

마이크로프로세서응용실험 LAB11 결과보고서

Direct memory access (DMA) AND ADC/DAC

20181536 엄석훈

1. 목적

- DMA의 동작원리와 구동방법을 이해한다. DMA의 구동을 위해 필요한 요소들과 연관 입출력소자의 DMA지원을 위한 설정 등을 Program 9.1와 Program 10.1에서의 경험을 기반으로 파악한다.
- ADC를 구동하는 방법을 이해한다. Polling과 DMA를 이용하여 ADC를 구동하는 방법을 확인한다.

2. 이론

1) DMA Introduction

마이크로프로세서에서 입출력 장치를 사용하는 경우는 매우 많다. 이때 CPU가 직접 I/O 장치와 데이터를 주고받으며 동작을 하게 되면 CPU가 I/O장치의 속도에 맞춰 동작해야 하기 때문에 CPU의 자원이 낭비될 수 있다. 따라서 CPU를 거치지 않고 I/O장치와 메모리 사이의 데이터 이동을 위해 고안한 방법이 DMA이고 이를 위한 별도의 소자 또는 제어방식을 DMAC라고 한다. DMAC가 address bus와 data bus를 사용해 데이터를 전송하는 동작을 수행한다. 이때 DMAC는 명령어에 따라 동작하는 것이 아니라 기계적인 동작을 통해 데이터를 전송하는데 따라서 CPU에 비해서 처리시간이 빠르다. 아래 [그림 1]과 같이 DMA의 구조를 볼 수 있다.

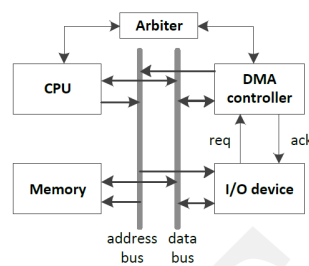


그림 1 DMA 구조

또한 DMAC는 CPU의 동작과는 독립적으로 동작하며 하나의 DMAC가 여러 channel을 통해서 I/O장치에 대해서 DMA를 수행할 수 있다. 우리가 사용하는 보드에서는 DMA1의 경우 7개의 channel을 통해 DMA 요청을 받을 수 있다.

2) DMA Transactions

모든 DMA 전송은 3가지 동작으로 이루어져 있다. 먼저 메모리 또는 주변장치에서 데이터를 가져온다. 이때 주변장치 또는 메모리 주소의 시작 주소는 DMA_CPAR 또는 DMA_CMAR 레지스터를 통해서 설정할 수 있다. 다음으로 가져온 데이터를 주변장치 또는 메모리에 internal current를 통해서 전달해준다. 마지막으로 앞으로 수행되어야 할 전송의 숫자가 저장된 DMA_CNDTR 레지스터의 값을 감소시켜준다.

다음으로 multi-layer 구조를 통해서 master가 서로 다른 slave에게 데이터를 전송해준다. 이를 통해서 데이터를 병렬적으로 전송할 수 있도록 하여 DMA를 사용할 때의 효율성을 높여준다. 이때 DMA의 데이터 전송에는 bus stealing과 burst mode가 있다. 이에 대한 데이터 전송의 구조는 아래 [그림 2]와 같다. Bus stealing은 데이터 전송을 위해 하나의 bus cycle만 사용해서 데이터를 전송하는 것을 말한다. 따라서 한 사이클만 DMA가 데이터를 전달하고 나머지는 다시 CPU가 bus를 사용한다. 다음으로 burst mode는 여러 사이클 동안 DMA가 bus를 이용해서 데이터를 전달하는 것을 의미한다.

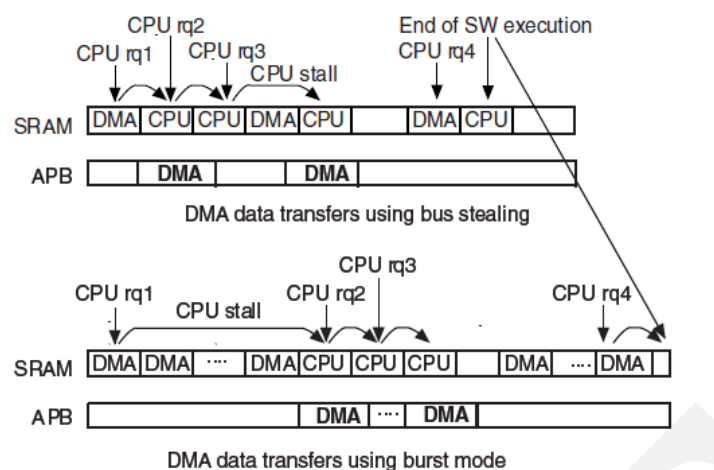


그림 2 DMA transfer

3) Arbiter

Arbiter는 DMA의 여러 채널의 요청을 우선순위에 따라 주변장치와 메모리의 접근 순서를 관리한다. DMA의 우선순위는 소프트웨어에 의해서는 DMA_CCR 레지스터를 통해 설정할 수 있으며 4가지 우선순위를 가지고 있다. 또한 하드웨어에 의해서는 두가지 우선순위가 같을 때 채널의 숫자가 높은 채널이 더 높은 우선순위를 가지도록 설정되어 있다. 이때 이러한 우선순위를 통해서 높은 속도와 높은 대역폭을 요구하는 주변장치를 높은 우선순위로 설정해야 DMA를 효율적으로 사용할 수 있다.

4) DMA Request Mapping

DMA는 다양한 주변소자로부터 request를 요청받을 수 있다. TIM, ADC1, SPI1, SPI/I2S2, I2C, USART와 연결될 수 있다. 아래 [그림 3]과 같이 DMA1이 채널별로 받을 수 있는 request이다. 그리고 [그림 4]는 DMA1의 request mapping에 대한 그림이다.

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
ADC1	ADC1	-	-	-	-	-	-
SPI/I ² S	-	SPI1_RX	SPI1_TX	SPI2/I2S2_RX	SPI2/I2S2_TX	-	-
USART	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C	-	-	-	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1	-	TIM1_CH1	-	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	-
TIM2	TIM2_CH3	TIM2_UP	-	-	TIM2_CH1	-	TIM2_CH2 TIM2_CH4
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	-	TIM3_CH1 TIM3_TRIG	-
TIM4	TIM4_CH1	-	-	TIM4_CH2	TIM4_CH3	-	TIM4_UP

그림 3 DMA1 채널

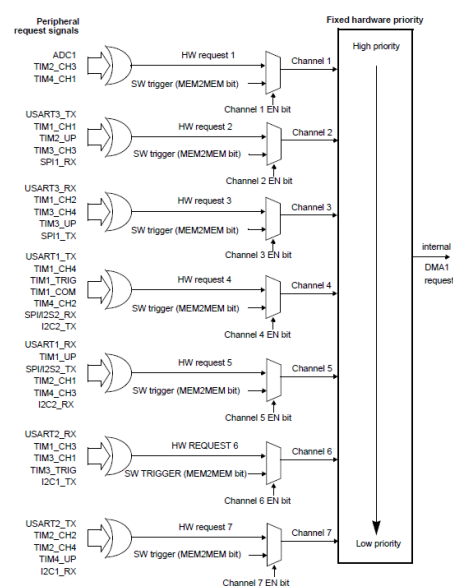


그림 4 DMA1의 request mapping

5) DMA Register Map

아래 [그림 5]는 DMA1의 register map의 일부 그림이다. 채널 1 ~ 4의 레지스터만 나타나 있지만 채널 5-7에 대해서 동일한 형태로 레지스터가 주어져 있다.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0x000	DMA_ISR	Reserved				TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5	TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1	0			
	Reset value					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x004	DMA_IFCR	Reserved				CTEIF7	CHTIF7	CTCIF7	CGIF7	CTEIF6	CHTIF6	CTCIF6	CGIF6	CTEIF5	CHTIF5	CTCIF5	CGIF5	CTEIF4	CHTIF4	CTCIF4	CGIF4	CTEIF3	CHTIF3	CTCIF3	CGIF3	CTEIF2	CHTIF2	CTCIF2	CGIF2	CTEIF1	CHTIF1	CTCIF1	CGIF1	0			
	Reset value					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x008	DMA_CCR1	Reserved																	MEM2MEM	PL [1:0]	M SIZE [1:0]		PSIZE [1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN	0				
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x00C	DMA_CNDTR1	Reserved																	NDT[15:0]																		
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x010	DMA_CPAR1	PA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x014	DMA_CMAR1	MA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x018	Reserved																																				
0x01C	DMA_CCR2	Reserved																	MEM2MEM	PL [1:0]	M SIZE [1:0]		PSIZE [1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN	0				
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x020	DMA_CNDTR2	Reserved																	NDT[15:0]																		
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x024	DMA_CPAR2	PA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x028	DMA_CMAR2	MA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x02C	Reserved																																				
0x030	DMA_CCR3	Reserved																	MEM2MEM	PL [1:0]	M SIZE [1:0]		PSIZE [1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN	0				
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x034	DMA_CNDTR3	Reserved																	NDT[15:0]																		
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x038	DMA_CPAR3	PA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x03C	DMA_CMAR3	MA[31:0]																																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
0x040	Reserved																																				

그림 5 DMA Registr Map

6) ADC Introduction

ADC는 아날로그 인풋을 디지털 신호로 바꿔주는 회로를 말한다. 이의 원리를 살펴보기 위해 [그림 6]을 보면서 간단히 설명하면 0 ~ 15V의 아날로그 전압을 JK flip-flop을 이용해 카운터의 형태로 저장되어 낮은 카운터에서 전압값에 따라 증가하면서 아날로그 전압이 디지털신호로 바뀌어 저장되고 출력된다. 이 회로의 경우 간단하지만 동작이 느리다는 단점이 있다.

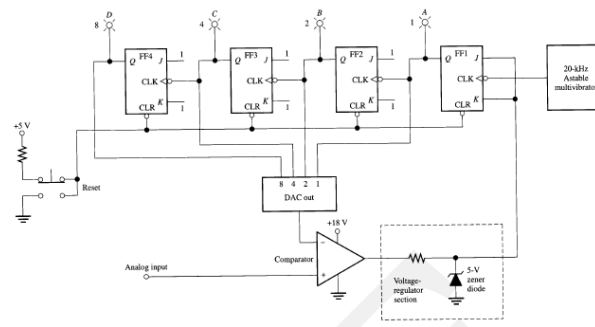


그림 6 간단한 4-bit ADC 회로

이러한 단점을 해결한 회로가 아래 [그림 7]과 같이 SAR 회로를 이용한 ADC 회로이다. 간단히 원리를 설명하면 매 클럭마다 SAR의 MSB인 D7부터 차례대로 1로 set하면서 이때의 DAC 출력이 아날로그 인풋보다 크다면 다시 0으로 바꾸고 아니라면 1로 계속 세팅하고 있어 매 클럭마다 아날로그 인풋 값에 가깝게 다다가는 방식의 회로이다. 아래 [그림 8]이 이러한 동작에 대한 예시이다.

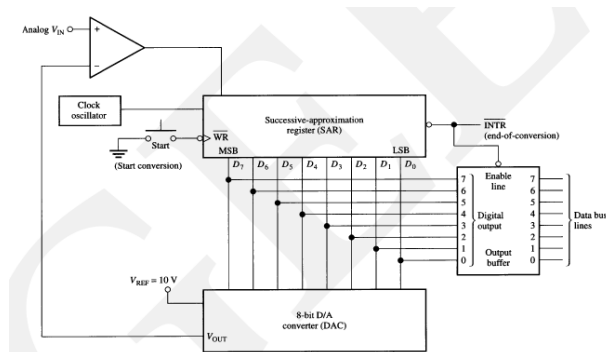


그림 7 SAR를 사용한 ADC 회로

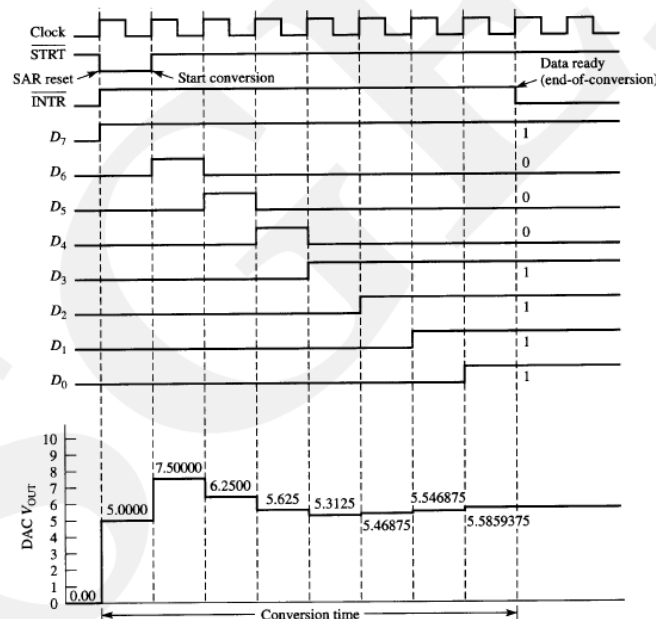


그림 8 SAR 동작 예시

다음으로 우리가 사용하는 보드에서의 ADC를 위한 회로는 아래 [그림 9]와 같이 구성되어 있다. 주요한 기능으로는 12-bit 해상도를 가지고 있고 end of conversion에서 인터럽트가 발생할 수 있으며, single conversion mode와 continuous conversion mode, scan mode가 존재하며 self-calibration이 가능하고 채널마다 sampling rate를 설정할 수 있고, external trigger option이 존재하는 등 다양한 기능을 포함하고 있다.

