stm32f429-DMA Tutorial

제작자: 유영재

목차

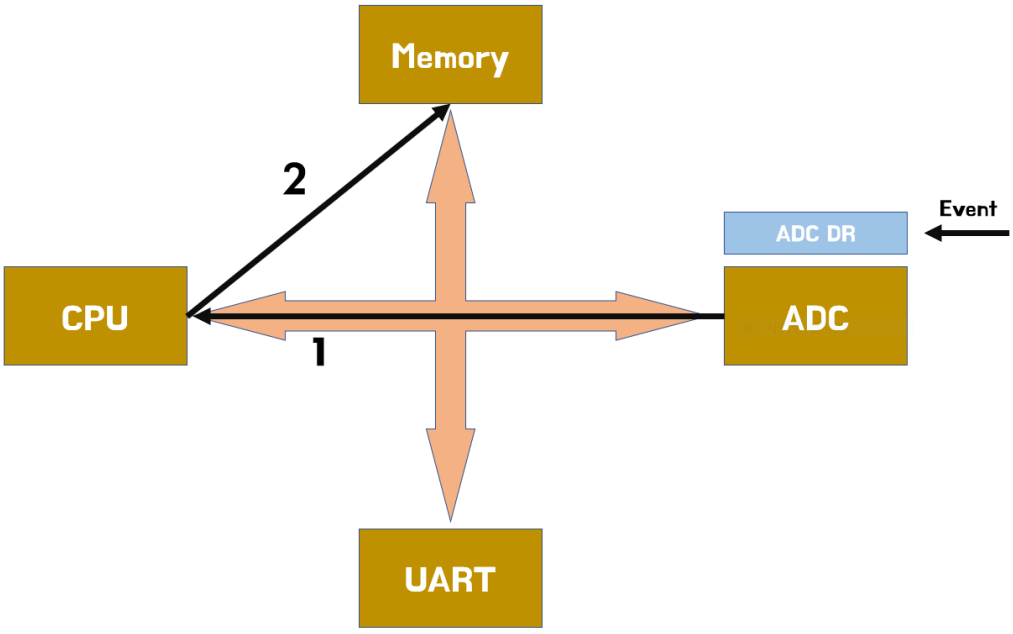
1. DMA 소개 3
1.1 Master-slave system 3
1.2 DMA 사용 예 4
1.3 MCU Block Diagram 5
1.4 Bus matrix 10

1. DMA 소개

- DMA 는 Direct Memory Access 의 약자로 최근 MCU 에는 기본적으로 들어가 있는 Peripheral 중 하나다. 현재 우리가 다룰 arm-cortex M4 기반의 stm32f429 도 총 2 개의 dma 컨트롤러가 내장되어 있다. dma 를 이해하기 위해선 arm-cortex 의 버스 시스템을 이해하고 있어야 한다

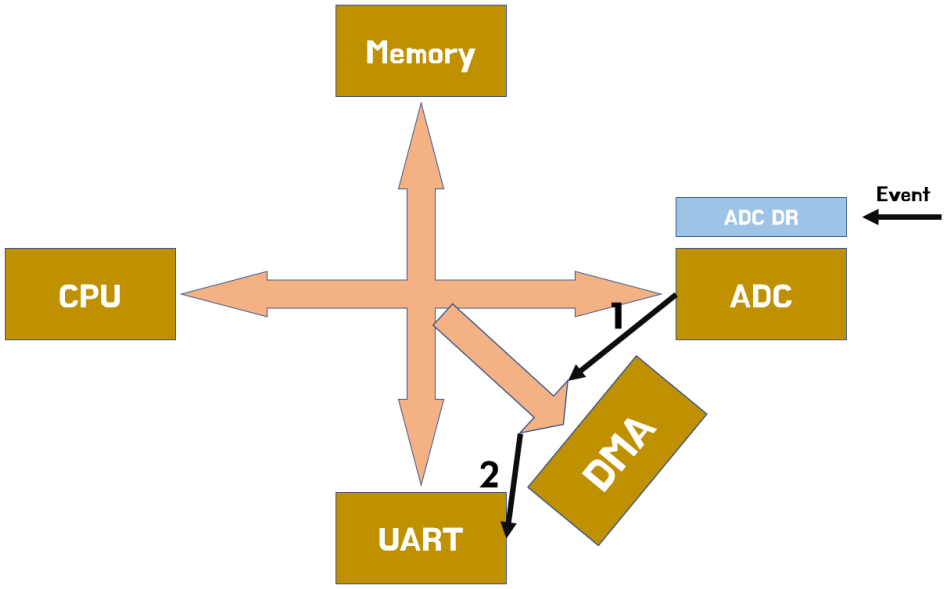
1.1 Master-slave system

아래 그림은 ARM의 버스 시스템을 나타낸 것이다. ARM 버스 시스템은 기본적으로 CPU와 주변 Peripheral들은 버스로 연결되어 있는 것을 볼 수 있다



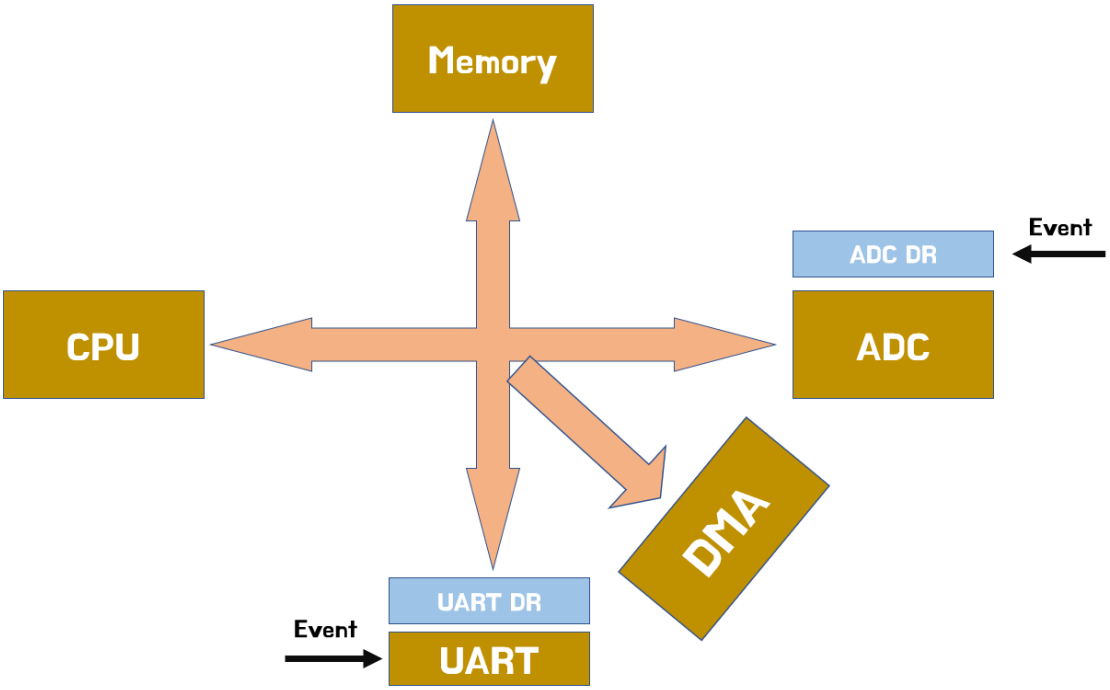
A. 하나의 상황을 예를 들어서 설명을 해보려고 한다. ADC에 이벤트가 발생해서 digitize 된 값이 ADC 데이터 레지스터로 써지게 된다
B. 그러면 ADC는 CPU에게 Event를 발생시켜서 CPU는 해당 데이터를 프로세서 레지스터에 저장한 이후 SRAM 메모리에 쓸 수 있게 된다
C. 위에서 설명했듯이 총 2개의 경로로 데이터는 메모리에 써지게 된다. 그러면 바로 ADC DR 값이 메모리로 쓸 수는 없을까?
D. 답은 그렇게 할 수 없다. 왜냐하면 명령어 중 하나인 load는 CPU만 가지고 있기 때문이다. 즉 각 주변장치들은 두뇌가 없기 때문에 해당 명령을 실행할 수 없다. 그래서 매번 CPU의 도움을 받기 위해서 Event를 발생시키는 것이다
E. 그래서 버스 시스템에서 버스를 컨트롤 할 수 있는 기능을 가지고 있는 것은 master라고 부르며, 아닌 것은 slave라고 지칭한다. 그래서 slave끼리는 버스를 끼고 데이터 공유가 불가능한 것이다
F. 그러면 CPU 말고도 master 역할을 대신 할 수 있는 것은 없나? 그 역할을 하는 것이 바로 DMA controller다.

