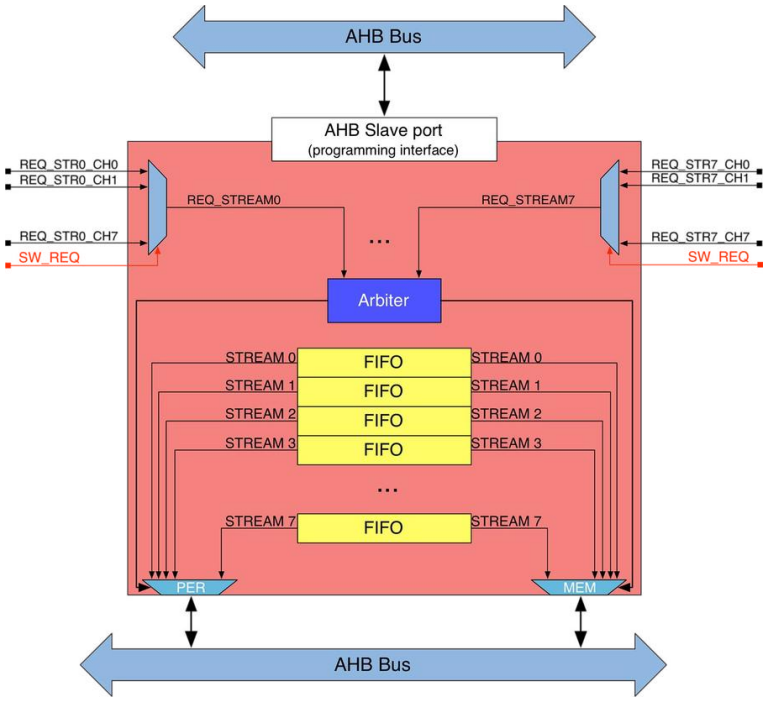
In every STM32 MCU, the DMA controller is a hardware unit that:

- 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 assigning 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*



The DMA architecture in an STM32 F0/F1/F3/L1 MCUs



The DMA architecture in an STM32 F2/F4/F7 MCU

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` (peripheral address increment) and `MINC` (memory address increment) 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

From the performance point of view, the DMA transfer in polling mode is meaningless as there is no reason to start a DMA transfer then consume a lot of CPU cycles just to wait for the transfer completion. So the best option is to arm the DMA and let it notify when the transfer is completed.

The DMA is able to generate interrupts related to channel activities (for example, the DMA1 in an STM32F030 MCU has one IRQ for channel 1, one for channels 2 and 3, one for channels 4 and 5). Moreover, three independent enable bits are available to enable IRQ on half transfer, full transfer and transfer error. Separate interrupt enable bits are available for flexibility.

5. STM32 CubeHAL Usage

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 on 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 a callback function by assigning 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 in byte-to-byte mode.

Requirement:

- Create a 4 KB data block in flash memory (slow speed)
- Create a 4 KB buffer in SRAM memory (high speed)
- Disable code optimization in compilation
- 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 analyzer

Target board:

Any board has GPIOs to output execution time pulses.

STM32F051R8	**Mode**
PC8	Output Push Pull, No Pull-up and No Pull-down
PC9	Output Push Pull, No Pull-up and No Pull-down