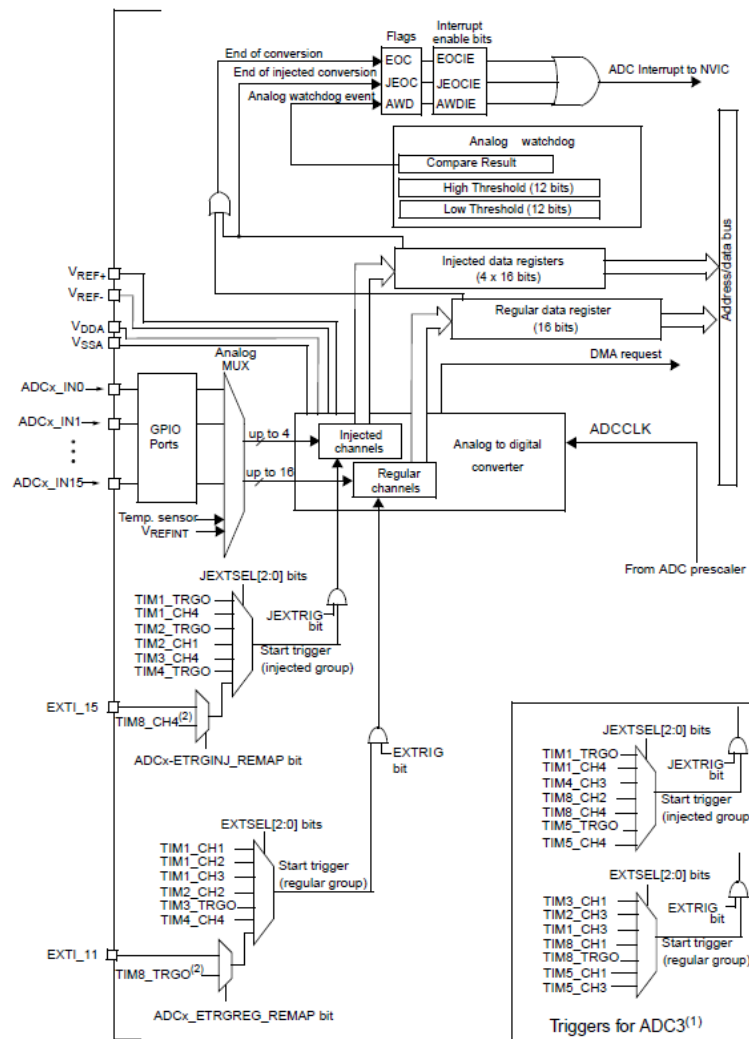


그림 9 ADC block diagram

7) ADC Functional Description

ADC는 필요에 따라 전원을 켜거나 끌 수 있으며 ADC_CR2 레지스터의 ADON비트를 통해서 제어할 수 있다. 또한 16개의 채널이 존재하며 최대 16개의 채널로 구성가능한 regular group과 최대 4개의 채널로 구성가능한 injected group이 존재하며 injected group의 우선순위가 더 높다. 그리고 regular group의 conversion 결과는 ADC_DR 레지스터에서 확인 가능하며 injected group의

conversion 결과는 ADC_DRJ1 레지스터에서 확인 가능하다. 또한 한 번만 conversion을 수행하는 single conversion과 계속해서 conversion을 수행하는 continuous conversion이 모두 가능하다. 아래 [그림 10]은 ADC 동작의 timing diagram의 예시이다. 이를 통해 확인 가능한 것은 ADC가 시작되기 전 안정화를 위해 t_{STAB} 의 시간이 필요하며 conversion시작 후 14사이클 이후에 EOC flag가 세팅된다.

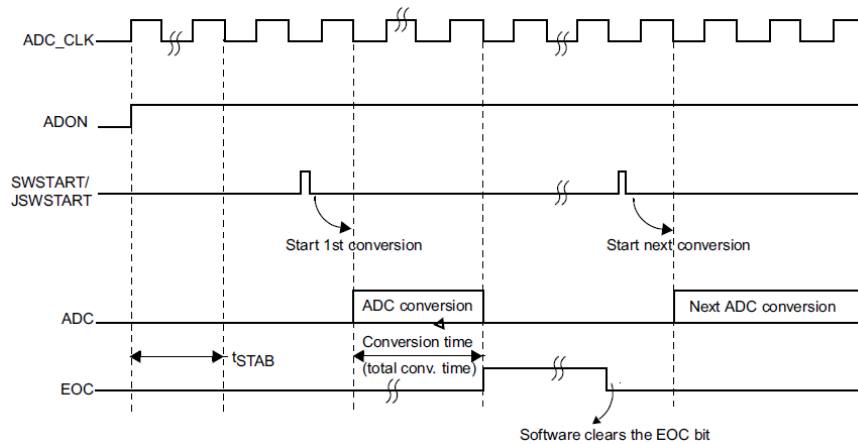


그림 10 ADC timing diagram

3. 실험 과정

1) 실험 1

STEP 1: Program 11.1은 Program 9.1의 경우와 마찬가지로 메모리에 저장된 font를 이용하여 dot matrix display에 특정 16진수 수를 표시하는 프로그램이다. Program 9.1과 달리 데이터의 출력에 DMA를 이용한다. 이 프로그램을 이용하여 프로젝트를 생성한다.

```

1  #include <stm32f10x.h>
2  #define DOT_MATRIX_COL 0x4001100C
3
4  u8 font8x8[16][8] = {
5      {0x3c, 0x42, 0x46, 0x4a, 0x52, 0x62, 0x3c, 0x00}, //0
6      {0x10, 0x30, 0x50, 0x10, 0x10, 0x10, 0x7c, 0x00}, //1
7      {0x3c, 0x42, 0x02, 0x0c, 0x30, 0x42, 0x7e, 0x00}, //2
8      {0x3c, 0x42, 0x02, 0x1c, 0x02, 0x42, 0x3c, 0x00}, //3
9      {0x08, 0x18, 0x28, 0x48, 0xf3, 0x08, 0x1c, 0x00}, //4
10     {0x7e, 0x40, 0x7c, 0x02, 0x02, 0x42, 0x3c, 0x00}, //5
11     {0x1c, 0x20, 0x40, 0x7c, 0x42, 0x42, 0x3c, 0x00}, //6
12     {0x7e, 0x42, 0x04, 0x08, 0x10, 0x10, 0x10, 0x00}, //7
13     {0x3c, 0x42, 0x42, 0x3c, 0x42, 0x42, 0x3c, 0x00}, //8
14     {0x3c, 0x42, 0x42, 0x3e, 0x02, 0x04, 0x38, 0x00}, //9
15     {0x18, 0x24, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x00}, //A
16     {0x7c, 0x22, 0x22, 0x3c, 0x22, 0x22, 0x7c, 0x00}, //B
17     {0x1c, 0x22, 0x40, 0x40, 0x40, 0x22, 0x1c, 0x00}, //C
18     {0x78, 0x24, 0x22, 0x22, 0x22, 0x24, 0x78, 0x00}, //D
19     {0x7e, 0x22, 0x28, 0x38, 0x28, 0x22, 0x7e, 0x00}, //E
20     {0x7e, 0x22, 0x28, 0x38, 0x28, 0x20, 0x70, 0x00}, //F
21 };
22
23 u8 data = 0x09;
24 u32 row = 1;
25
26 int main(void) {
27     RCC->APB2ENR |= 0x0000001D;
28     GPIOC->CRL = 0x33333333;
29     GPIOB->CRH = 0x33333333;
30
31     RCC->APB1ENR |= 0x00000001;
32     TIM2->CR1 = 0x04;
33     TIM2->CR2 = 0x00;
34     TIM2->PSC = 0x07FF;
35     TIM2->ARR = 0x000F;
36     TIM2->DIER = 0x0101;
37     NVIC->ISER[0] = (1 << 28);
38
39     RCC->AHBENR |= 0x00000001;
40     DMA1_Channel2->CCR = 0x000000B0;
41     DMA1_Channel2->CNDTR = 8;
42     DMA1_Channel2->CPAR = DOT_MATRIX_COL;
43     DMA1_Channel2->CMAR = (u32) font8x8[data];
44
45     DMA1_Channel2->CCR |= 0x00000001;
46
47     TIM2->CR1 |= 0x0001;
48
49     while(1) {}
50 } // end main
51
52 void TIM2_IRQHandler(void) {
53     if((TIM2->SR & 0x0001) != 0) {
54         GPIOB->ODR = (~row) << 8;
55         row = row << 1;
56         if(row == 0x100) {row = 1;}
57         TIM2->SR &= ~(1 << 0); //clear UIF
58     }
59 }
60

```

그림 11 직접 작성한 lab11_1.c 코드

STEP 2: DMA의 데이터 port access를 위해 195페이지의 Step 1의 dot matrix display와 GPIO port와의 연결을 다음과 같이 변경하여 연결한다.

row no. (pin no.)	1(9)	2(14)	3(8)	4(12)	5(1)	6(7)	7(2)	8(5)
GPIO port	PB8	PB9	PB10	PB11	PB12	PB13	PB14	PB15
col no. (pin no.)	1(13)	2(3)	3(4)	4(10)	5(6)	6(11)	7(15)	8(16)
GPIO port	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0

그림 12 교재에서 주어진 연결

교재에서 주어진 대로 위 [그림 12]를 바탕으로 보드와 dot matrix를 아래 [그림 13]과 같이 연결하였다.

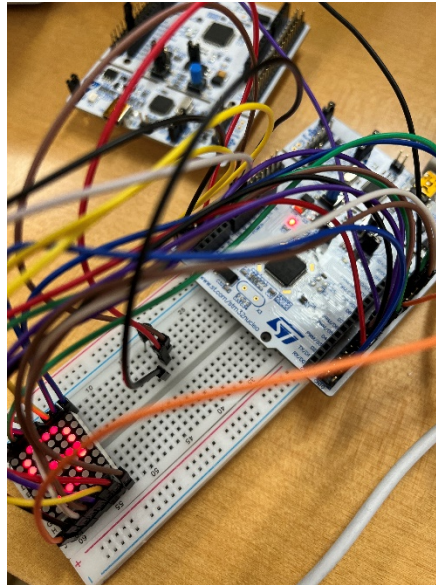


그림 13 실험 1을 위한 보드 연결

STEP 3: Lines 32 - 37은 TIM2의 update event를 DMA request에 이용하기 위한 과정이다. Program 9.2의 update event를 처리하기 위한 설정부분을 DMA request를 위한 설정과 비교해 보자. 달라진 부분은 무엇인가? 그 의미를 reference manual(STMicroelectronics (2011))을 통해 확인한다.

```

10  TIM2->CR1 = 0x04;
11  TIM2->CR2 = 0x00;
12  TIM2->PSC = 0x5FFF;
13  TIM2->ARR = 0x1FFF;
14  TIM2->DIER = 0x0041;
15
16  TIM2->SMCR = 0x8074;
17
18  NVIC->ISER[0] = (1 << 28);

```

그림 14 프로그램 9.2 코드 일부

위의 [그림 14]는 프로그램 9.2에서 update event를 처리하기 위해 설정한 부분이다. 이를 통해서 따 [그림 14]의 프로그램 9.2의 코드와 [그림 11]의 line 32 - 37과 비교해보면 다음과 같이 다른 점이 있다. 먼저 TIM2->PSC와 TIM2->ARR에 저장하는 값이 다른데 이는 update event가 발생하는 주기가 다르게 하기 위해서 다른 값으로 설정한 것이다. 중요하게 다른 부분은 DMA/interrupt enable register인 TIM2->DIER에 프로그램 9.2에서는 0x0041을 저장한 반면 프로그램 11.1에서는 0x0101을 저장하였다. 0x0041로 설정하면 0번 bit인 UIE bit와 6번 bit인 TIE bit를 1로 set하여 update interrupt와 trigger interrupt가 발생할 수 있도록 설정한 데 비해서 0x0101로 설정하면 0

번 bit인 UIE bit와 8번 bit인 UDE bit를 1로 set하여 update interrupt와 update DMA request가 발생할 수 있도록 설정해주었다. 즉 프로그램 11.1에서는 TIM2->DIER 레지스터의 8번 bit를 set하여 DMA request가 발생할 수 있도록 설정하였다.

STEP 4: Lines 39 - 43은 DMA1에 속한 channel2를 설정하는 과정이다. 각 단계를 reference manual(STMicroelectronics (2011))을 통해 확인한다. 이 초기화 과정을 다음과 같이 단계적으로 파악한다.

● Line 39는 무엇을 하기 위함인가?

먼저 line 39의 `RCC->AHBENR |= 0x00000001;` 을 통해서 `RCC->AHBENR` 레지스터의 0번 bit인 `DMA1EN` bit를 1로 set하고 기존의 다른 값은 건드리지 않도록 or 연산을 통해 동작을 수행하였다. 이를 통해 `DMA1` clock이 `AHB` bus에 연결되도록 enable되었다.

● 이 과정에서 DMA의 여러 channel 중에서 Channel2를 초기화하는 이유는 무엇인가? 표 11.3를 참조하여 그 이유를 설명한다.

아래 [그림 15]는 교재의 표 11.3으로 각 채널에 연결된 다양한 주변장치의 DMA request 목록이다. 그림을 보면 `TIM2_UP`은 channel2로 되어있는 것을 볼 수 있다. 따라서 `TIM2`의 update event의 DMA request는 channel 2번으로 들어오기 때문에 실험 11.1에서도 channel 2를 초기화 하였다.