G. DMA 컨트롤러가 master라고 지칭할 수 있어도, 명령어(instruction)를 제공하는 장치는 CPU밖에 존재하지 않는다. 다만 DMA가 하드웨어 로직으로 CPU의 일을 offload할 수 있게 설계되어 있을 뿐이다. 그래서 도움을 받아 버스를 control 할 수 있게 된다



1.2 DMA 사용 예

- DMA가 CPU load를 덜어주는 역할을 한다고 하지만, 언제 사용하게 되는가? 단순하게 CPU가 하나의 일만 수행을 한다면 굳이 사용할 필요는 없다. 다음과 같은 예를 들어보기로 한다



A. 두 인터럽트가 동시에 config되어 있고, ADC, UART 모두 동시에 들어오는 상황이라고 가정을 한다. 인터럽트로 받은 데이터를 모두 메모리에 저장해야 하는 상황이다
B. ADC 우선순위가 더 높은 상황이고, 데이터는 두 주변장치에서 지속적으로 들어오고 있다. 그래서 ADC는 데이터가 지속적으로 SRAM에 저장되지만, UART는 그렇지 않다
C. 이럴 때 UART 데이터 손실을 막기 위해서 UART 데이터는 DMA 컨트롤러로 처리를 하게 하는 것이다. 따라서 DMA를 거쳐 SRAM으로 저장되게 된다

- 그리고 두번째 이유로 뒤에 실험을 하겠지만, ARM ON일 때와 ARM OFF/DMA ON일 때의 전류를 비교하면 차이가 생기게 된다. 전력을 아끼는 쪽이 DMA이기 때문에 좋은 Application에는 DMA 사용이 반드시 따라오게 된다

1.3 MCU Block Diagram

MCU Block Diagram은 DMA를 이해하는 데 반드시 숙지해야 하는 부분이다. RM 문서 2장 Memory and bus architecture을 보면 위에서 언급한 master/slave 개념이 드러나고 있다. stm32f429에는 2개의 DMA를 포함해서 10개의 master 버스가 존재한다

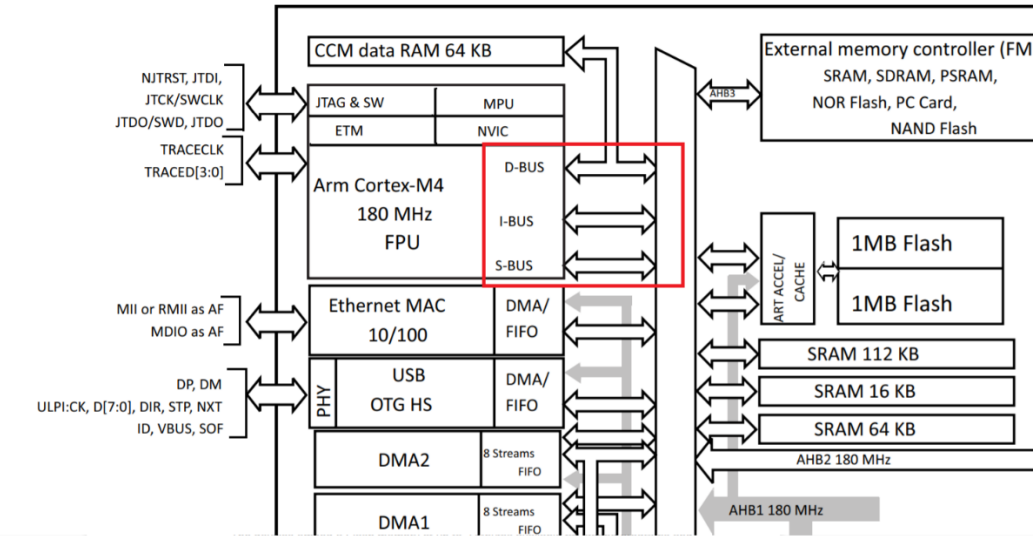
In the STM32F42xx and STM32F43xx devices, the main system consists of 32-bit multilayer AHB bus matrix that interconnects:

- Ten masters:
  - Cortex®-M4 with FPU core I-bus, D-bus and S-bus
  - DMA1 memory bus
  - DMA2 memory bus
  - DMA2 peripheral bus
  - Ethernet DMA bus
  - USB OTG HS DMA bus
  - LCD Controller DMA-bus
  - DMA2D (Chrom-Art Accelerator™) memory bus
- Eight slaves:
  - Internal Flash memory ICode bus
  - Internal Flash memory DCode bus
  - Main internal SRAM1 (112 KB)
  - Auxiliary internal SRAM2 (16 KB)
  - Auxiliary internal SRAM3 (64 KB)
  - AHB1peripherals including AHB to APB bridges and APB peripherals
  - AHB2 peripherals
  - FMC

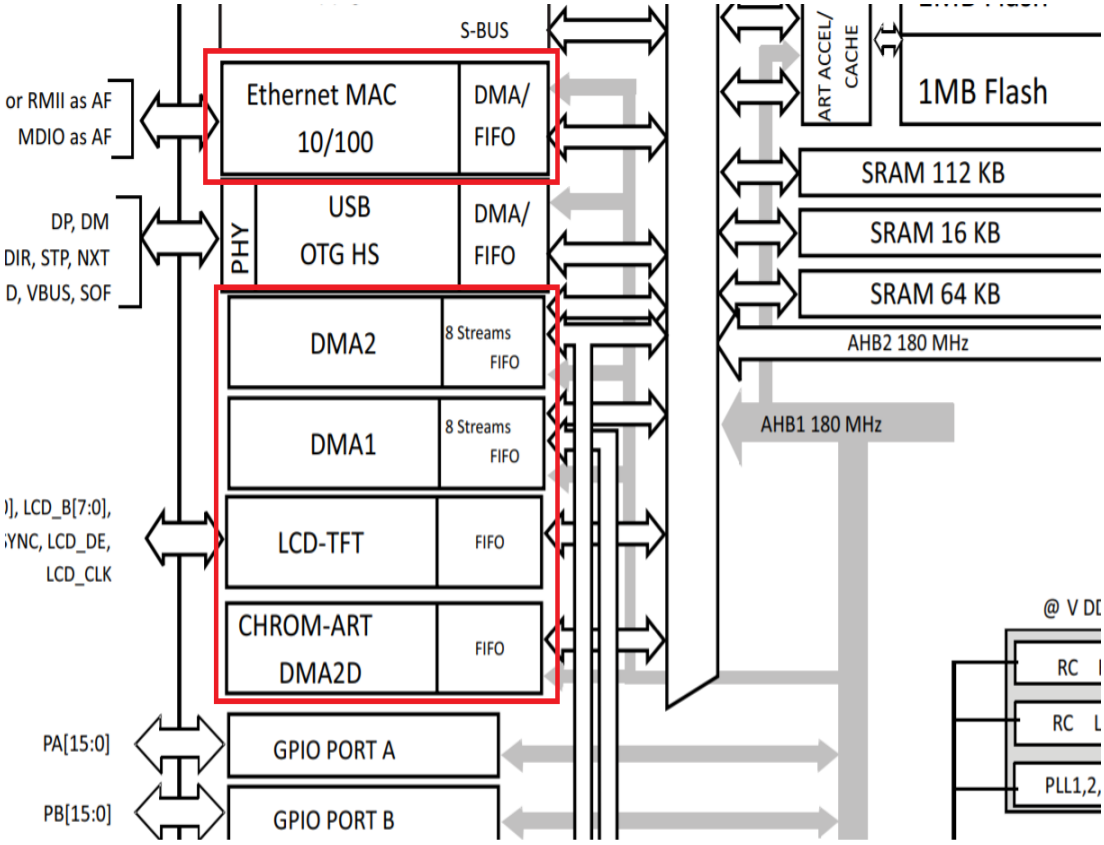
The bus matrix provides access from a master to a slave, enabling concurrent access and efficient operation even when several high-speed peripherals work simultaneously. The 64-Kbyte CCM (core coupled memory) data RAM is not part of the bus matrix and can be accessed only through the CPU. This architecture is shown in *Figure 2*.