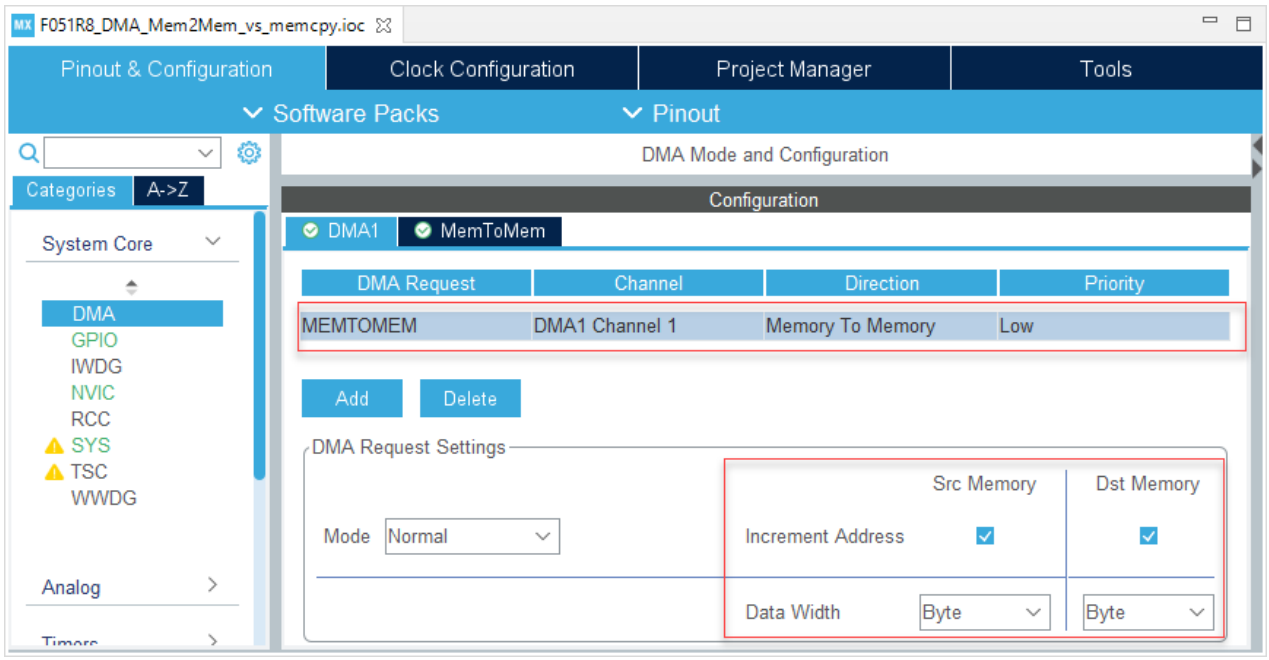
6.1. Start new project

If start a new project on an ST's Discovery / Nucleo board, select to use default settings for the board.

If start a new custom project, make sure to set up the system clock, and programmer interface.

6.2. Enable DMA

1. Go to **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

6.3. Generated code

An instance of `DMA_HandleTypeDef hdma_m2m_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:

- Enable DMA clock
- Setup DMA properties: Channel / Mode / Memory Address Increment / Data size /Priority

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

  /* Configure DMA request hdma_m2m_dma1_channel1 on DMA1_Channel1 */
  hdma_m2m_dma1_channel1.Instance = DMA1_Channel1;
  hdma_m2m_dma1_channel1.Init.Direction = DMA_MEMORY_TO_MEMORY;
  hdma_m2m_dma1_channel1.Init.PeriphInc = DMA_PINC_ENABLE;
  hdma_m2m_dma1_channel1.Init.MemInc = DMA_MINC_ENABLE;
  hdma_m2m_dma1_channel1.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
  hdma_m2m_dma1_channel1.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  hdma_m2m_dma1_channel1.Init.Mode = DMA_NORMAL;
  hdma_m2m_dma1_channel1.Init.Priority = DMA_PRIORITY_LOW;