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
ADC1	ADC1	-	-	-	-	-	-
SPI	-	SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX	-	-
USART	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I ² C	-	-	-	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1	-	TIM1_CH1	-	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3 TIM1_CH2 TIM1_CH1	-
TIM2	TIM2_CH3	TIM2_UP	-	-	TIM2_CH1	-	TIM2_CH2 TIM2_CH4
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	-	TIM3_CH1 TIM3_TRIG	-
TIM4	TIM4_CH1	-	-	TIM4_CH2	TIM4_CH3	-	TIM4_UP
TIM6/DAC_Channel1	-	-	TIM6_UP/DAC_Channel1	-	-	-	-
TIM7/DAC_Channel2	-	-	-	TIM7_UP/DAC_Channel2	-	-	-
TIM15	-	-	-	-	TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM	-	-
TIM16	-	-	-	-	-	TIM16_CH1 TIM16_UP	-
TIM17	-	-	-	-	-	-	TIM17_CH1 TIM17_UP

그림 15 교재 표 11.3

● Line 40을 통해 DMA의 해당 channel이 어떻게 설정되고 있는가? 11.2.3절을 참고하여 DMA를 통해 구현될 동작을 염두에 두고 다음을 중심으로 그 의미와 함께 확인해 보자

- MSIZE

- PSIZE
- MINC
- PINC
- DIR
- CIRC

Line 40의 DMA1_Channel2->CCR = 0x000000B0; 를 통해서 Channel2의 CCR 레지스터에 4번, 5번, 7번 bit를 1로 set하고 나머지는 0으로 설정하였다. 이를 통해 DMA동작에 대해 분석해보면 MSIZE는 MSIZE bit가 00이기 때문에 8-bits이고 PSIZE도 PSIZE bit가 00임으로 8-bits이다. MINC bit는 1임으로 memory increment mode가 enable되어있고 PINC bit는 0임으로 peripheral increment mode는 disable되어있다. DIR bit는 1임으로 memory에서 데이터를 읽어드리는 방향으로 동작하며, CIRC bit도 1임으로 circular mode가 enable되어 있다.

이를 바탕으로 DMA의 동작을 살펴보면 우선 주변장치와 메모리 모두 8bit씩 데이터가 전달되는데 DIR bit가 1임으로 메모리에서 주변장치로 8-bits의 데이터가 이동한다. 그리고 circular mode임으로 반복적으로 동일한 데이터가 이동한다. 또한 PINC는 0임으로 주변장치의 주소는 계속 그대로 유지되며 MINC는 1임으로 메모리의 주소는 DMA가 데이터를 이동시키고 나서 자동으로 증가한다. 즉 8-bits씩 데이터를 메모리의 주소를 증가시키며 주변장치로 이동시키며 이 동작을 반복한다.

● **Line 41에서 초기화되는 레지스터의 역할은 무엇인가? 이 레지스터에 저장되는 값의 의미는 이 응용에서 무엇인가?**

Line 41의 DMA1_Channel2->CNDTR = 8; 을 통해서 DMA가 전달할 데이터의 개수를 설정해주었다. 즉 DMA는 총 8개의 데이터를 전달해주는 동작을 수행한다. 위의 line 40의 설정과 합쳐서 생각하면 반복적으로 8-bits의 데이터 8개를 주소를 증가시키며 메모리에서 주변장치로 이동시키는 동작을 DMA가 수행한다.

● **Line 42에서 초기화되는 레지스터의 역할은 무엇인가? 이 line을 통해 이 레지스터에 저장되는 값의 근거를 표 8.1을 통해 확인한다.**

Line 42의 DMA1_Channel2->CPAR = DOT_MATRIX_COL; 에서 먼저 DOT_MATRIX_COL은 코드에서 #define을 통해 0x4001100C로 되어있었으므로 사실상 코드는 DMA1_Channel2->CPAR =

0x4001100C; 로 생각할 수 있다. 먼저 아래 [그림 16]은 교재 표 8.1의 일부이다. 이때 0x4001100C의 경우는 GPIOC영역의 주소임을 알 수 있다. 그리고 이 주소는 GPIOC의 base address로부터 0x0C만큼 offset이 떨어져 있는데 reference manual을 보면 GPIO의 ODR레지스터가 offset이 0x0C인 것을 [그림 17]을 통해 확인할 수 있다. 즉 0x4001100C는 GPIOC->ODR 레지스터의 주소를 나타낸다. 그리고 DMA1_Channel2->CPAR 레지스터는 DMA의 주변장치의 주소를 저장하는 레지스터이다. 따라서 DMA1_Channel2를 통해 연결하는 주변장치는 GPIOC->ODR 레지스터이다.

Boundary address	Peripheral	Bus
0x4000 3000 – 0x4000 33FF	Independent watchdog (IWDG)	APB1
0x4000 3400 – 0x4000 37FF	Reserved	
0x4000 3800 – 0x4000 3BFF	SPI2/I2S	
0x4000 3C00 – 0x4000 43FF	Reserved	
0x4000 4400 – 0x4000 47FF	USART2	
0x4000 4800 – 0x4000 4BFF	USART3	
0x4000 4C00 – 0x4000 53FF	Reserved	
0x4000 5400 – 0x4000 57FF	I2C1	
0x4000 5800 – 0x4000 5BFF	I2C2	
0x4000 5C00 – 0x4000 5FFF	USB device FS registers	
0x4000 6000 – 0x4000 63FF	Shared USB/CAN SRAM 512 bytes	
0x4000 6400 – 0x4000 67FF	bxCAN	
0x4000 6800 – 0x4000 6BFF	Reserved	
0x4000 6C00 – 0x4000 6FFF	Backup registers (BKP)	
0x4000 7000 – 0x4000 73FF	Power control PWR	
0x4000 7400 – 0x4000 7FFF	Reserved	APB2
0x4001 0000 – 0x4001 03FF	AFIO	
0x4001 0400 – 0x4001 07FF	EXTI	
0x4001 0800 – 0x4001 0BFF	GPIO Port A	
0x4001 0C00 – 0x4001 0FFF	GPIO Port B	
0x4001 1000 – 0x4001 13FF	GPIO Port C	
0x4001 1400 – 0x4001 17FF	GPIO Port D	
0x4001 1800 – 0x4001 1BFF	GPIO Port E	
0x4001 1C00 – 0x4001 23FF	Reserved	
0x4001 2400 – 0x4001 27FF	ADC1	
0x4001 2800 – 0x4001 2BFF	ADC2	
0x4001 2C00 – 0x4001 2FFF	TIM1 timer	
0x4001 3000 – 0x4001 33FF	SPI1	
0x4001 3400 – 0x4001 37FF	Reserved	
0x4001 3800 – 0x4001 3BFF	USART1	
0x4001 3C00 – 0x4001 7FFF	Reserved	AHB
0x4001 8000 – 0x4001 FFFF	Reserved	
0x4002 0000 – 0x4002 03FF	DMA	
0x4002 0400 – 0x4002 0FFF	Reserved	
0x4002 1000 – 0x4002 13FF	Reset and clock control RCC	
0x4002 1400 – 0x4002 1FFF	Reserved	
0x4002 2000 – 0x4002 23FF	Flash memory interface	
0x4002 3000 – 0x4002 33FF	CRC	
0x4002 3400 – 0x5003 FFFF	Reserved	

그림 16 교재 표 8.1 일부

9.2.4 Port output data register (GPIOx_ODR) (x=A..G)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y= 0 .. 15)

These bits can be read and written by software and can be accessed in Word mode only.

Note: For atomic bit set/reset, the ODR bits can be individually set and cleared by writing to the GPIOx_BSRR register (x = A .. G).

그림 17 GPIOx_ODR 설명

● **Line 43에서 초기화되는 레지스터의 역할은 무엇인가? 이 line을 통해 어떤 값이 이 레지스터에 저장되는가?**

Line 43의 `DMA1_Channel2->CMAR = (u32)font8x8[data];` 에서 `DMA1_Channel2->CMAR` 레지스터는 DMA1의 channel2에서 사용하는 메모리 주소를 저장하는 레지스터이다. 즉 `font8x8`이라는 2차원 배열에서 `font8x8[data]`에 해당하는 데이터가 저장된 주소의 시작번지를 `DMA1_Channel2`의 메모리 주소로 사용한다.

즉 여기까지의 분석을 요약해서 DMA의 동작을 설명하자면 `font8x8[data][0]`의 주소부터 8-bits씩 메모리의 주소를 증가시켜가며(즉 `font8x8[data][1]`, `font8x8[data][2]` ... 순으로) 8개의 데이터(즉 `font8x8[data][0]`부터 `font8x8[data][7]` 까지의 데이터)를 `GPIOC->ODR`로 데이터를 이동시키는데 이를 계속해서 반복해서 수행한다.

STEP 5: Lines 45, 47의 역할을 설명해보자.

먼저 line 45의 `DMA1_Channel2->CCR |= 0x00000001;` 의 코드를 통해서 `DMA1_Channel2->CCR` 레지스터의 0번째 bit인 `EN` bit를 1로 set하고 나머지는 그대로 두었다. 이를 통해 DMA1의 channel2를 enable해주었다. 다음으로 line 47의 `TIM2->CR1 |= 0x0001;` 을 통해서 `TIM2->CR1`의 0번째 bit인 `CEN` bit를 1로 set하고 나머지는 그대로 두었다. 이를 통해 TIM2의 카운터를 enable해주었다. 즉 line 45와 line 47을 통해 DMA와 TIM가 동작하도록 각각을 enable 해주었다.

STEP 6: Lines 53-60은 dot matrix display의 동작 속성상 row scan이 지원되어야 하는데, DMA request로 사용되는 TIM2의 update event를 인터럽트와 연동하여 구현하고 있다. 이 handler에서 처리되고 있는 내용을 파악한 후 DMA1 channel2가 수행하는 동작을 설명해보자.

Lines 53-60은 `TIM_2_IRQHandler`로 TIM2에 인터럽트가 발생하였을 때의 동작에 관한 코드이다. 먼저 line 54에서 `TIM2->SR & 0x0001`을 통해서 TIM2 핸들러가 호출되었을 때 update interrupt인지 여부를 확인하고 update interrupt가 발생하였다면 lines 55-58의 코드를 수행한다. Lines 55-58 코드는 `GPIOB`에 연결되어 있는 dot matrix의 row중 하나에만 0으로 reset하여 해당 row를 활성화시킨다. 그리고 row를 하나 증가시키고 만약 0x100이 되어 범위를 벗어나면 다시 0x001로 설정한다. 그리고 TIM2의 update interrupt flag를 0으로 reset해준다. 우리가 위에서 설정한 대로 TIM2

의 update event가 발생하면 DMA1의 channel2에 request가 발생한다. 따라서 TIM2의 update event가 발생하면 TIM2의 interrupt가 발생하 row하나 증가시키고 동시에 DMA1의 channel2에 request가 발생하여 font8x8에 저장된 데이터를 GPIOC->ODR로 가져와 dot matrix의 col에 해당하는 데이터가 세팅된다. 따라서 TIM2의 update event가 발생할 때 마다 interrupt가 row를 설정하고 DMA가 col을 설정하는 역할을 수행한다.

STEP 7: 이 프로그램이 예상한 대로 동작하는지 data의 내용을 변경하면서 확인한다. 이상의 과정을 통해 확인한 내용들을 바탕으로 DMA가 동작하기 시작하면 구체적으로 메모리의 어떤 부분이 입출력장치의 어떤 부분으로 전달될 것인지 그림으로 표현해보자. 그림 11.1을 활용하되, 구체적인 주소들을 표기한다.

아래 [그림 18]을 통해서 data내용에 따른 dot matrix의 동작을 확인할 수 있다. 또한 아래 [그림 19]는 DMA가 동작할 때 메모리의 font8x8[data]의 데이터가 입출력장치의 0x4001100C인 GPIOC->ODR로 데이터가 이동하는 것을 나타낸 그림이다. 이때 font8x8[data]가 저장된 구체적인 주소는 프로그램이 수행될 때 결정되기 때문에 명시하지 못하였다.

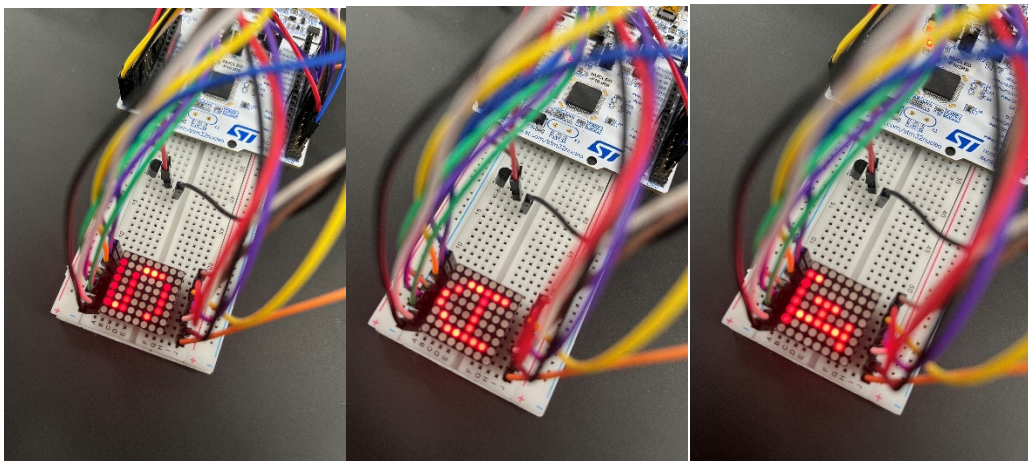


그림 18 다양한 data 값에 따른 결과(data = 0x05, 0x0A, 0x0F)

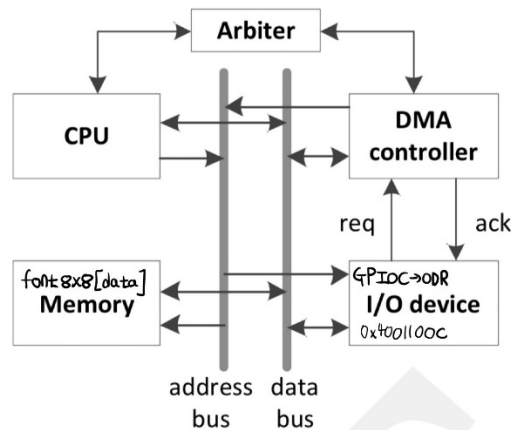


Figure 11.1: A system with DMA.