A. Master

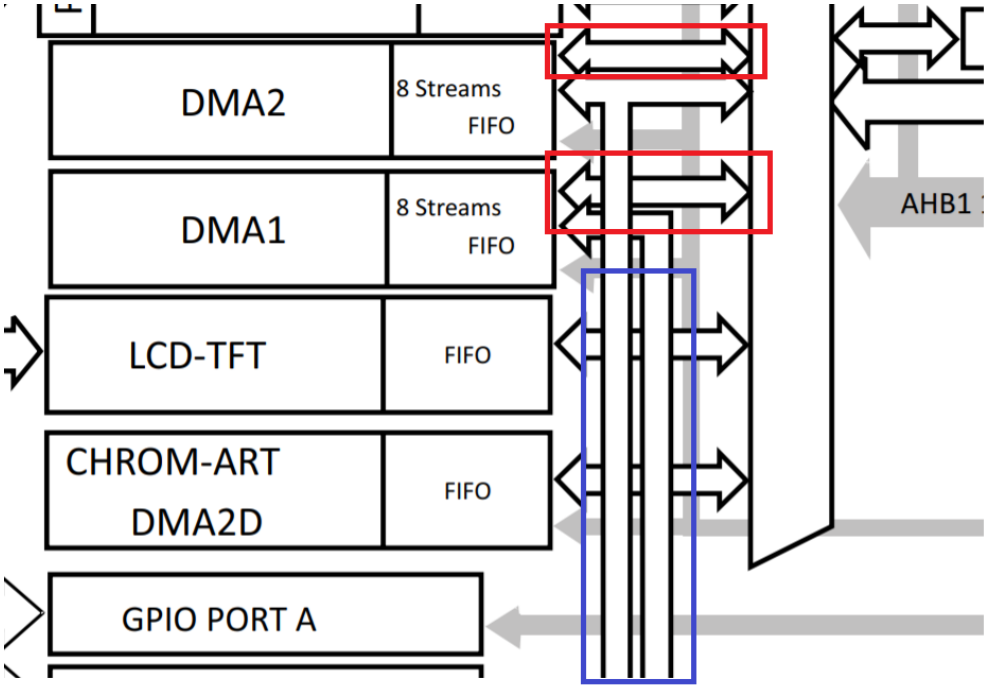
a. Core I-bus, S-bus, D-bus



b. 다른 Master



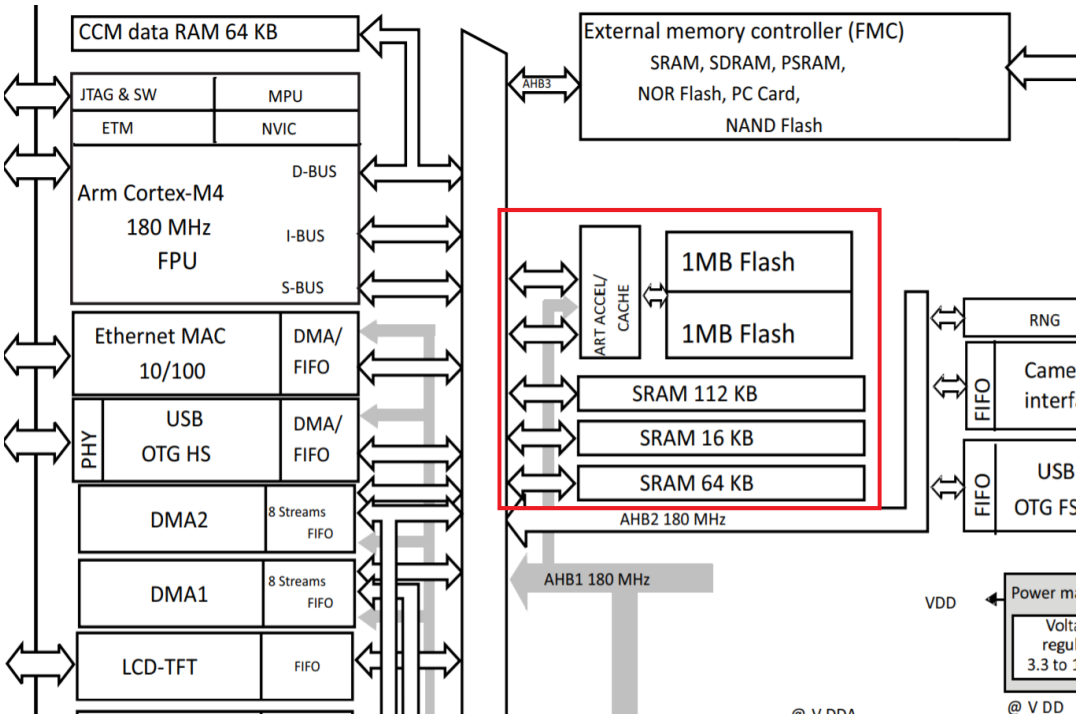
참고로 DMA controller에서 두 갈래의 길로 나눠진다. 하나는 memory쪽 버스와 나머지는 peripheral쪽 버스가 된다. 아래 버스에서 빨간색이 memory, 파란색이 peripheral로 해당된다



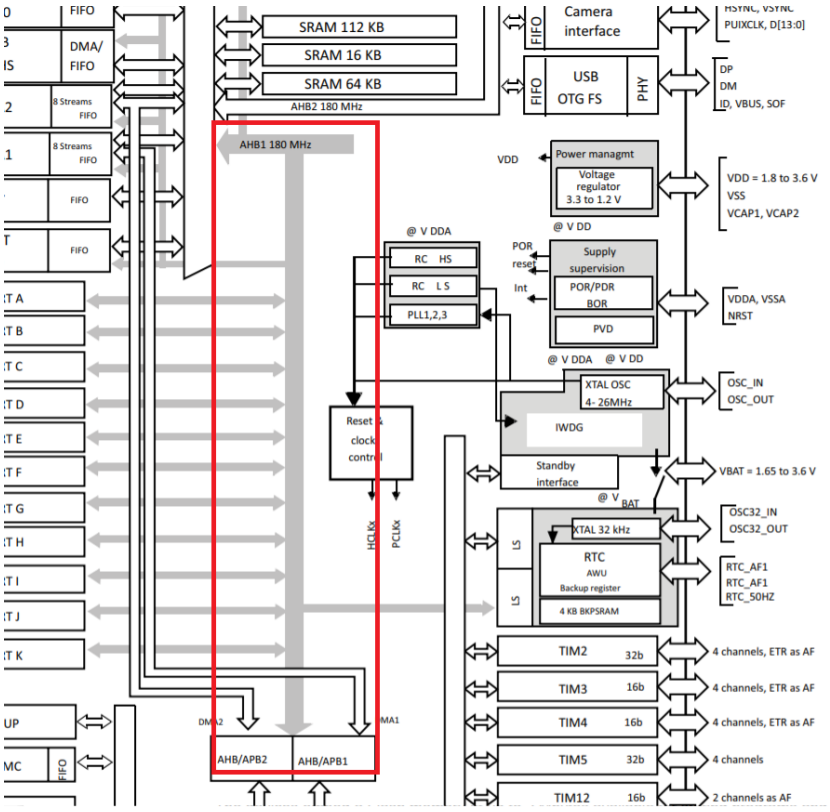
따라서 DMA 컨트롤러를 통해서 M2M, P2M, M2P가 가능하게 된다