  if (HAL_DMA_Init(&hdma_m2m_dma1_channel1) != HAL_OK) {
    Error_Handler( );
  }
}
```

6.4. User code

Declare memory block

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

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

Always allocated memory for DMA at the *global scope*

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

Measure execution time

To measure the execution time in a logic analyzer, output a pulse on **PC9** during the transfers. This pin is set to LOW at the startup, and set to HIGH when starting transfer, and set back to LOW when the transfer completes.

Test the CPU `memcpy()` function

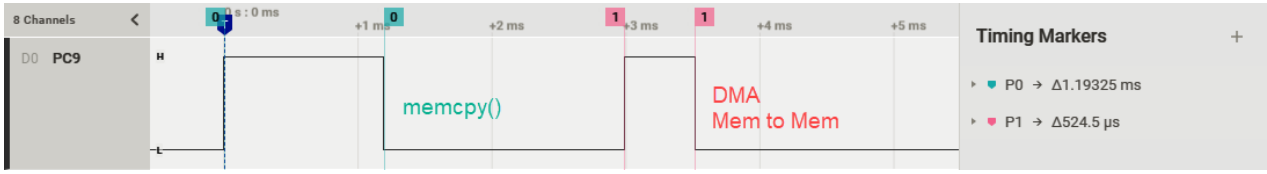
```
HAL_Delay(1);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_SET);
memcpy(&sram_buffer, &flash_data, TRANSFER_SIZE);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_RESET);
```

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(GPIOC, GPIO_PIN_9, GPIO_PIN_SET);
HAL_DMA_Start(&hdma_m2m_dma1_channel1,
                (uint32_t)&flash_data,
                (uint32_t)&sram_buffer,
                TRANSFER_SIZE);
HAL_DMA_PollForTransfer(&hdma_m2m_dma1_channel1,
                            HAL_DMA_FULL_TRANSFER,
                            HAL_MAX_DELAY);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_RESET);
```

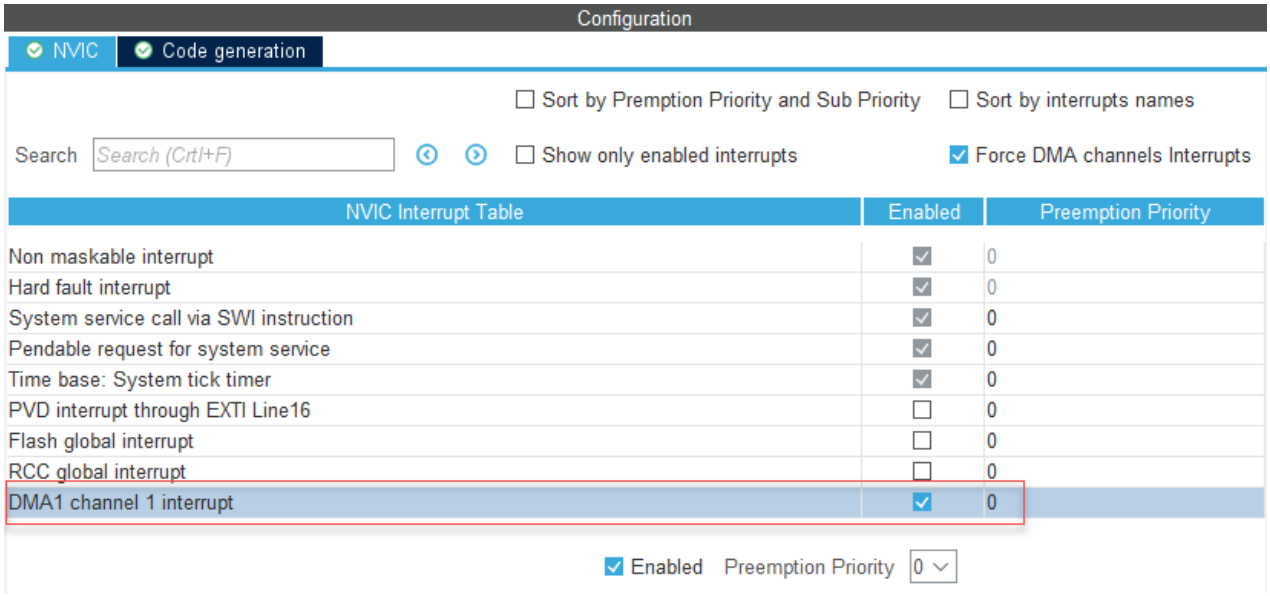
Here are the output pulses showing that `memcpy()` needs 1193 us to complete while DMA only needs 525 us.