그림 19 DMA 동작 묘사

2) 실험 2

STEP 8: Program 11.2는 Program 10.1의 array에 저장된 string data를 USART1을 이용하여 외부로 송신하는 모듈이다. Program 10.1과 달리 데이터의 출력에 DMA를 이용한다. 이 프로그램을 이용하여 프로젝트를 생성한다.

```

1  #include <stm32f10x.h>
2
3  #define USART1_DR_BASE 0x40013804
4  u8 txdata[]="DMA1/USART2 interface\r\n";
5
6  int main(void) {
7      RCC->APB2ENR |= 0x00004004;
8      GPIOA->CRH &= ~(0xFFu << 4);
9      GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000000;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000008;
16     USART1->CR3 |= 0x00000080;
17
18     RCC->AHBENR |= 0x00000001;
19     DMA1_Channel4->CCR = 0x00000090;
20     DMA1_Channel4->CNDTR = sizeof(txdata)/sizeof(*(txdata)) - 1;
21     DMA1_Channel4->CPAR = USART1_DR_BASE;
22     DMA1_Channel4->CMAR = (u32)txdata;
23
24     DMA1_Channel4->CCR |= 0x00000001;
25
26     USART1->CR1 |= 0x00002000;
27
28     while(!(DMA1->ISR & 0x00002000)) {};
29
30     while(1) {};
31 } // end main

```

그림 20 직접 작성한 lab11_2.c 코드

STEP 9: Lines 7-16의 초기화 과정을 Program 10.1의 해당 부분과 비교해보자. 달라진 부분이 있다면 무엇을 위함인지 그 의미를 reference manual(STMicroelectronics(2011))을 통해 확인해보자.

```

7  RCC->APB2ENR = 0x00004004;
8  GPIOA->CRH &= ~(0xFFu << 4);
9  GPIOA->CRH |= (0x04B << 4);
10
11  USART1->BRR = 0x0EA6;
12  USART1->CR1 = 0x00000000;
13  USART1->CR2 = 0x00000000;
14  USART1->CR3 = 0x00000000;
15  USART1->CR1 |= 0x00000008;

```

그림 21 프로그램 10.1 초기화 코드

위 [그림 21]은 이번 프로그램 11.2의 lines 7-16에 해당하는 프로그램 10.1에서의 초기화 코드이다. 차이점을 살펴보면 나머지 모든 부분이 동일하고 프로그램 11.2에서는 line 16의 USART1->CR3 |= 0x00000008; 코드만 추가되었다. USART1->CR3 레지스터의 7번 bit인 DMAT bit만 추가적으로 1로 set하였는데 이를 통해서 DMA mode가 transmission에 대해서 enable해주었다. 즉 프로그램 10.1과 다른 부분은 모두 동일하고 프로그램 11.2에서는 transmission에 대해서 DMA mode를 enable해주었다.

STEP 10: Lines 18-22는 DMA를 초기화하는 과정을 보여준다. 이 초기화 과정을 다음과 같이 단계적으로 파악해보자.

● Line 18은 무엇을 위함인가? Reference manual(STMicroelectronics (2011))에서 해당부분을 찾아 확인한다.

Line 18의 RCC->AHBENR |= 0x00000001; 을 통해서 STEP 4에서 설명한 대로 RCC->AHBENR 레지스터의 0번 bit인 DMA1EN bit를 1로 set하고 기존의 다른 값은 건드리지 않도록 or 연산을 통해 동작을 수행하였다. 이를 통해 DMA1 clock이 AHB bus에 연결되도록 enable되었다.

● 이 과정에서 Channel4를 초기화하는 이유는 무엇인가? 표 11.3를 참조하여 그 이유를 설명한다.

STEP 4의 [그림 15]는 교재의 표 11.3의 내용이다. [그림 15]를 보면 USART1_TX는 channel 4와 연결되어 있는 것을 확인할 수 있다. 따라서 실험 11.2에서는 USART1_TX를 DMA와 연결하기 위해서 channel 4를 초기화 하였다.

● Line 19를 통해 DMA의 해당 channel이 어떻게 설정되고 있는가? DMA를 통해 구현될 동작을 염두에 두고 다음을 중심으로 그 의미와 함께 확인해 보자.

-MSIZE

-PSIZE

-MINC

-PINC

-DIR

Line 19의 DMA1_Channel4->CCR = 0x00000090; 을 통해서 Channel4의 CCR 레지스터의 4번, 7번 bit를 1로 set하고 나머지는 0으로 설정하였다. 이를 통해 DMA동작에 대해 분석해보면 MSIZE는 MSZIE bit가 00이기 때문에 8-bits이고 PSIZE도 PSIZE bit가 00임으로 8-bits이다. MNIC bit는 1임으로 memory increment mode가 enable되어있고 PINC bit는 0임으로 peripheral increment mode는 disable되어있다. DIR bit는 1임으로 memory에서 데이터를 읽어드리는 방향으로 동작한다.

이를 바탕으로 DMA의 동작을 살펴보면 주변장치와 메모리 모두 8-bits씩 데이터가 이동하며 DIR bit가 1임으로 메모리에서 주변장치로 8-bits 데이터가 이동한다. 또한 PINC는 0임으로 데이터를 전송받는 주변장치의 주소는 그대로이고 MINC는 1임으로 메모리 주소는 DMA가 데이터를 이동시키고 나서 자동으로 주소가 증가한다. 즉 메모리 주소를 증가시키며 메모리 데이터를 8-bits씩 고정된 주변장치의 주소로 이동시킨다.

● **Line 20에서 초기화되는 레지스터의 역할은 무엇인가? 이 line을 통해 어떤 값이 이 register에 저장되는가?**

Line 20의 DMA1_Channel4->CNDTR = sizeof(txdata)/sizeof(*(txdata)) - 1; 을 통해서 CNDTR 레지스터에 sizeof(txdata)/sizeof(*(txdata)) - 1값을 저장하여 해당 숫자만큼 데이터를 이동시킨다. sizeof(txdata)/sizeof(*(txdata)) - 1를 계산해보면 txdata는 u8 데이터형 가지고 있는 문자열 임으로 txdata 문자열의 길이(23) + 1 = 24가 sizeof(txdata)이다. 그리고 sizeof(*(txdata))는 *(txdata)의 크기인데 *(txdata)는 txdata[1]과 같다. 즉 문자 하나의 크기임으로 sizeof(*(txdata))는 1이 된다. 따라서 $24/1 - 1 = 23$ 임으로 DMA1_Channel4->CNDTR에는 txdata문자열에 저장된 문자의 개수인 23이 저장된다.

● **Line 21에서 초기화되는 레지스터의 역할은 무엇인가? 이 line을 통해 이 레지스터에 저장되는 값의 근거를 표 8.1와 표 11.4을 통해 확인한다.**

Line 21의 DMA1_Channel4->CPAR = USART1_DR_BASE; 인데 USART1_DR_BASE는 #define에 의해서 0x40013804이다. 즉 DMA1_Channel4->CPAR = 0x40013804; 라고 생각할 수 있다. 먼저 표

11.4는 DMA의 register map인데 DMA1_Channel4->CPAR는 DMA1의 channel4의 CPAR레지스터는 주변장치의 주소를 저장하는 레지스터이다. 그리고 STEP 4의 [그림 16]은 교재의 표 8.1인데 0x40013804는 USART1의 주소에 속하는 것을 확인할 수 있다. 또한 USART1의 base 주소인 0x40013800에서 0x04 offset만큼 떨어져 있는 것을 알 수 있다. 그리고 아래 [그림 22]의 USART_DR 레지스터를 보면 offset이 0x04인 것을 확인할 수 있다. 즉 0x40013804는 USART1의 DR레지스터의 주소인 것을 확인할 수 있다. 즉 line 21을 통해서 DMA1 channel4의 주변장치 주소로 USART1_DR레지스터를 지정하고 있는 것을 확인할 수 있다.

27.6.2 Data register (USART_DR)

Address offset: 0x04

Reset value: Undefined

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								DR[8:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:9 Reserved, forced by hardware to 0.

Bits 8:0 **DR[8:0]**: Data value

Contains the Received or Transmitted data character, depending on whether it is read from or written to.

The Data register performs a double function (read and write) since it is composed of two registers, one for transmission (TDR) and one for reception (RDR)

The TDR register provides the parallel interface between the internal bus and the output shift register (see Figure 1).

The RDR register provides the parallel interface between the input shift register and the internal bus.

When transmitting with the parity enabled (PCE bit set to 1 in the USART_CR1 register), the value written in the MSB (bit 7 or bit 8 depending on the data length) has no effect because it is replaced by the parity.

When receiving with the parity enabled, the value read in the MSB bit is the received parity bit.

그림 22 USART_DR 레지스터 설명

● Line 22에서 초기화되는 레지스터의 역할은 무엇인가? 이 line을 통해 어떤 값이 레지스터에 저장되는가?

Line 22의 DMA1_Channel4->CMAR = (u32)txdata; 를 통해서 DMA1 channel4에서 사용하는 메모리 주소로 txdata로 설정하였다. 즉 txdata 문자열의 시작주소를 DMA의 메모리 주소로 설정하였다.

STEP 11: Lines 24, 26의 의미는 무엇인지 각각에 대해 확인하고 설명해보자. 또한 line 28

의 의미는 무엇인가?

이 방식을 이용하여 DMA를 polling하는 대신에 interrupt를 이용하여 구현한다면 어떤 부분이 어떻게 수정/추가되어야 하는가? 해당 interrupt service routine에서는 어떤 조치를 해 주어야 하는가?

Line 24의 DMA1_Channel4->CCR |= 0x00000001; 을 통해서 기존 설정은 그대로 둔 채로 DMA1 channel4의 CCR레지스터의 0번 bit인 EN bit를 1로 set하여 DMA1의 channel4를 enable해주었다.

그리고 line 26의 USART1->CR1 |= 0x00002000; 을 통해서 기존 설정은 그대로 둔 채로 USART1의 CR1레지스터의 13번 bit인 UE bit를 1로 set하여 USART1을 enable해주었다.

또한 DMA를 polling하는 대신에 interrupt를 이용하여 구현하기 위해서는 pooling위한 코드인 line 28의 while문을 삭제한다. 그리고 세팅 과정에서 DMA1의 channel4에 의한 인터럽트가 발생할 수 있도록 NVIC->ISER[0] = (1 << 12); 코드를 추가하여 준다. 또한 DMA1_Channel4->CCR |= 0x00000002; 를 추가하여 DMA transfer가 complete되고 인터럽트가 발생할 수 있도록 설정해준다. 그리고 DMA1_Channel4_IRQHandler에서는 먼저 DMA1->ISR 레지스터에서 0x2000과 AND로 마스킹하여 TCIF4 bit가 1인지 먼저 확인하여 channel 4에서 transfer complete되어 interrupt가 발생하였는지 확인한다. 그리고 해당 TCIF4 bit를 0으로 reset해주고 핸들러를 나가 원래의 코드를 수행한다.

STEP 12: 이 프로그램이 Program 10.1과 같이 동작하는지 확인하기 위해 수신 보드에는 Program 10.1의 Rx module을 수행한다. 이 때 이 프로그램의 txdata[]의 크기에 따라 Rx module의 string[]의 크기를 증가시켜야할 수도 있다. 이상의 과정을 통해 확인한 내용들을 바탕으로 DMA가 동작하기 시작하면 구체적으로 메모리의 어떤 부분이 입출력장치의 어떤 부분으로 전달된 것인지 그림으로 표현해보자. 그림 11.1을 활용하되, 구체적인 주소들을 표기한다.

프로그램 10.1의 Rx 모듈과 프로그램 11.2를 Tx 모듈로 하여 USART 통신을 수행하였다. 그 결과 [그림 23]과 같이 txdata에 들어있던 내용이 프로그램 10.1의 string으로 잘 전달된 것을 디버거를 통해 확인할 수 있었다. 또한 DMA가 동작하기 시작하면 [그림 24]처럼 메모리의 txdata[]의 데이터가 입출력장치의 USART1_DR 레지스터 주소인 0x40013804로 이동시키는 역할을 수행한다.

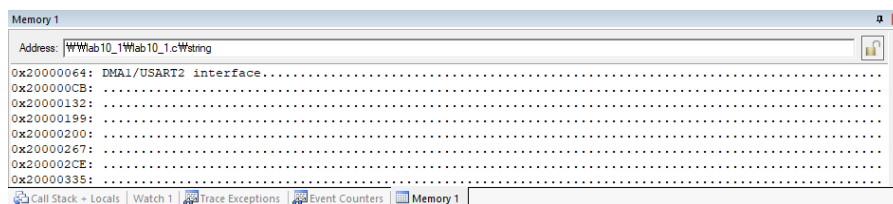


그림 23 프로그램 11.2 수행 결과

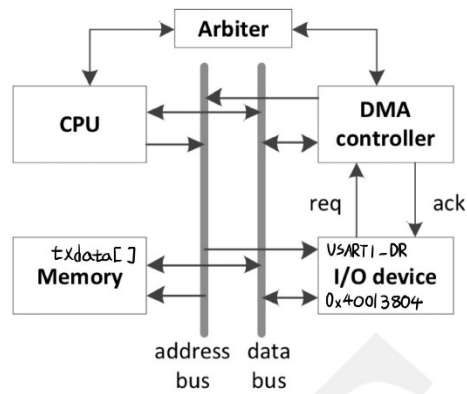


Figure 11.1: A system with DMA.
그림 24 프로그램 11.2의 DMA 동작

3) 실험 3

STEP 13: Program 11.3은 마이크로컨트롤러에 내장된 ADC를 polling 방식으로 구동하는 방법을 확인하기 위해 최대한 간결하게 작성된 프로그램이다. 이 프로그램을 이용하여 프로젝트를 생성한다.

```

1  #include <stm32f10x.h>
2
3  u16 ADConverted;
4
5  int main(void) {
6      RCC->APB2ENR |= 0x00000005;
7      GPIOA->CRL = 0x44404444;
8
9      RCC->APB2ENR |= 0x00000200;
10     ADC1->CR1 = 0x00000000;
11     ADC1->CR2 = 0x001E0002;
12     ADC1->SMPR2 = 0x00007000;
13     ADC1->SQR1 = 0x00000000;
14     ADC1->SQR2 = 0x00000000;
15     ADC1->SQR3 = 0x00000004;
16
17     ADC1->CR2 |= 0x00000001;
18     ADC1->CR2 |= (1 << 22);
19
20     while(1) {
21         while(!(ADC1->SR & 0x02));
22         ADConverted = ADC1->DR;
23     }
24 }

```

그림 25 직접 작성한 lab11_3.c 코드

STEP 14: Lines 6-7의 초기화 과정은 무엇을 위함인가? 그 의미를 reference manual(STMicroelectronics (2011))을 통해 확인해보자.