B. Slave

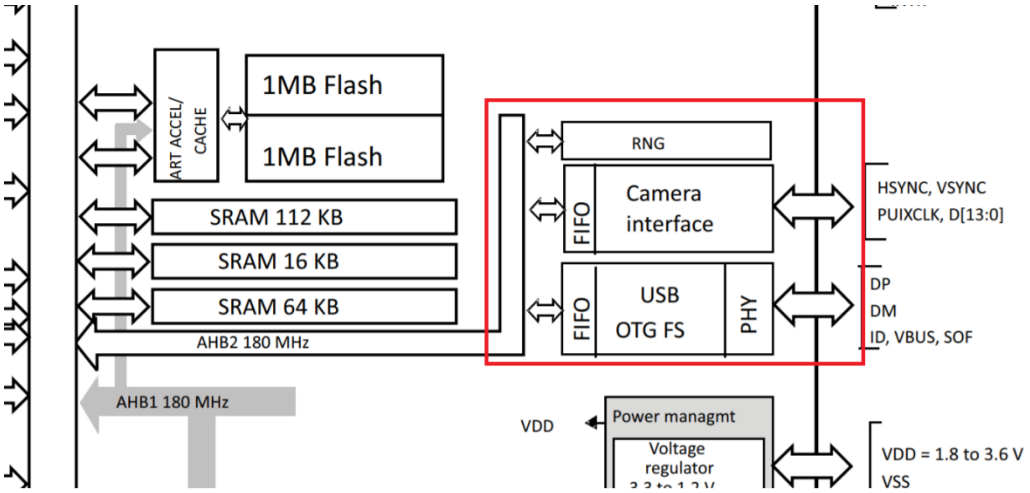
a. Flash I-code, D-code, SRAM



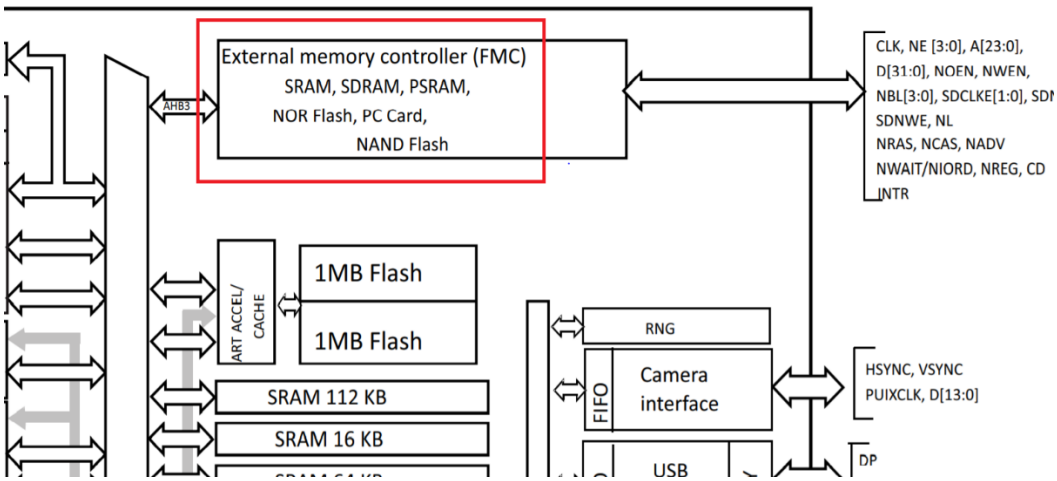
b. AHB1은 APB1, 2 버스를 포함하고 있다



c. AHB2


ART ACCEL/ CACHE
1MB Flash
1MB Flash
SRAM 112 KB
SRAM 16 KB
SRAM 64 KB
AHB2 180 MHz
AHB1 180 MHz
RNG
FIFO
Camera interface
FIFO
USB OTG FS
PHY
HSYNC, VSYNC
PUIXCLK, D[13:0]
DP
DM
ID, VBUS, SOF
VDD
Power managmt
Voltage regulator
VDD = 1.8 to 3.6 V
VSS

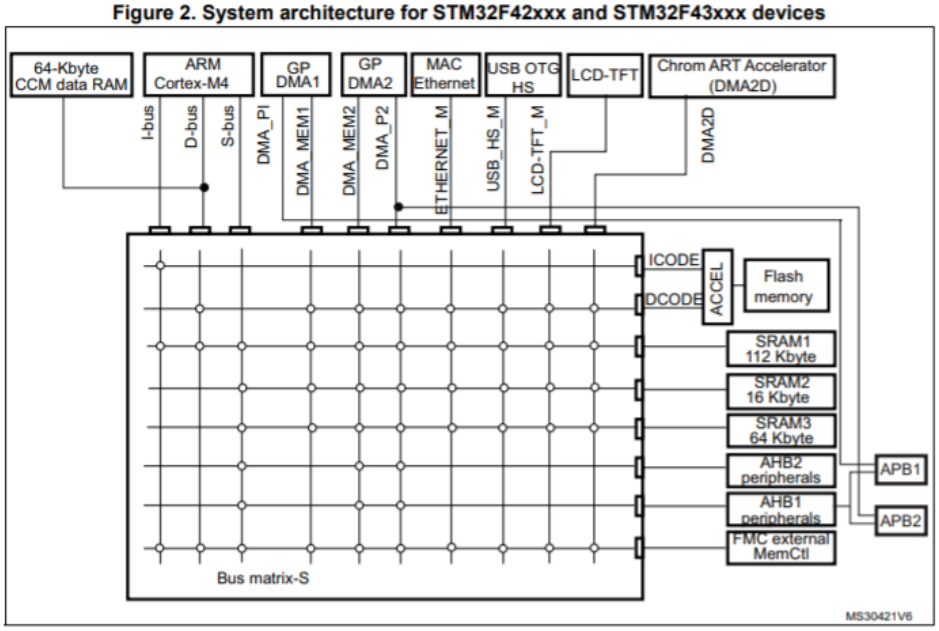
d. FMC


External memory controller (FMC)
SRAM, SDRAM, PSRAM,
NOR Flash, PC Card,
NAND Flash
AHB3
CLK, NE [3:0], A[23:0],
D[31:0], NOEN, NWEN,
NBL[3:0], SDCLKE[1:0], SDN
SDNWE, NL
NRAS, NCAS, NADV
NWAIT/NIORD, NREG, CD
INTR
ART ACCEL/ CACHE
1MB Flash
1MB Flash
SRAM 112 KB
SRAM 16 KB
RNG
FIFO
Camera interface
HSYNC, VSYNC
PUIXCLK, D[13:0]
USB
DP

1.4 Bus matrix

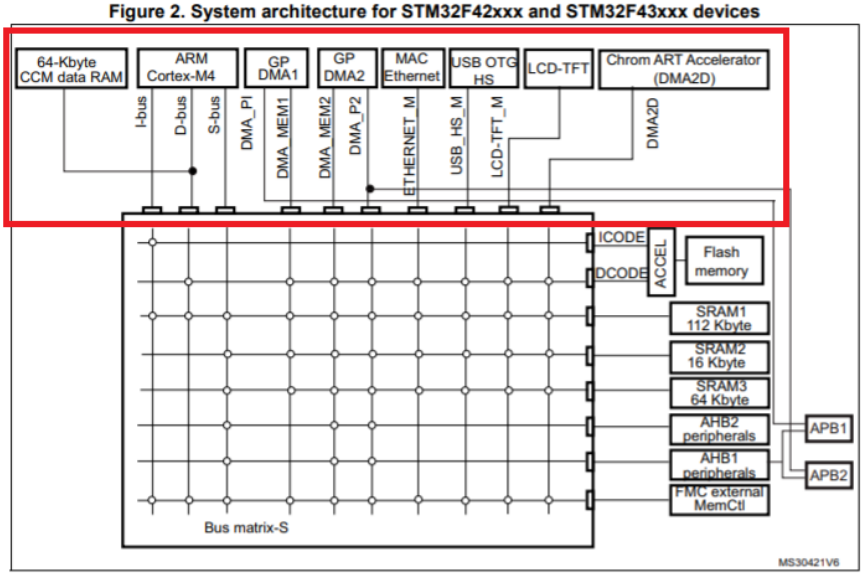
- RM 데이터시트에는 바로 밑 페이지에 위 내용을 matrix 형식으로 정리한 diagram을 제공하고 있다. 아래 matrix로 DMA를 좀 더 쉽게 이해해 보고자 한다

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



A. Master

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices

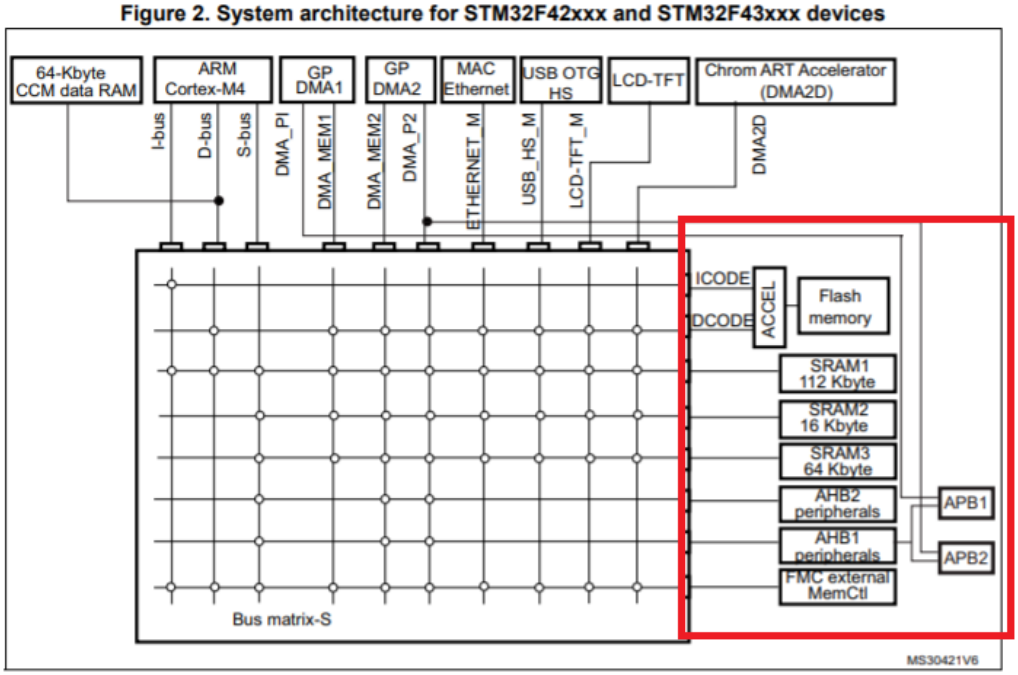


Q. ARM과 DMA를 제외한 나머지들이 master 역할을 할 수 있는 이유는?

- in this MCU, we have advance peripherals that uses an advanced bus matrix interconnect on which the USB, TFT, Ethernet acts as a master which means it can do its own memory transfer based on the software configuration.