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

6.5. DMA 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 generated code will add into an IRQ handler in the `stm32xxxx_it.c` file:

```
void DMA1_Channel1_IRQHandler(void) {
  HAL_DMA_IRQHandler(&hdma_m2m_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, these callbacks are not set, therefore, write 2 functions to handle DMA interrupts. Note that, a new GPIO **PC8** is used to show the last half transfer time.

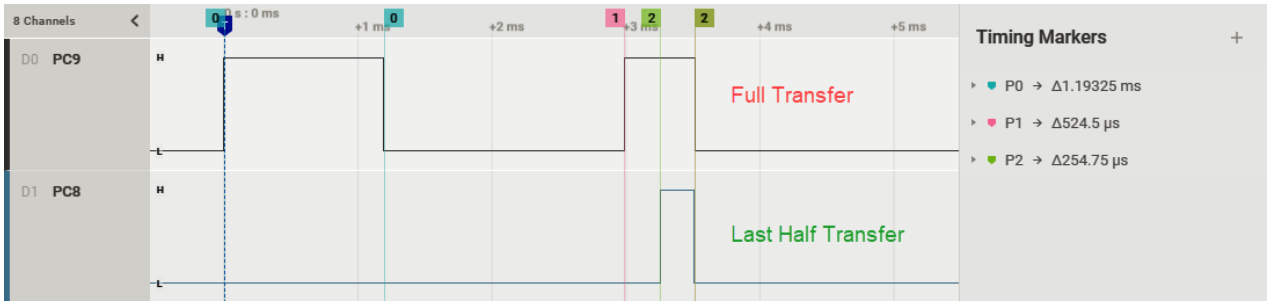
```
void DMA_HalfTransferCallback(DMA_HandleTypeDef * _hdma) {
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
}
void DMA_FullTransferCallback(DMA_HandleTypeDef * _hdma) {
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_RESET);
}
```

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

```
int main(void)
{

``` cpp
  HAL_Delay(1);
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_SET);
#ifdef USE_DMA_INTERRUPT
  HAL_DMA_RegisterCallback(&hdma_m2m_dma1_channel1,
                           HAL_DMA_XFER_HALFCPLT_CB_ID,
                           DMA_HalfTransferCallback);
  HAL_DMA_RegisterCallback(&hdma_m2m_dma1_channel1,
                           HAL_DMA_XFER_CPLT_CB_ID,
                           DMA_FullTransferCallback);
  HAL_DMA_Start_IT(&hdma_m2m_dma1_channel1,
                       (uint32_t)&flash_data,
                       (uint32_t)&sram_buffer,
                       TRANSFER_SIZE);
#else
  /* DMA Polling mode */
#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.



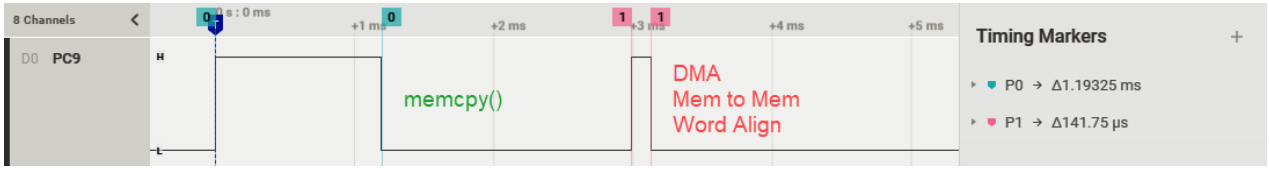
The DMA interrupt indicates time execution of the 2nd half transfer
```

6.6. DMA Data size

Using Word-aligned memory block to speed up the DMA transfer. Note that transfer size will be reduced.

```
static void MX_DMA_Init(void) {
    ...
    hdma_m2m_dma1_channel1.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
    hdma_m2m_dma1_channel1.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
    ...
}
int main(void) {
    HAL_DMA_Start_IT(&hdma_m2m_dma1_channel1,
                     (uint32_t)&flash_data,
                     (uint32_t)&sram_buffer,
                     TRANSFER_SIZE/4);
}
```

As seen in below image, the DMA transferring time in Word-aligned mode is only 142us, comparing the 525us in Byte-aligned mode.



Using World-aligned to reduce DMA transfer time