Line 6의 `RCC->APB2ENR = 0x00000005;` 를 통해서 `RCC->APB2ENR` 레지스터의 0번, 2번 bit를 1로 set하였다. 즉 AFIO와 GPIOA가 APB2 bus를 이용할 수 있도록 enable해주었다. 또한 line 7의 `GPIOA->CRL = 0x44404444;` 를 통해서 `GPIOA->CRL`의 default 값인 `0x44444444`에서 `0x44404444`로 바꾸어 PA4를 analog input모드로 사용할 수 있도록 설정해주었다. 왜냐하면 우리가 실험 11.3

에서 사용하는 온도센서의 Vout을 PA4로 연결하여 값은 입력 받을 것이기 때문이다.

STEP 15: Lines 9-18은 ADC를 초기화하는 과정을 보여준다. 이 초기화 과정을 다음과 같이 단계적으로 파악해보자.

● **Line 9는 무엇을 위함인가? Reference manual(STMicroelectronics (2011))에서 해당부분을 찾아 확인한다.**

Line 9의 `RCC->APB2ENR |= 0x00000200;` 을 통해서 `RCC->APB2ENR`의 나머지는 변화없이 9번 bit인 `ADC1EN` bit를 1로 설정하여 실험에서 사용할 ADC1이 APB2 bus를 사용할 수 있도록 enable 해주었다.

● **Lines 10-11의 CR 초기화 과정에서 어떤 설정이 이뤄지는지 표 11.5를 참조하여 확인한다.**

교재 표 11.5는 ADC의 register map이다. Line 10의 `ADC1->CR1 = 0x00000000;` 을 통해서 default 값과 동일하게 유지하였다. 그리고 line 11의 `ADC1->CR2 = 0x001E0002;` 를 통해서 `ADC1->CR2` 레지스터의 1번, 17, 18, 19, 20번 bit를 1로 set하고 나머지 bit는 0으로 설정하였다. `ADC1->CR2`의 1번 bit인 `CONT` bit를 1로 set하여 ADC가 continuous conversion mode로 동작하도록 하였다. 또한 17, 18, 19번 bit인 `EXTSEL`을 111로 set하여 ADC가 시작되는 external trigger로 `SWSTART`로 설정하여 `SWSTART`가 1로 set되면 conversion이 시작되도록 하였다. 마지막으로 20번 bit인 `EXTTRIG` bit를 1로 set하여 external trigger시에 conversion이 시작할 수 있도록 설정하였다. 즉 세팅시 바로 conversion이 시작된다.

● **Lines 13-15의 설정과정의 의미를 어떻게 설명할 수 있는가? 11.5.2의 channel selection을 참고하여 확인한다.**

Line 13의 `ADC1->SQR1 = 0x00000000;` 과 line 14의 `ADC1->SQR2 = 0x00000000;` 를 통해서 ADC register map의 default 설정으로 해주었다. 다음으로 line 15의 `ADC1->SQR3 = 0x00000004;` 를 통해서 `ADC1->SQR3` 레지스터의 2번 bit를 1로 set하여 SQ1에 4를 저장하여 GPIO4와 연결된 `ADC1_IN4`인 4번 channel이 regular sequence의 첫 번째로 사용되도록 하였다. 그리고 `ADC1->SQR1`레지스터의 20, 21, 22, 23번 비트로 구성된 L값이 0000이기 때문에 1 conversion만이 일어난다. 즉 single conversion mode로 `ADC1_IN4`에서 한번의 conversion만 발생하여 `ADC1_DR`에 저장되고 conversion후에는 `EOC` flag가 set된다.

● **Lines 17-18의 설정과정은 무엇을 위함인가? Line 11의 동일한 레지스터에 대한 설정과 구분한 이유를 어떻게 설명할 수 있는가?**

Line 17의 `ADC1->CR2 |= 0x00000001;` 을 통해서 `ADC1->CR2` 레지스터의 0번 bit인 `ADON` bit를 1로 set하고 나머지 세팅은 그대로 두었다. `ADON` bit를 1로 set하여 `ADC converter`를 ON으로 켜주어 `ADC`가 동작하도록 하였다. 그리고 line 18의 `ADC1->CR2 |= (1 << 22);` 를 통해서 `ADC1->CR2`의 22번 bit인 `SWSTART` bit를 1로 set하여 위에서 line 11에서 설정하였듯 `SWSTART` bit가 1로 set되어 `external trigger`로 동작하여 `conversion`이 동작할 수 있도록 해주었다. Line 11과 나누어서 설정한 이유는 동시에 세팅을 하면 `SWSTART` 신호에 의해 `external trigger`가 발생할지 알 수 있기 때문에 먼저 세팅을 해주고 `trigger`를 제공하였다고 생각할 수 있다. 또한 `ADON` bit도 마찬가지로 모든 설정을 마치고 `ADC`를 켜주어 우리가 원하는 동작을 정상적으로 수행할 수 있도록 모든 설정 이후에 `ADC`를 시작할 수 있도록 해준 것이다.

STEP 16: Line 21은 polling 과정을 보여준다. EOC의 의미는 무엇인가? ADC에서는 이 flag가 왜 중요한가? 언제 set되고 또 어떻게 reset되는가? 이 line을 주석처리한다면 ADC 결과를 저장하는 line 22 동작에는 어떤 영향을 미칠 수 있는가?

Line 21의 `while(!(ADC->SR & 0x02));` 는 `ADC->SR & 0x02`가 1이면 while문을 탈출한다. 즉 `ADC->SR` 레지스터의 1번 bit인 `EOC` bit가 1이 되면 while문을 탈출하며 그때까지 pooling을 하고 있다. 여기서 `EOC` bit는 `end of conversion`으로 `channel group`의 `analog` 데이터가 디지털 데이터로 `conversion`이 끝나면 자동으로 1로 set되는 bit이다. 이번 실험 11.3에서는 `channel group`이 `channel4`만 있는 `single group`이기 때문에 `channel4`의 데이터의 `conversion`이 끝나면 `EOC` bit가 1로 set된다. 우리가 사용하는 보드에서 `ADC`는 `SAR`방식으로 아날로그 입력이 디지털 신호로 변환된다. 해당 방식은 한 번에 아날로그 데이터가 디지털 데이터로 변환되는 것이 아니라 한 사이클씩 진행되며 점점 아날로그 입력에 가까워지는 방식으로 변환을 한다. 따라서 데이터가 입력되고 전부 다 변환되기까지 일부 사이클이 소요된다. 따라서 해당 변환이 완료되었다고 알려주는 `EOC`가 1로 set되고 데이터를 가져가야 알맞은 데이터를 가져갈 수 있다. `EOC`는 위에서 말했듯 `conversion`이 완료되면 자동으로 set되며 해당 bit에 software에 의해 0을 써서 reset하거나 `ADC_DR` 레지스터를 읽으면 자동으로 reset된다. 만약 line 22를 주석처리하면 `ADC_DR`에서 변환

중인 데이터를 가져가 매번 알맞지 않은 값을 ADCConverted에 저장하게 될 것이다.

STEP 17: 이 프로그램이 의도한대로 동작하는지 ADCConverted에 저장되는 내용을 반복해서 확인해 보자.

- 254페이지의 그림을 참고해서 TMP36의 +Vs에는 CN7의 18번 핀(+5V)를, GND에는 CN7의 20번 핀(GND)를 각각 연결하고, Vout에는 CN7의 32핀(PA4)를 연결한다.

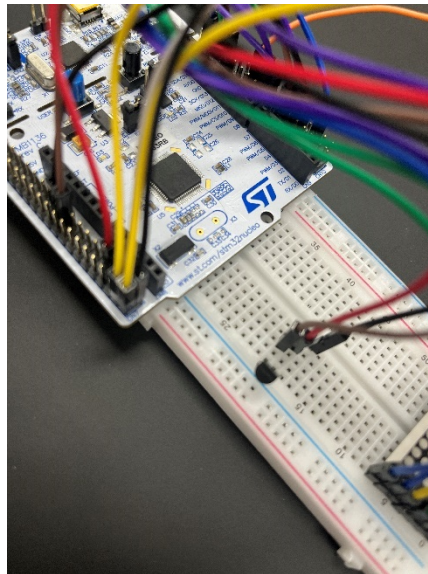


그림 26 실험 11.3을 위한 보드 연결

위 [그림]처럼 TMP36과 보드를 연결하였다.

- TMP36 패키지 부분을 손가락으로 잡았다 때는 동작을 반복하면서 온도가 변하도록 한다.



그림 27 상온에서의 출력

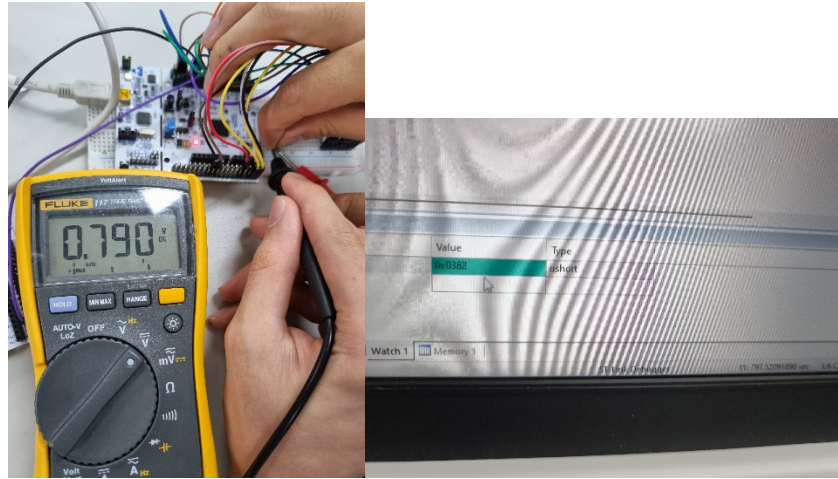


그림 28 손가락으로 잡았을 때 출력

[그림 27]과 [그림 28]과 같이 상온에서와 손가락으로 센서를 잡았을 때의 출력이 다르게 나오는 것을 확인할 수 있었다. 물론 위 사진에서 전류를 측정하는 순간과 디버거를 통해 ADC값을 측정하는 순간의 타이밍이 약간은 달라 정확하게 일치하는 값이 나오지는 않았다. 하지만 이를 바탕으로 ADC결과와 온도를 계산해보면 다음과 같다. 계산에는 <https://learn.adafruit.com/tmp36-temperature-sensor> 사이트의 아래 [그림 29]를 활용하였다. [그림 29]내부의 Figure 6은 TMP36소자의 온도와 출력전압에 대한 그래프인데 빨간 줄로 표시된 TMP36을 보면 [그림 29]의 온도 = $[(V_{out} \text{ in mV}) - 500] / 10$ 의 공식을 구할 수 있다.

How to Measure Temperature

Using the TMP36 is easy, simply connect the left pin to power (2.7-5.5V) and the right pin to ground. Then the middle pin will have an analog voltage that is directly proportional (linear) to the temperature. The analog voltage is independent of the power supply.

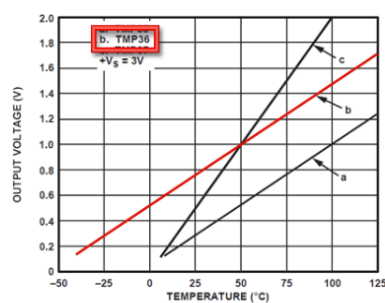


Figure 6. Output Voltage vs. Temperature

To convert the voltage to temperature, simply use the basic formula:

$$\text{Temp in } ^\circ\text{C} = [(V_{out} \text{ in mV}) - 500] / 10$$

So for example, if the voltage out is 1V that means that the temperature is $((1000 \text{ mV} - 500) / 10) = 50 ^\circ\text{C}$

If you're using a LM35 or similar, use line 'a' in the image above and the formula: $\text{Temp in } ^\circ\text{C} = (V_{out} \text{ in mV}) / 10$

그림 29 TMP36 데이터 계산법

상온에서의 Vout은 [그림 27]에 의해 767mV임을 알 수 있고 이를 [그림 29]의 식에 대입하면 온도는 26.7도씨가 나오며 손가락으로 온도센서에 열을 준 경우에는 [그림 28]에 의해 출력이 790mV이고 29도씨 정도로 온도가 오른 것을 확인할 수 있다. 그리고 ADC의 결과를 해석해보면 ADC 결과는 12bit로 저장되는데 0V일 때 0x000이 나오고 3.6V에서 0xFFF가 나온다. 따라서 ADC 데이터에 * 3.6 / 4095를 통해서 전압값을 구할 수 있다. 이에 대한 근거로는 STM32F103xB data sheet에서 가져온 아래 [그림 30]에 따르면 0V부터 3.6V를 conversion 해주는 것을 알 수 있다. 이를 통해 전압값을 구해보면 [그림 27]의 0x338은 724mV정도가 나오며 767mV와 약간의 차이는 있지만 잘 나온 것을 확인할 수 있다. 또한 [그림 28]의 0x382는 789mV정도가 나오며 790mV와 거의 오차없이 계산된 것을 확인할 수 있다.