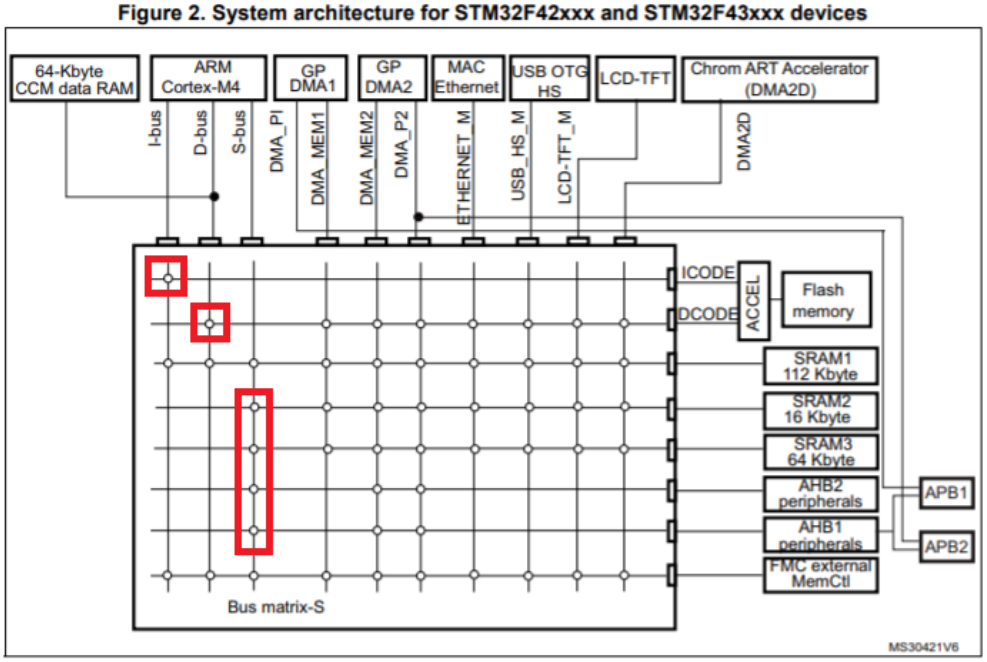
B. Slave

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



C. Matrix 보는 법

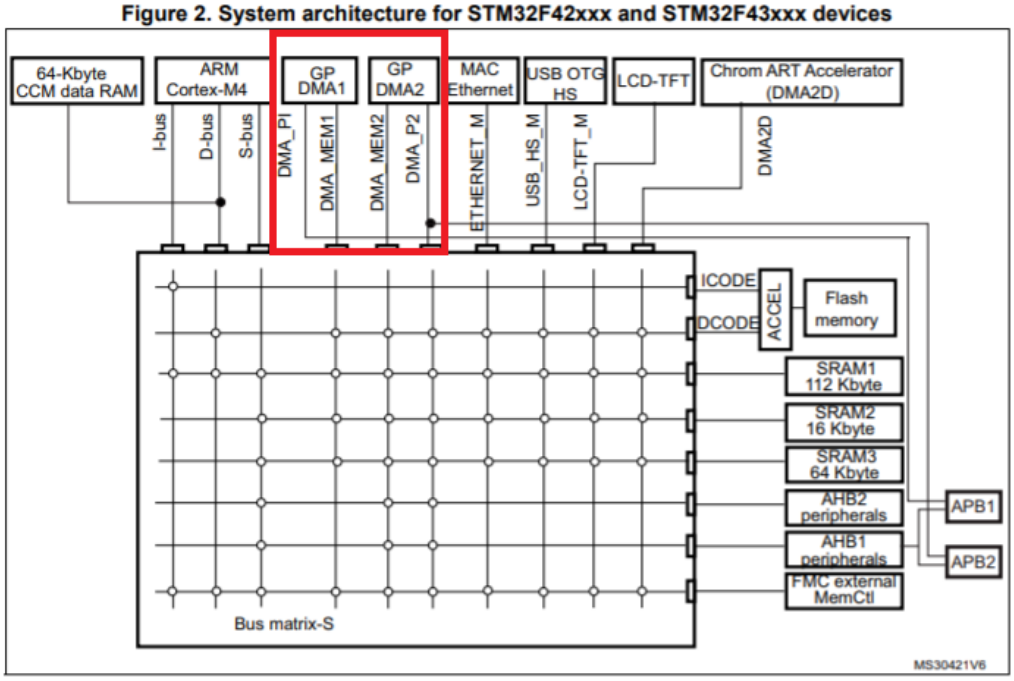
Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



빨간색 박스가 쳐진 부분은 연결된 부분을 뜻한다. 따라서 ARM 코어를 기반으로 보면 Flash 메모리와는 I-bus와 D-bus로 내용을 주고받는 것을 볼 수 있다. 또한 SRAM2, SRAM3, rr 다른 peripheral bus에는 S-bus만 접근 가능한 것을 확인할 수 있다

D. DMA Matrix

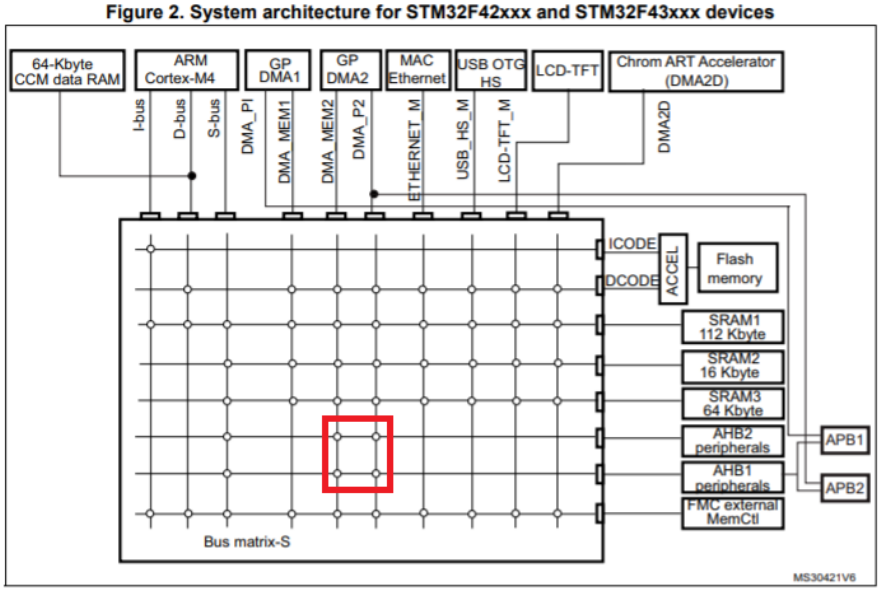
Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



stm32f429에는 총 2개의 DMA 컨트롤러가 존재하고 있다. 각 컨트롤러에는 DMA_P1, DMA_MEM1 2가지 줄기가 뻗어져 있다. 선을 잘 따라가보면 Px 라인에는 APB1, 2와 연결되어 있다. 추측해보면 peripheral과 memory 사이의 처리는 dma 컨트롤러를 통해서 처리될 수 있는 것으로 보인다

또한 DMA2는 AHB 사이의 처리도 담당해줄 수 있는 것으로 보인다. 다만 DMA1은 P2M(Peripheral to Memory)만 가능한 것을 볼 수 있다

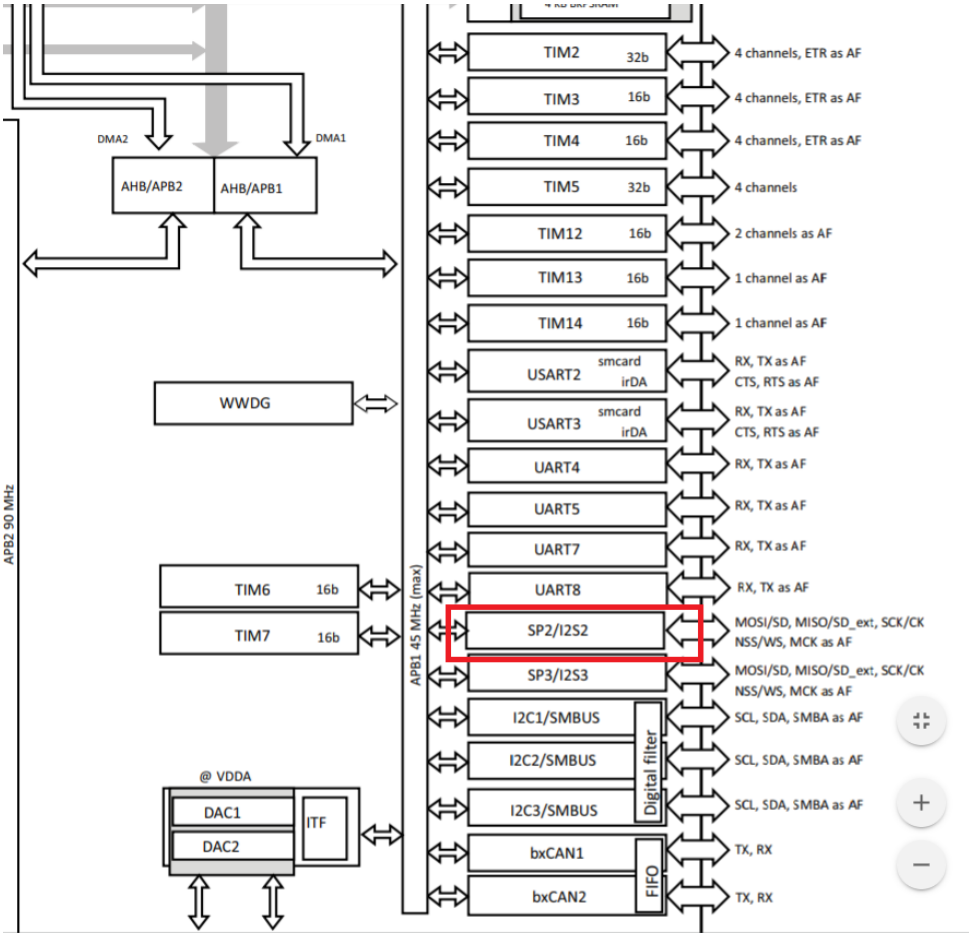
Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



E. 상황 예제

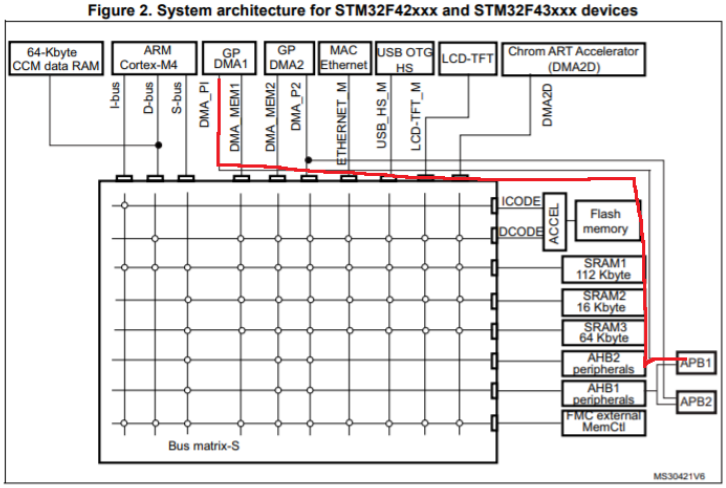
예를 들어 SPI2의 데이터를 SRAM1으로 쓰고 싶을 때 어떻게 DMA Path는 구성될 수 있을까?

먼저 SPI2는 APB1 버스에 연결된 것을 확인할 수 있다



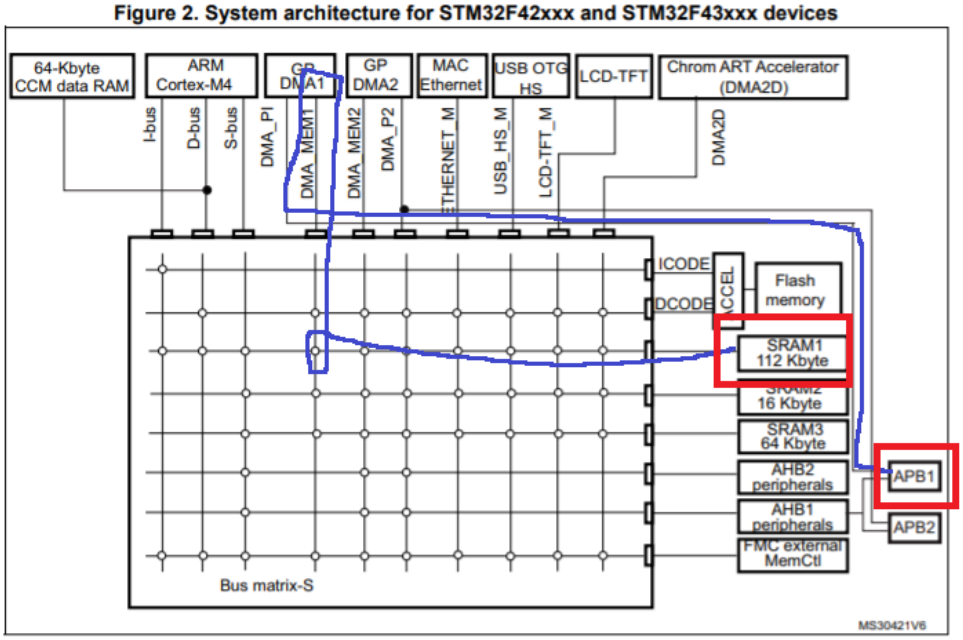
Bus matrix로 봤을 때 APB1은 DMA1_P1과 연결되어 있는 것을 볼 수 있다

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



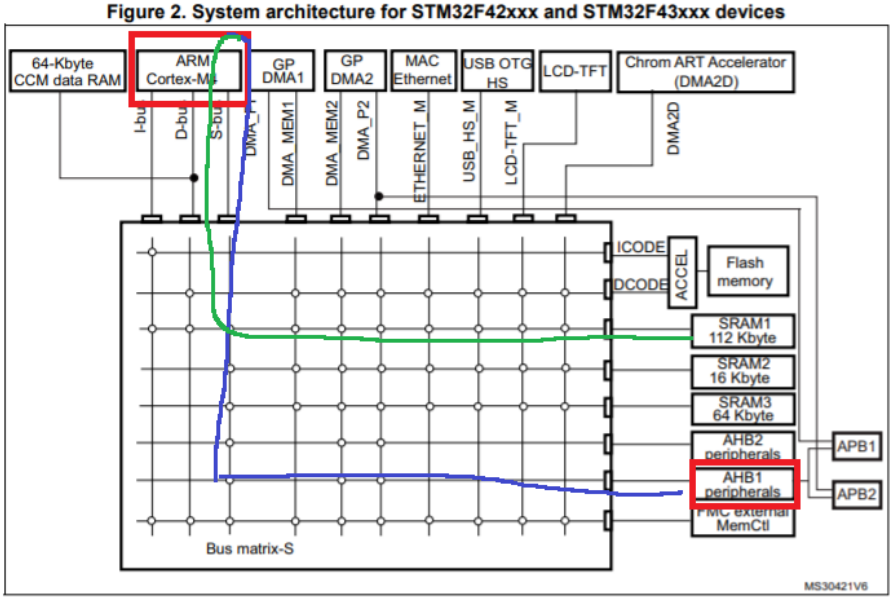
그러면 다음과 같은 path로 DMA1을 통해서 SRAM1 데이터 저장이 가능하다. 다음 path를 확인하면 ARM 코어를 전혀 거치지 않는다는 점을 확인할 수 있다