- 2x 12-bit, 1 μ s A/D converters (up to 16 channels)
 - Conversion range: 0 to 3.6 V
 - Dual-sample and hold capability
 - Temperature sensor

그림 30 보드의 ADC관련 data sheet

4) 실험 4

STEP 18: Program 11.4을 이용하여 프로젝트를 생성한다. ADC는 데이터 변환을 반복적으로 수행하도록 초기화되며, 변환 결과는 DMA에 의해 메모리의 특정 주소에 저장된다.

```

1  #include <stm32f10x.h>
2  #define ADC1_DR_BASE 0x4001244c
3
4  ul6 ADCConverted;
5
6  int main(void) {
7      RCC->APB2ENR |= 0x00000005;
8      GPIOC->CRL = 0x44404444;
9
10     RCC->AHBENR |= 0x00000001;
11     DMA1_Channel1->CCR = 0x00003520;
12     DMA1_Channel1->CNDTR = 1;
13     DMA1_Channel1->CPAR = ADC1_DR_BASE;
14     DMA1_Channel1->CMAR = (u32)&ADCConverted;
15
16     RCC->APB2ENR |= 0x00000200;
17     ADC1->CR1 = 0x00000000;
18     ADC1->CR2 = 0x001E0102;
19     ADC1->SMPR2 = 0x00007000;
20     ADC1->SQR1 = 0x00000000;
21     ADC1->SQR2 = 0x00000000;
22     ADC1->SQR3 = 0x00000004;
23
24     DMA1_Channel1->CCR |= 0x00000001;
25     ADC1->CR2 |= 0x00000001;
26     ADC1->CR2 |= 0x00400000;
27
28     while(1) {;}
29 }
```

그림 31 직접 작성한 lab11_4.c 코드

STEP 19: 초기화 과정 중 lines 16 - 22를 Program 11.3의 해당 부분과 비교해본다. 어떤 부분이 달라졌는가? 그 의미는 무엇인가?

프로그램 11.3과 프로그램 11.4의 ADC1 초기화 과정 중 나머지 부분은 모두 동일하지만 line 18이 프로그램 11.3에서는 `ADC1->CR2 = 0x001E0002;` 인 반면 프로그램 11.4에서는 `ADC1->CR2 = 0x001E01002;` 로 `ADC1->CR2` 레지스터의 8번 bit인 DMA bit가 추가적으로 1로 set되어 있어서 DMA mode가 enable되어 DMA request를 요청할 수 있도록 설정하였다.

STEP 20: 초기화 과정 중 lines 10 - 14을 통해 설정되는 내용을 확인해보자. 이 과정에서 Channel1이 사용되는 근거는 무엇인가?

먼저 line 10의 `RCC->AHBENR = 0x00000001;` 을 통해서 0번 bit인 DMA1EN을 1로 set하여 DMA1을 AHB bus에 연결하여 enable해주었다. 다음으로 line 11의 `DMA1_Channel1->CCR = 0x00003520;` 을 통해서 5번, 8번, 10번, 12번, 13번 bit를 1로 set하고 나머지는 0으로 reset하였다. 그 결과 PL이 11이 되어 channel priority가 very high로 설정되었으며 MSIZE는 01로 memory size는 16-bits로 설정하였고 PSIZE도 01로 peripheral size도 16-bits로 설정하였다. 또한 CIRC bit도 1로 설정하여 circular mode로 동작하도록 하였으며 DIR bit는 0으로 data가 주변장치에서 메모리로 이동하도록 방향을 설정하였다. 다음으로 line 12의 `DMA1_Channel1->CNDTR = 1;` 을 통해 데이터 1개만 이동하도록 설정하였다. Line13의 `DMA1_Channel1->CPAR = ADC1_DR_BASE;` 를 통해서 DMA1의 channel1에서 사용하는 주변장치의 주소를 `ADC1_DR` 레지스터로 설정하고 line 13의 `DMA1_Channel1->CMAR = (u32)&ADCConverted;` 를 통해서 DMA1의 channel1에서 사용하는 메모리 주소로 `ADCConverted`라는 변수가 저장된 주소로 설정하였다. 요약하면 `ADC1_DR` 레지스터의 16-bits 데이터를 `ADCConverted` 변수에 1번 이동시키는 동작을 반복하도록 설정하였다. 이 과정에서 DMA1의 channel1을 사용하는 이유는 [그림 15]에 의하면 ADC1을 DMA와 연결하기 위해서는 DMA1의 channel1을 사용해야 하기 때문이다.

STEP 21: Line 24 - 26를 통해 구현되는 내용은 무엇인가? Line 별로 설명해보자. 특히 line 26를 수행한 후 ADC는 어떻게 동작하고 있을지 설명해보자.

Line 24의 DMA1_Channel1->CCR |= 0x00000001; 을 통해 기존 설정은 그대로 둔 채로 DMA1 channel1의 EN bit를 1로 set하여 DMA가 동작하도록 enable해주었다. Line 25의 ADC1->CR2 |= 0x00000001; 과 line 26의 ADC1->CR2 |= 0x00400000; 을 통해 각각 ADC1->CR2 레지스터의 ADON bit와 SWSTART bit를 1로 set하여 ADC1이 conversion을 시작하도록 설정하였다. Line 26이 후에는 lines 16 – 22의 설정에 따라서 SWSTART신호가 external trigger가 되어 ADC가 conversion을 시작하여 GPIOA4에서 들어오는 analog신호를 디지털 신호로 conversion하여 ADC1->DR 레지스터에 저장하고 EOF flag가 발생하면 DMA request를 요청하여 DMA에 의하여 conversion된 데이터가 메모리로 이동한다. 그리고 이 동작을 자동으로 반복해서 수행한다.

STEP 22: 이 프로그램이 의도한대로 동작하는지 ADCConverted에 저장되는 내용을 반복해서 확인해 보자.

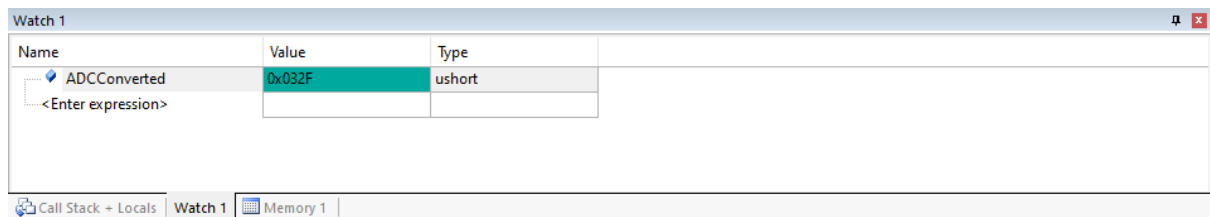


그림 32 상온에서의 결과

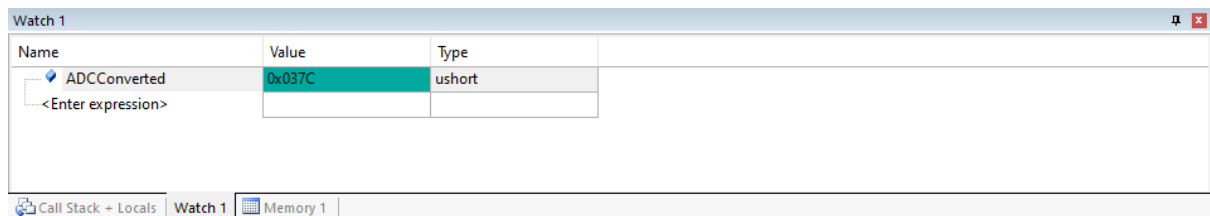


그림 33 손가락으로 잡았을 때 결과

[그림 32]와 [그림 33]과 같이 프로그램을 수행시키며 ADCConverted에 저장되는 값이 계속해서 바뀌는 것을 확인할 수 있었다. [그림 32]의 상온에서의 결과를 바탕으로 온도를 계산해보면 $ADCConverted * 3600 / 4095$ 로 Vout의 mV를 구하고 $[Vout - 500] / 10$ 를 통해 도씨를 구할 수 있다. 즉 $((0x032F * 3600 / 4095) - 500) / 10 = 21.6$ 도씨 가 되고 [그림 33]의 결과를 통해서도 $((0x037C * 3600 / 4095) - 500) / 10 = 28.4$ 도씨가 나오는 것을 확인할 수 있다.

4. Exercises

1) 실험과정에서 제시된 프로그램 중 DMA를 사용하는 Progam 1, 2, 4에서 CPU의 부담

(역할)이 DMA에 의해 어떻게 분산되었는지 설명하시오.

CPU는 우리가 작성한 코드를 바탕으로 만들어진 어셈블리어를 통해 명령어를 수행한다. 따라서 프로그램 11.1, 11.2, 11.4에서 기존 CPU가 반복적으로 데이터를 이동시키는 작업을 삭제하고 DMA에게 해당 역할을 부여하였으므로 CPU의 부담은 줄어들었다. 즉 DMA가 데이터 이동 작업을 수행함으로써 CPU는 데이터를 이동시키는데 어떠한 관여도 하지 않아도 됨으로 기존의 프로그램에 비해서 부담이 줄어들었다.

2) Program 11.1에서 DMA request는 dot matrix display의 scanning 주기를 이용하였고 이를 위해 timer의 update event 주기가 사용되었다. 데이터를 block 단위로 전송하는 communication, display 장치 또는 소자들과의 연동에서 DMA request가 어떻게 이뤄지는지 알아보자.

DMA의 설정에 따라서 주변장치와 메모리에서 이동하는 데이터의 크기는 8-bits이다. 또한 TIM2->DIER 레지스터의 8번 bit를 set하여 update DMA request를 enable하여 타이머에서 update event가 발생하면 DMA request를 요청하고 DMA는 request가 주어지면 정해진 대로 데이터를 메모리에서 주변장치로 전달한다. 즉 timer의 update event에 interrupt와 DMA request가 동시에 발생하여 col과 row의 데이터가 동시에 업데이트 되는 것이다.

3) ADC의 구동관점에서 변환이 완료되었는지를 확인하는 과정을 프로그램 및 관련 하드웨어 장치들과 연동하는 과정이 전체적인 시스템의 효율 확보차원에서 중요하다. Program 11.4에서는 이 부분이 어떤 설정과정을 통해 DMA와 연동되었는지 설명해보자. 또한 Program 11.3을 interrupt를 이용하여 구동한다면 어떤 부분을 어떻게 변경하여야 하는가?

프로그램 11.4에서 ADC1->CR 레지스터의 8번 bit인 DMA bit를 1로 set하여 ADC의 conversion이 완료되면 EOC가 발생하고 DMA request를 요청하도록 되어있다. 아래 reference manual의 [그림 34]를 보면 ADC에서 EOC가 발생하였을 때만 DMA request가 발생한다고 정리되어 있다.

11.8 DMA request

Since converted regular channels value are stored in a unique data register, it is necessary to use DMA for conversion of more than one regular channel. This avoids the loss of data already stored in the ADC_DR register.

Only the end of conversion of a regular channel generates a DMA request, which allows the transfer of its converted data from the ADC_DR register to the destination location selected by the user.

그림 34 ADC의 DMA request 설명

또한 프로그램 11.3을 pooling 방식이 아닌 interrupt 방식을 사용하기 위해서는 EOC가 발생하였을 때 interrupt를 요청할 수 있도록 ADC1->CR1 레지스터의 5번 bit인 EOCIE bit를 1로 set하여 EOC가 발생하였을 때 인터럽트가 발생할 수 있도록 설정해야 한다. 또한 인터럽트 핸들러 내부에서는 ADC1->SR 레지스터의 EOC bit를 확인해 EOC가 맞는지 확인하고 ADCConverted에 ADC1->DR 레지스터의 데이터를 가져오면 된다. 이때 ADC1->DR의 데이터를 읽어드리면 자동으로 EOC bit는 reset되기 때문에 따로 reset과정이 필요하지는 않다.

4) 실험에서 사용하는 마이크로컨트롤러에 온도측정 센서가 내장되었다고 한다. 어떤 목적으로 사용될 수 있는가? 이 센서를 사용하기 위해서는 어떤 과정이 필요한지 reference manual을 통해 확인해보자.

마이크로컨트롤러 내부의 온도측정 센서를 사용하면 내부 소자나 시스템의 온도를 측정할 수 있다. 이를 통해서 내부 시스템의 온도를 확인해 너무 많은 부하가 걸리고 있지는 않은 지 확인할 수 있다. 이러한 센서를 사용하기 위해서는 reference manual에 의하면 아래 [그림 35]의 절차를 따라서 온도센서를 사용할 수 있다. 즉 ADC_CR2 레지스터의 TSVREFE bit를 1로 set하고 ADCx_IN16 채널을 통해서 온도에 대한 디지털 전압 값을 측정해 공식을 통해 구할 수 있다.

Reading the temperature

To use the sensor:

1. Select the ADCx_IN16 input channel.
2. Select a sample time of 17.1 μ s
3. Set the TSVREFE bit in the [ADC control register 2 \(ADC_CR2\)](#) to wake up the temperature sensor from power down mode.
4. Start the ADC conversion by setting the ADON bit (or by external trigger).
5. Read the resulting V_{SENSE} data in the ADC data register
6. Obtain the temperature using the following formula:
$$\text{Temperature (in } ^\circ\text{C)} = \{(V_{25} - V_{SENSE}) / \text{Avg_Slope}\} + 25.$$

Where,
 V_{25} = V_{SENSE} value for 25° C and
Avg_Slope = Average Slope for curve between Temperature vs. V_{SENSE} (given in mV/° C or μ V/° C).

Refer to the Electrical characteristics section for the actual values of V_{25} and Avg_Slope.

그림 35 온도센서 사용법

5. 추가 실험

1) DMA Overhead Evaluation

DMA를 사용하면 CPU를 사용할 때에 비해서 CPU의 부담을 줄일 수 있다는 장점이 있다. 하지만 DMA를 사용하기 위해서는 추가적으로 설정을 해야하는 등 아주 약간의 overhead가 발생한다. 따라서 프로그램 11.3의 pooling방식과 프로그램 11.4의 DMA방식의 소요시간을 확인해보고자 한다.