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



F. ARM 코어를 통한 Path (SPI2, AHB1 peripheral)

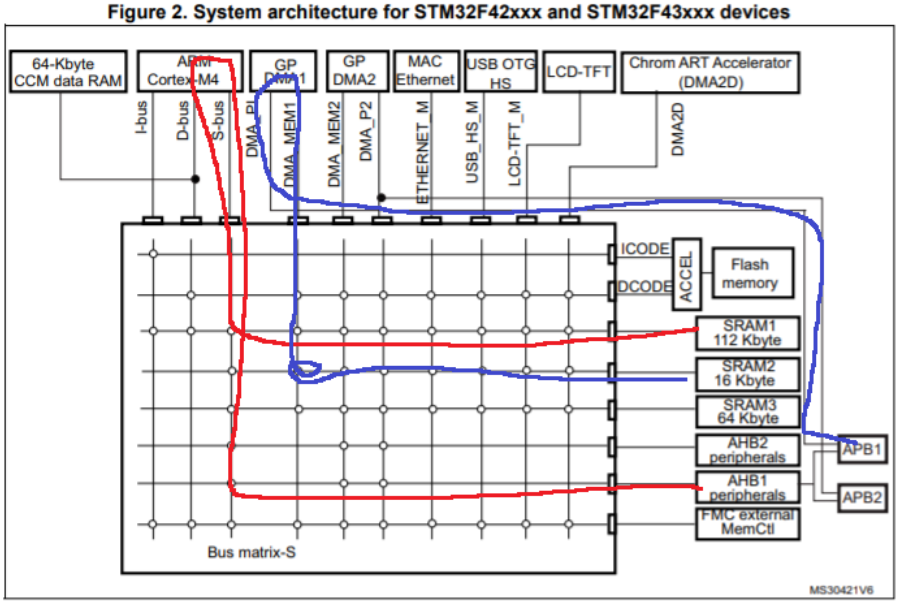
Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



G. DMA와 ARM 코어를 이용한 동시 데이터 전달

위의 내용을 종합해보면 DMA는 ARM과 별개로 데이터 처리를 할 수 있도록 되어 있다. 예를 들어, 1) UART 데이터를 SRAM1에 저장 2) ADC 데이터를 SRAM2에 저장이 아래 Path와 같이 동시다발적으로 동작이 가능하다

Figure 2. System architecture for STM32F42xxx and STM32F43xxx devices



H. 그런데 만일 RAM 메모리가 SRAM1 하나만 존재한다면, 동시적으로 SRAM1에 저장하는 길을 동시에 이용하게 된다. 순간적인 동시 이용은 불가능하기 때문에 priority가 높은 master가 먼저 bus를 선점하게 된다

이렇게 되면 데이터를 잃을 수도 있지만, DMA에는 따로 FIFO가 존재하기 때문에 손실없이 bus의 기회가 올때까지 기다리게 된다

1.5 Interrupt와 DMA 비교

- 두 프로그램을 구성해서 비교해보려고 한다. 하나는 ARM 코어가 지속적으로 while 무한루프를 수행하고 있는데, 인터럽트로 코어가 다른 일을 처리하게끔 하는 것이다. 나머지는 DMA로 코어가 일을 멈추는 것없이 DMA 컨트롤러에 일을 맡기는 것이다
- User Application은 SRAM1에 지속적으로 데이터를 쓰는 작업을 하는 것이다

```
/* USER CODE BEGIN 0 */
uint8_t src_data[50];
#define OFF_SET  0X500
#define DEST_ADDRESS  (volatile uint8_t*)(SRAM1_BASE + OFF_SET)

for( uint32_t i = 0 ; i < 50 ; i++)
{
        src_data[i] = 0xAA;
}

// receive 250 bytes
HAL_UART_Receive_IT(&huart1,(uint8_t*)SRAM2_BASE, 250);
 /* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
        HAL_GPIO_WritePin(GPIOG,GPIO_PIN_13,GPIO_PIN_SET);

        for( i = 0 ; i < 50 ; i++)
        {
                *(DEST_ADDRESS+i) = src_data[i];

        }
        HAL_GPIO_WritePin(GPIOG,GPIO_PIN_13,GPIO_PIN_RESET);
       }
}
```

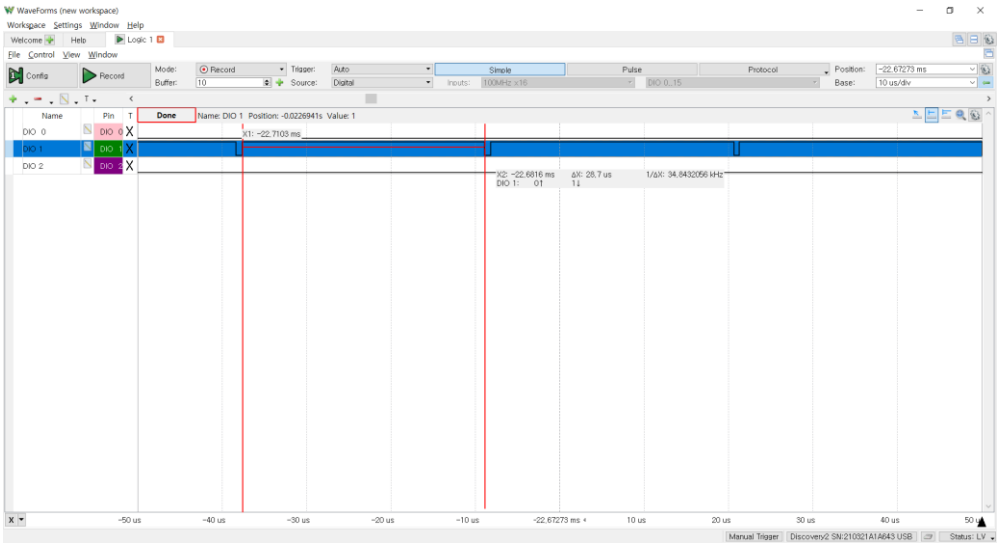
A. Logic 분석기로 총 3가지 신호를 파악하려고 한다. While 루프에서 GPIOG_13에 대한 set, unset을 보려고 한다
B. 그리고 UART 인터럽트를 1바이트 받을 때마다, 250바이트를 모두 받을 때까지의 신호를 측정하려고 한다

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  /* Prevent unused argument(s) compilation warning */
  // UNUSED(huart);
HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_RESET);

HAL_UART_Receive_IT(&huart1,(uint8_t*)SRAM2_BASE, 250);
  /* NOTE: This function Should not be modified, when the callback is
needed,
           the HAL_UART_TxCpltCallback could be implemented in the user file
   */
}
```

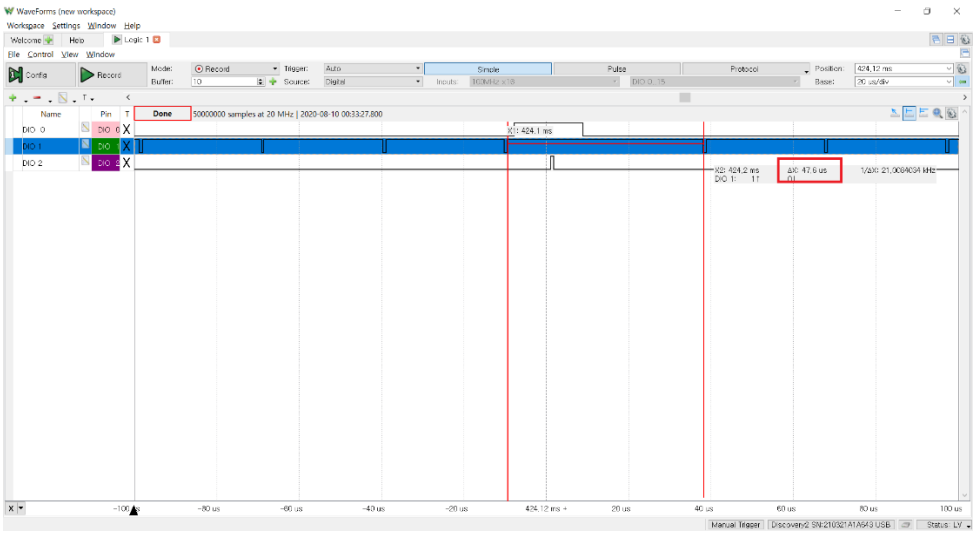
```
void USART1_IRQHandler(void)
{
  /* USER CODE BEGIN USART1_IRQn 0 */
       HAL_GPIO_WritePin(GPIOG,GPIO_PIN_12,GPIO_PIN_SET);
  /* USER CODE END USART1_IRQn 0 */
  HAL_UART_IRQHandler(&huart1);
  /* USER CODE BEGIN USART1_IRQn 1 */
       HAL_GPIO_WritePin(GPIOG,GPIO_PIN_12,GPIO_PIN_SET);
  /* USER CODE END USART1_IRQn 1 */
}
```

C. 코드를 다운로드 한 후 ARM 코어가 SRAM2에 Write를 하는 시간을 측정해보려고 한다. Sample Rate는 측정하고자 하는 주파수의 5배로 지정한다. 현재 16MHz의 클럭으로 동작하고 있기 때문에, 여러 실험을 통해 약 5MHz 이상의 Sample Rate에서 정확한 측정을 할 수 있었다. 그렇지 않으면 신호를 놓치게 되었다

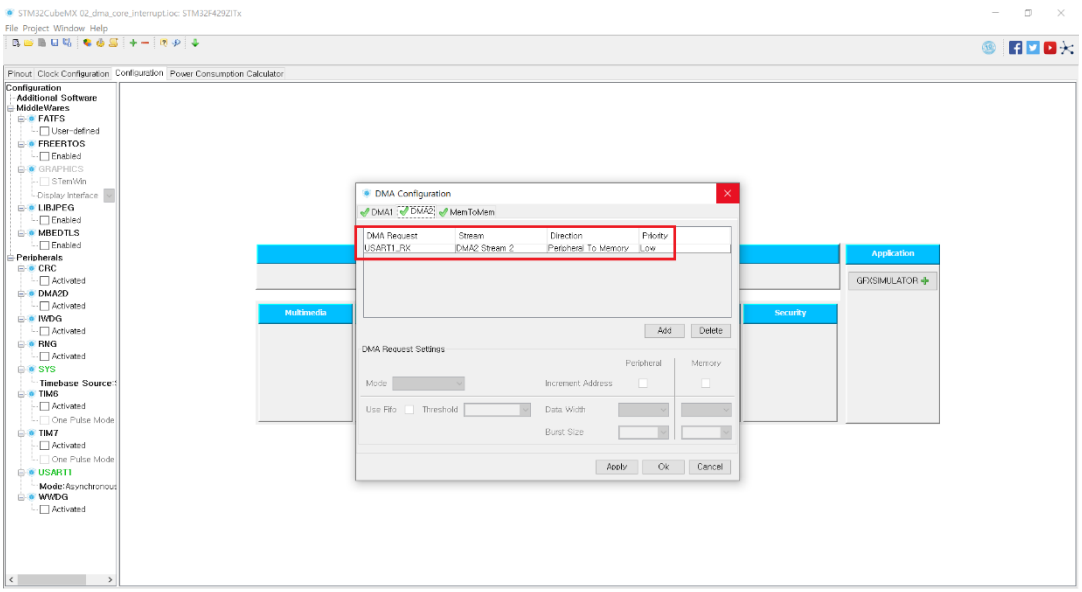
따라서 아래 측정 결과를 보면 인터럽트가 없을 때는 약 29us에 SRAM1에 데이터를 모두 쓰게 된다



D. 그리고 인터럽트를 발생시켰을 때는 47us라는 긴 시간이 걸린 것을 볼 수 있다



E. 이제는 DMA 코드를 넣어서 실행해보려고 한다. 먼저 CubeMX로 설정을 하도록 한다. 자세한 내용은 이후에 다루려고 한다

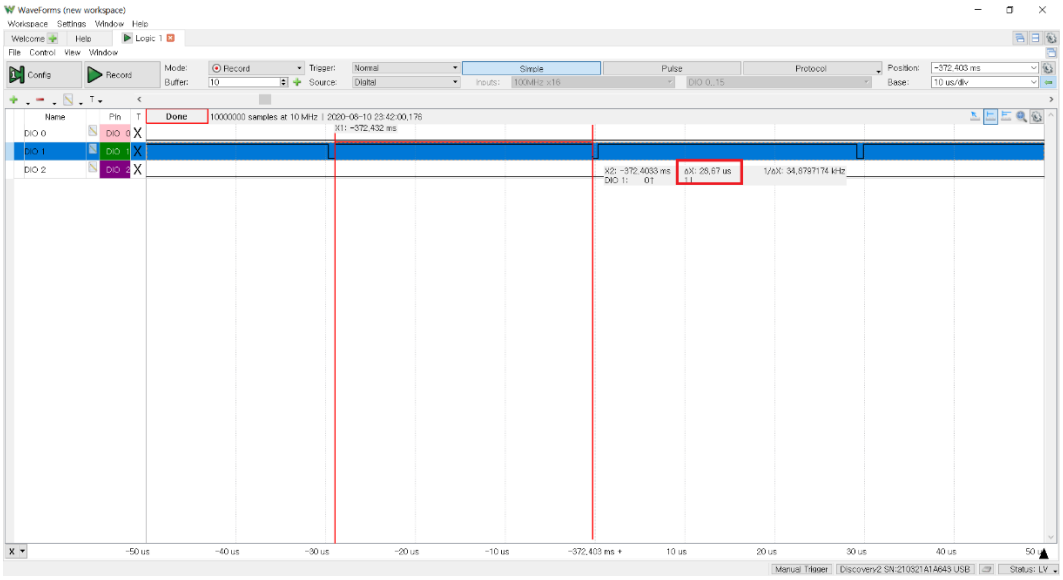


F. 그리고 소스코드를 인터럽트가 아닌 DMA로 받도록 한다

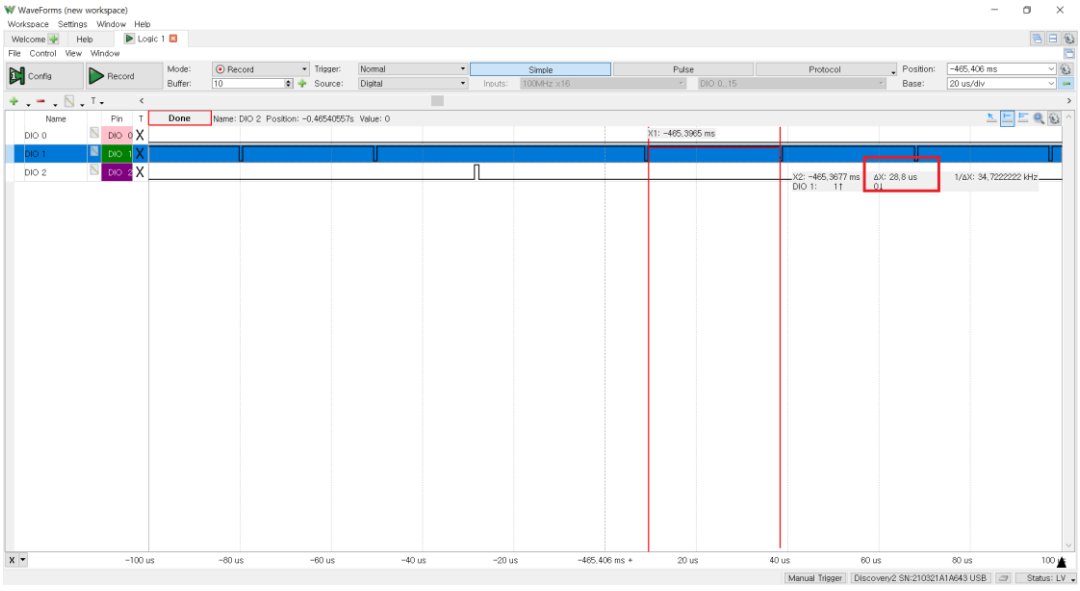
```
  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
  {
    /* Prevent unused argument(s) compilation warning */
    // UNUSED(huart);
        HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_RESET);

        HAL_UART_Receive_DMA(&huart1,(uint8_t*)SRAM2_BASE, 250);
    /* NOTE: This function Should not be modified, when the callback
  is needed,
            the HAL_UART_TxCpltCallback could be implemented in the
  user file
     */
    }

// HAL_UART_Receive_IT(&huart1,(uint8_t*)SRAM2_BASE, 250);
 HAL_UART_Receive_DMA(&huart1,(uint8_t*)SRAM2_BASE, 250);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
        HAL_GPIO_WritePin(GPIOG,GPIO_PIN_13,GPIO_PIN_SET);

        for( i = 0 ; i < 50 ; i++)
        {
                *(DEST_ADDRESS+i) = src_data[i];
```

```
    }
    HAL_GPIO_WritePin(GPIOG,GPIO_PIN_13,GPIO_PIN_RESET);
}
```

프로그램을 다운로드 한 후 측정을 하면 UART 데이터를 넣지 않았을 때는 28us로 기존 인터럽트와 비슷했다



G. 지속적으로 250바이트가 들어오지만 while문 안에는 28us를 유지하는 것을 볼 수 있다



그리고 인터럽트 때만 ARM 코어가 DMA handler를 처리하는 것을 볼 수 있다