먼저 아래 [그림 36]은 pooling 방식을 사용할 때의 소요시간 측정을 진행한 것이다. 먼저 가장 왼쪽 사진의 시스템이 시작되고 시스템 기본설정을 마치고 main함수로 들어가는데 소요된 시간이 0.00028620sec이고 가운데 사진의 ADC세팅을 마치는데까지 소요된 시간이 0.00029020sec이고 오른쪽 사진의 첫번째 ADC conversion이 끝나고 데이터가 메모리에 저장되는데까지 걸린 시간이 0.00029330sec이다. 즉 설정에는 4μsec가 소요되었고 세팅이 완료되고 ADC conversion과 메모리 저장에는 3.1μsec가 소요되었다.

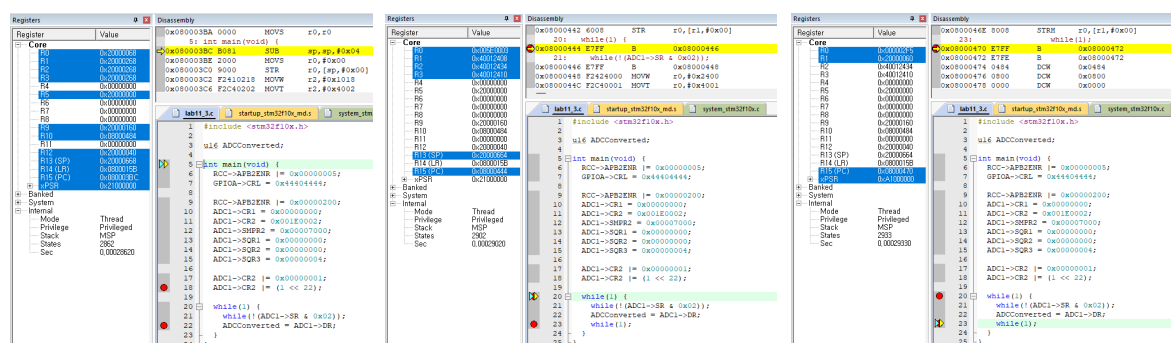


그림 36 Pooling 방식의 소요시간

다음으로 DMA 방식을 사용할 때의 소요시간을 측정하기 위해 프로그램 11.4에 DMA1->ISR레지스터에서 TCIF1을 bit가 1인지 확인해 Transmission이 끝났는지 확인하는 코드를 추가해주었다. 이를 바탕으로 아래 [그림 37]에서 볼 수 있듯이 main함수가 시작되기까지 0.00028620sec가 소요되었고 세팅까지 0.00029350sec가 소요되었고 DMA transmission 완료까지 0.00029510sec가 소요되었다. 즉 세팅은 7.3μsec가 소요되어 기존보다 3.3μsec가 더 소요되었다. 그리고 ADC되고 메모리에 저장되기 까지는 1.6μsec가 소요되었다.

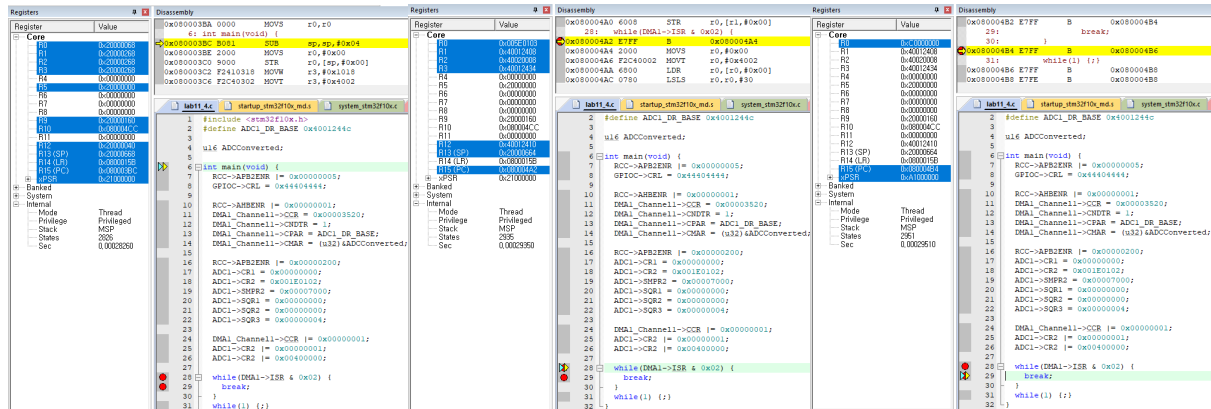


그림 37 DMA 방식의 소요시간

즉 두 방식을 비교해 보면 세팅에는 DMA 세팅까지 해야하는 DMA방식이 더 많은 시간이 소요되었지만 데이터가 저장되기 까지는 DMA방식은 CPU를 거치지 않고 전달되기 때문에 더 짧은 시간이 소요되었다. 이를 통해서 DMA를 활용하면 초기 세팅에서 약간의 overhead 발생하기 때문에 이동할 데이터가 적다면 Pooling방식을 사용하는 것이 시간적인 측면에서 더 효율적일 수 있지만 DMA방식은 세팅만 해두면 CPU의 자원을 소모하지 않고 데이터를 효율적으로 이동시킬 수 있기 때문에 DMA방식이 대부분 경우에 더 좋다고 말할 수 있다.

2) ADC Sampling Rate

우리가 사용하는 STM32F103RB보드에서 ADC를 사용할 때 sampling rate는 매우 중요하다. Sampling rate 계산하기 위해서는 reference manual의 [그림 38]을 참고한다. 기본적으로 ADC conversion을 하는데 12.5clock소요된다. 그리고 analog 신호를 sample하는 sampling time을 추가적으로 더해야 conversion을 하는데 소요되는 시간을 확인할 수 있다. Sampling time은 ADC_SMPR1 레지스터나 ADC_SMPR2 레지스터를 통해 채널별로 설정할 수 있으며 아래 [그림 39]와 같이 최소 1.5cycle에서 최대 239.5cycle을 사용할 수 있다.

Channel-by-channel programmable sample time

ADC samples the input voltage for a number of ADC_CLK cycles which can be modified using the SMP[2:0] bits in the ADC_SMPR1 and ADC_SMPR2 registers. Each channel can be sampled with a different sample time.

The total conversion time is calculated as follows:

$$T_{conv} = \text{Sampling time} + 12.5 \text{ cycles}$$

Example:

With an ADCCLK = 14 MHz and a sampling time of 1.5 cycles:

$$T_{conv} = 1.5 + 12.5 = 14 \text{ cycles} = 1 \mu\text{s}$$

그림 38 Sample time 계산

Bits 23:0 **SMPx[2:0]**: Channel x Sample time selection

These bits are written by software to select the sample time individually for each channel.
During sample cycles channel selection bits must remain unchanged.

000: 1.5 cycles
001: 7.5 cycles
010: 13.5 cycles
011: 28.5 cycles
100: 41.5 cycles
101: 55.5 cycles
110: 71.5 cycles
111: 239.5 cycles

그림 39 Sampling time 설정

따라서 최소 14cycles가 소요되는데 실제 소요시간을 확인할 때는 ADCCLK가 중요하다. 아래 [그림 40]을 보면 우측 하단에 ADCCLK는 최대 14MHz인 것을 확인할 수 있다. 따라서 ADC에서 최대한 빨리 conversion을 수행하더라도 14Mhz로 14clock이 소요됨으로 1μsec가 소요되며 sampling rate의 최대는 1/1μsec인 1Mhz이다.

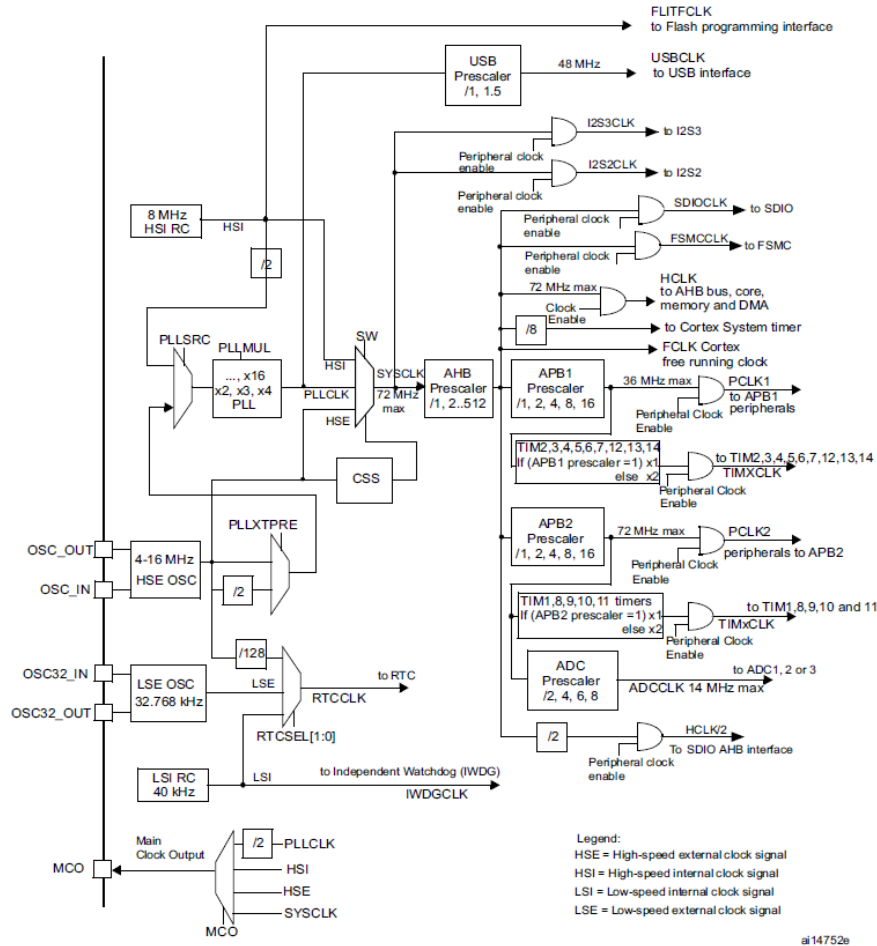
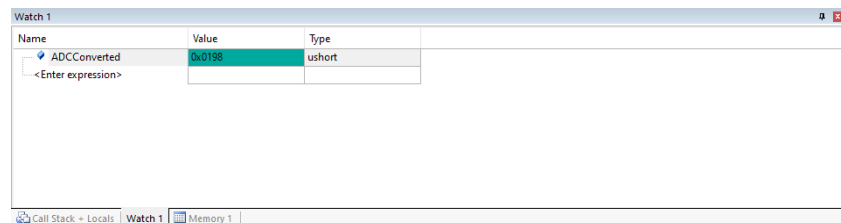


그림 40 Clock 정보

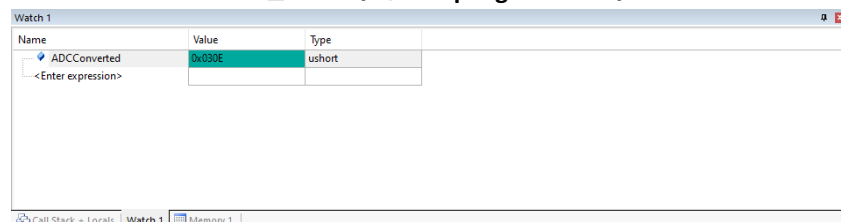
그리고 이를 활용하여 프로그램 11.3을 바탕으로 최대 sampling 주파수와 최소 sampling주파수로 실험을 진행해보았다. 기존 프로그램 11.3의 코드에서는 ADCCLK도 14MHz이고 channel4의 sample time를 설정하는 ADC_SMPR2 = 0x00007000을 저장하여 sample time을 239.5 cycle로 설정하였다. 이때 ADC_SMPR2 = 0x00000000을 저장하도록 하여 sample time을 1.5cycle로 감소시킨

결과 아래 [그림 41]과 같이 조금은 이상한 값이 계속해서 나왔다. 다음으로 RCC->CFGR 레지스터에서 AHB, APB2, ADC prescaler 값을 모두 최대로 해서 ADCCLK를 최대한 낮은 상태로 실험을 진행해본 결과 [그림 42]처럼 기존과 동일한 결과가 나왔다. 아무래도 온도센서의 경우는 계속해서 비슷한 값이 출력되다 보니 sampling rate가 높을 때는 이상한 결과가 나왔지만 낮을 때는 문제가 없었다. 하지만 나중에 주파수가 어느정도 있는 신호를 sampling 할 때는 Nyquist theorem에 따라 어느정도의 sampling rate를 확보해주어야 정상적인 결과가 나올 것이라 기대할 수 있다.



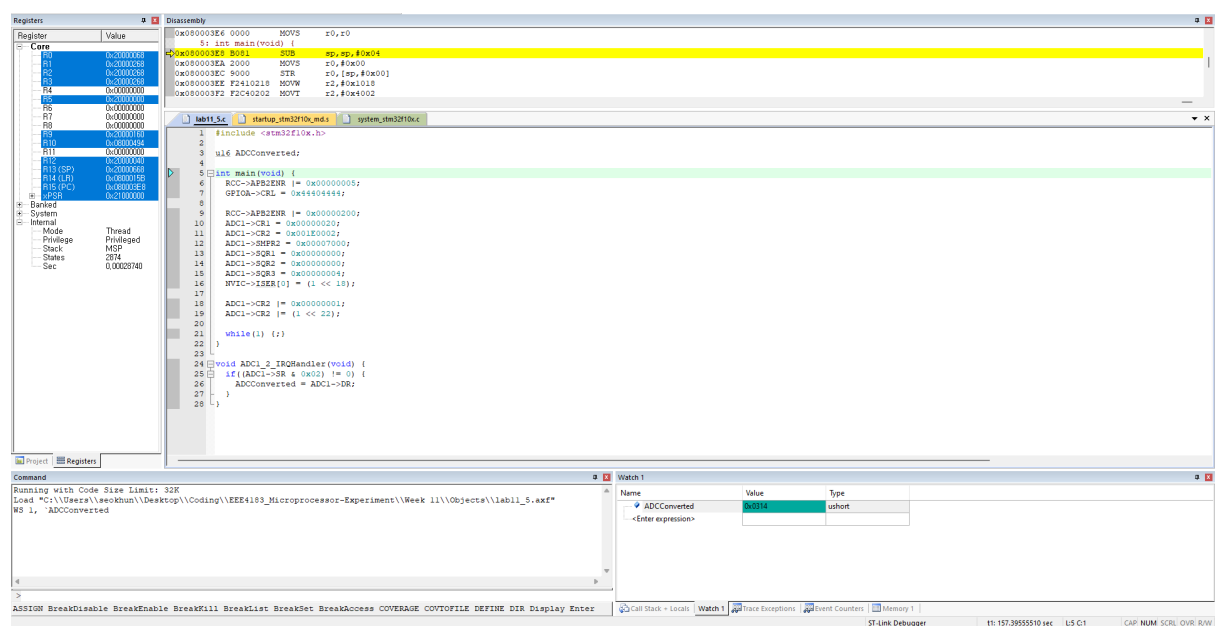
Name	Value	Type
ADCConverted	0x0198	ushort
<Enter expression>		

그림 41 최대 sampling rate 결과



Name	Value	Type
ADCConverted	0x030E	ushort
<Enter expression>		

3) ADC With Interrupt



The screenshot displays the ST-Link Debugger interface. On the left, the 'Registers' window shows the state of various registers. The main window shows the assembly code for the 'main' function, with the following key lines highlighted:

```

1: include <stm32f10x.h>
2:
3: ul6 ADCConverted;
4:
5: int main(void) {
6:     RCC->APB2ENR |= 0x00000005;
7:     GPIOA->CR1 = 0x44044444;
8:
9:     RCC->APB2ENR |= 0x00000200;
10:    ADC1->CR1 = 0x00000020;
11:    ADC1->CR2 = 0x00120022;
12:    ADC1->SMPR2 = 0x00007000;
13:    ADC1->SQR1 = 0x00000000;
14:    ADC1->SQR2 = 0x00000000;
15:    ADC1->SQR3 = 0x00000004;
16:    NYTC->ISER[0] = (1 << 18);
17:
18:    ADC1->CR2 |= 0x00000001;
19:    ADC1->CR2 |= (1 << 22);
20:
21:    while (1) {}
22:
23:
24: void ADC1_2_IRQHandler(void) {
25:     if ((ADC1->SR & 0x02) != 0) {
26:         ADCConverted = ADC1->DR;
27:     }
28: }

```

At the bottom, the 'Watch' window shows the value of 'ADCConverted' as 0x030E, which is consistent with the previous figure.

그림 42 ADC with interrupt

실험에서는 ADC를 polling 방식과 DMA 방식을 이용해서 구현해보았다. 따라서 추가실험에서는

ADC를 interrupt를 활용해 [그림 42]와 같이 구현해보았다. 먼저 ADC1->CR1 레지스터의 5번 bit 인 EOCIE bit를 1로 set하여 EOC에 의해 interrupt가 발생할 수 있도록 해주었다. 그리고 NVIC->ISER[0] = (1 << 22); 를 통해서 22번 interrupt인 ADC1_2 interrupt가 발생할 수 있도록 해주었고 인터럽트 핸들러에서는 ADC1에서 EOC가 set되어 있는지 확인하고 ADCConverted에 ADC1->DR 레지스터의 값을 가져왔다. 이를 통해 ADC에서 conversion이 끝날 때 마다 인터럽트가 발생하여 ADCConverted로 conversion된 값을 가져오도록 하였고 [그림 42]의 메모리 영역에서 확인 가능 하듯이 정상적으로 동작하였다.

4) Internal Temperature Sensor

Exercise 4번에서 내부의 온도 센서를 사용하는 것에 대해서 살펴보았다. 이를 직접 확인해보기 위해 아래 [그림 43]과 같이 코드를 작성하였다. Exercise 4번에서 알아본 대로 ADC1->SQR3 = 0x00000010; 을 통해 ADC1_IN16을 input channel로 사용하도록 하고 ADC1->SMPR1 = 0x001C0000; 을 통해서 channel 16의 sample time을 239.5 cycle로 설정하여 sample time은 17.1μ sec가 되도록 하였고 ADC1->CR2의 23번 bit인 TSVREFE bit를 1로 set하였다.

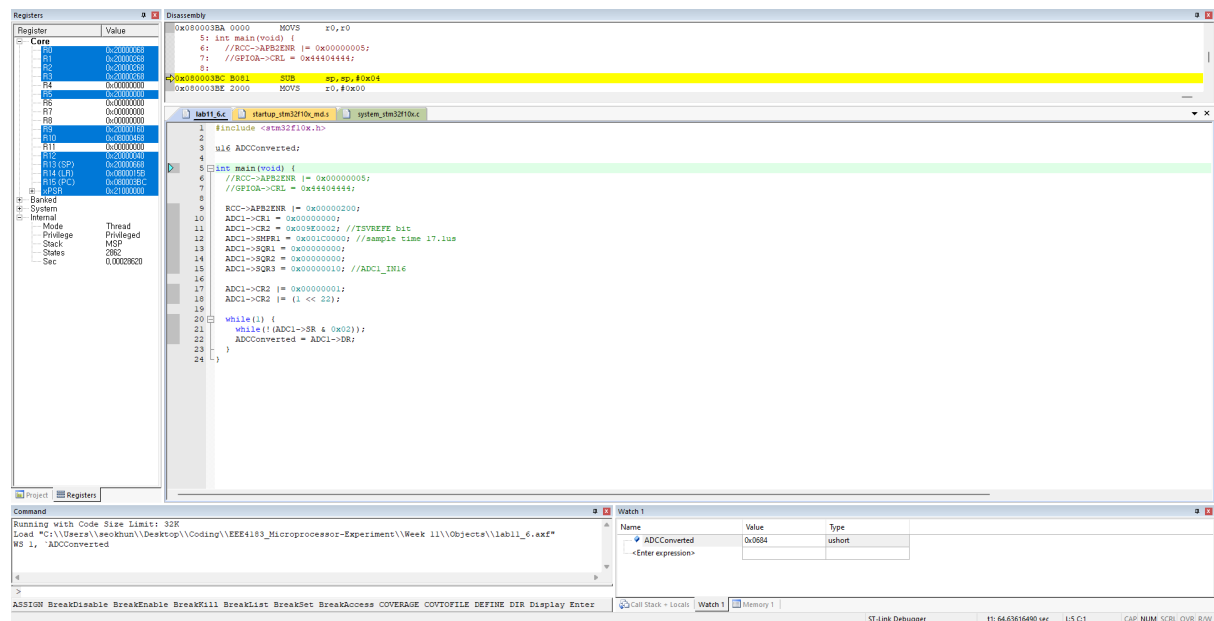


그림 43 Internal temperature sensor

Temperature sensor characteristics

Table 50. TS characteristics

Symbol	Parameter	Min	Typ	Max	Unit
$T_L^{(1)}$	V_{SENSE} linearity with temperature	-	± 1	± 2	$^{\circ}\text{C}$
$\text{Avg_Slope}^{(1)}$	Average slope	4.0	4.3	4.6	$\text{mV} / ^{\circ}\text{C}$
$V_{25}^{(1)}$	Voltage at 25°C	1.34	1.43	1.52	V
$t_{\text{START}}^{(2)}$	Startup time	4	-	10	μs
$T_{S_temp}^{(3)(2)}$	ADC sampling time when reading the temperature	-	-	17.1	

1. Evaluated by characterization, not tested in production, unless otherwise specified.

2. Specified by design, not tested in production.

3. Shortest sampling time can be determined in the application by multiple iterations.

그림 44 STM32F103RB 보드의 온도 관련 특성

그 결과 [그림 43]에서 확인 가능하듯이 ADCConverted에는 0x684가 저장되었고 이를 온도로 해석하기 위해서는 data sheet에서 가져온 [그림 44]의 소자 특성을 기반으로 계산하였다. 먼저 0x684를 Vsense로 바꾸면 $V_{\text{sense}} = 0x684 * 3300 * 4095\text{mV} = 1344\text{mV}$ 이다. 그리고 [그림 35]의 공식 $\text{온도} = (V_{25} - V_{\text{sense}} / \text{AVG_Slope}) + 25$ 에 대입하면 $\text{온도} = (1.43 - 1.344) / 4.3 + 25 = 25.02$ 도씨가 나오는 것을 확인할 수 있었다.

5) ADC / DMA Abstraction

```
#include <stm32f10x.h>

#define ADC1_DR_BASE 0x4001244c

u16 ADCConverted;

#define DMA_8bits 0x01
#define DMA_16bits 0x02
#define DMA_32bits 0x03

#define DMA_Priority_veryhigh 0x00003000
#define DMA_Priority_high 0x00002000
#define DMA_Priority_medium 0x00001000
#define DMA_Priority_low 0x00000000
```

```

#define DMA_MINC 0x00000080

#define DMA_PINC 0x00000040

#define DMA_CIRC 0x00000020

#define DMA_MEMTOPERI 0x00000010

void init_DMA1(int data_size, int data_num, int mode, int mem_addr, int peri_addr) {

    RCC->AHBENR |= 0x00000001;

    if(data_size == DMA_8bits) {

        DMA1_Channel1->CCR = 0x00000000;

    }

    else if(data_size == DMA_16bits) {

        DMA1_Channel1->CCR = 0x00000500;

    }

    else if(data_size == DMA_32bits) {

        DMA1_Channel1->CCR = 0x00000A00;

    }

    DMA1_Channel1->CCR |= mode;

    DMA1_Channel1->CNDTR = data_num;

    DMA1_Channel1->CPAR = peri_addr;

    DMA1_Channel1->CMAR = mem_addr;

    DMA1_Channel1->CCR |= 0x00000001;

}

#define ADC_CONTINOUS 0x00000002

#define ADC_DMA 0x00000100

```

```

#define ADC_EXTTRIG_SWSTART 0x001E0000

#define ADC_17us 0x00007000

void init_ADC1_Channel4(int mode, int sample_time){

    RCC->APB2ENR |= 0x00000200;

    ADC1->CR1 = 0x00000000;

    ADC1->CR2 |= mode;

    ADC1->SMPR2 = ADC_17us;

    ADC1->SQR1 = 0x00000000;

    ADC1->SQR2 = 0x00000000;

    ADC1->SQR3 = 0x00000004;


    ADC1->CR2 |= 0x00000001;

    ADC1->CR2 |= 0x00400000;

}

int main(void) {

    RCC->APB2ENR |= 0x00000005;

    GPIOC->CRL = 0x44404444;


    init_DMA1(DMA_16bits, 1, DMA_Priority_veryhigh | DMA_CIRC, (u32)&ADCConverted,
ADC1_DR_BASE);

    init_ADC1_Channel4(ADC_CONTINUOUS | ADC_DMA | ADC_EXTTRIG_SWSTART,
ADC_17us);


    while(1) {}

}

```

위의 코드처럼 프로그램 11.4를 기반으로 DMA1과 ADC1_Channel4를 편한게 세팅할 수 있도록

함수를 만들어 코드를 추상화하였다. 설정에 필요한 값들을 #define을 통해 설정하고 해당 추상화된 데이터로 값을 입력받아 가독성이 좋도록 만들었다. 그리고 아래 [그림 45]와 같이 원래의 코드처럼 정상적으로 동작하는 것을 확인할 수 있었다.

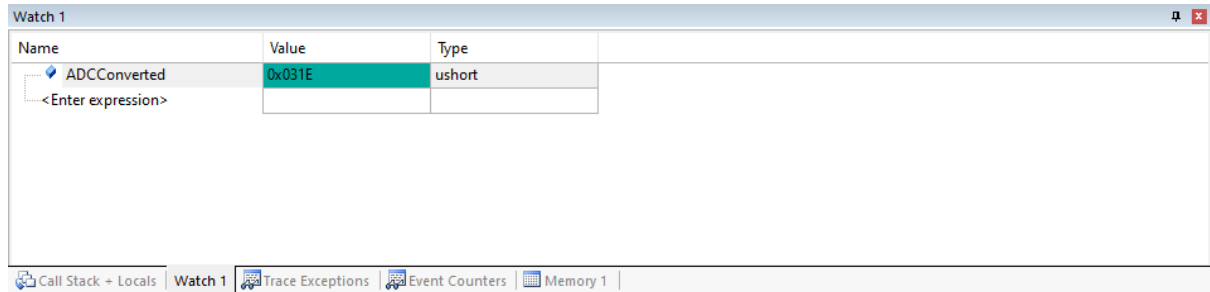


그림 45 추가실험 5 결과

6. 결론

이번 실험을 통해서 CPU대신 데이터를 이동시키는 동작을 수행해주는 DMA를 사용하기 위해 설정하는 방법과 DMA의 동작이 어떻게 이루어지는지 확인할 수 있었다. 또한 ADC 기능을 사용해서 외부 analog input을 해석하는 방법과 ADC 데이터를 읽어들이기 위해 pooling, interrupt, DMA를 모두 사용해서 ADC 기능을 사용해보았다. 이번 실험까지 통해서 STM32F103RB보드를 사용해서 수행해볼 수 있는 다양한 동작을 간단한 프로그램들을 통해서 익혀보았다.

7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론.

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6)

STMicroelectronics (2021). STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MUCs Reference Manual (Rev 21).

STMicroelectronics (2022). STM32F103x8, STM32F103xB Data Sheet (Rev 18).

Yiu, J. (2010). The Definitive Guide To The ARM Cortex-M3 (2nd Edition)

히언. (2021). EMBEDDED RECIPES.