마이크로프로세서응용실험 LAB09 결과보고서

Timers and Counters

20181536 엄석훈

1. 목적

● 타이머의 동작모드 설정에 사용되는 레지스터들의 구성과 초기화 방법을 이해한다.

● 타이머에 의해 발생하는 인터럽트의 구동방법을 GPIO와 연동되어 동작하는 소자/장치들을 통해 확인한다.

● 한 개 이상의 타이머들에 의해 발생하는 인터럽트의 동작을 확인한다.

● 타이머의 동작을 trigger source를 연동하여 활용하는 방법(reset, gated modes)에 대해 이해한다.

● 타이머의 clock에 스위치 신호를 연결하여 counter로 사용하는 방법을 이해한다.

2. 이론

1) Timer In STM32F10x

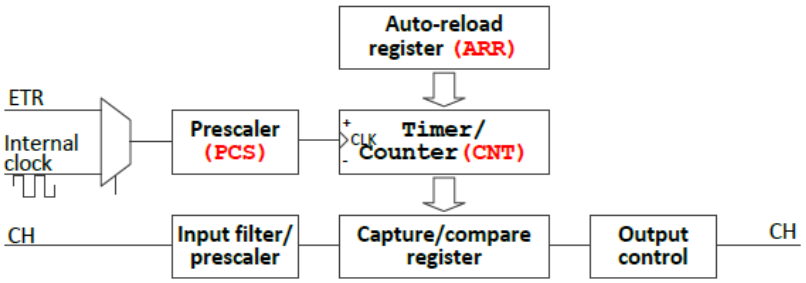


그림 1 타이머의 기본 구조

임베디드 시스템에서 타이머는 매우 중요한 요소이다. 이러한 타이머의 기본 구조는 위 [그림 1]과 같이 이루어져 있다. 우선 타이머의 기준 시간을 위부신호인 ETR과 내부 clock 중에서 선택할 수 있으며 prescaler를 통해 분주된 신호가 타이머에서 사용된다. 또한 타이머는 timer/counter 설

정에 따라서 up/down counting을 설정할 수 있다. 그리고 auto-reload register의 값이 타이머의 초기값으로 사용되며 capture/compare register를 통해서 타이머가 0또는 초기값에 도달하였는지 확인할 수 있다. 또한 동작중인 타이머의 값을 capture/compare register를 통해서 읽을 수 있다. 이 기능을 사용하면 CH로 들어오는 입력의 pulse 폭을 측정하는 등으로 활용할 수 있다.

우리가 사용하는 STM32F10x 보드에서는 서로 다른 목적의 타이머가 7개 존재한다. 간단히 하나씩 소개하면 OS를 위해 존재하지만 downcounter로도 사용 가능한 SysTick timer, 실험에서 주로 사용하는 Advanced-control timer인 TIM1와 General-purpose timer인 TM2, TM3, TM4과 소자 치기화 및 timeout관리에 사용하는 Independent watchdog, main clock에 의해 구동되는 window watchdog으로 총 7개의 타이머가 존재한다. 실험에서 주로 사용되는 TM1-TM4의 구조는 아래 [그림 2]와 같다.

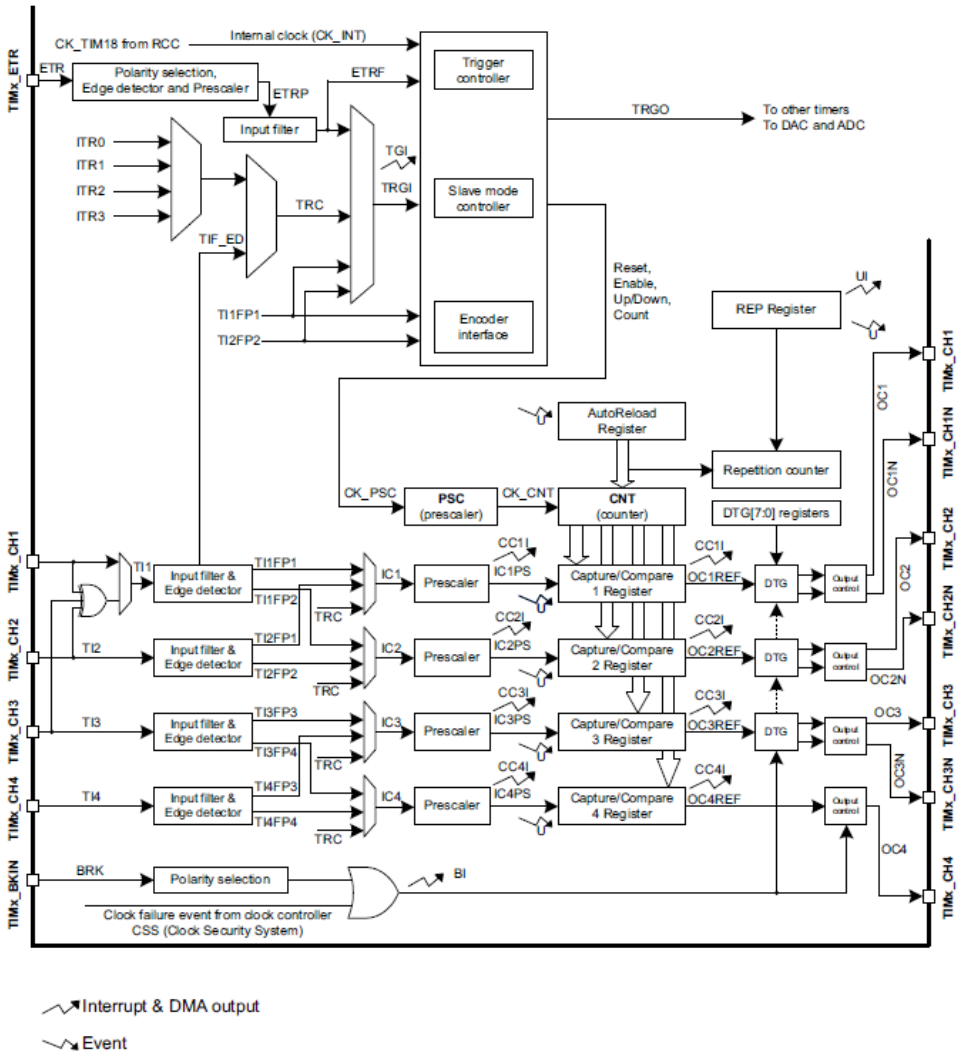


그림 2 TM1-TM4의 기본 구조

2) Clock input sources

카운터에서 사용할 clock은 internal clock, external clock mode1, external clock mode2, internal trigger inputs 총 4가지 중에 하나를 선택하여 사용할 수 있다.

Internal clock을 사용하기 위해서는 TIMx_SMCR의 SMS비트를 0으로 설정하고 TIMx_CR1의 CEN 비트를 1로 설정하면 프로세서 내부 신호인 CK_INT이 타이머의 clock신호가 되어 prescaler로 들어간다. 아래 [그림 3]은 내부 clock이 연결되었을 때 관련 신호들과 upcounter의 동작이다.

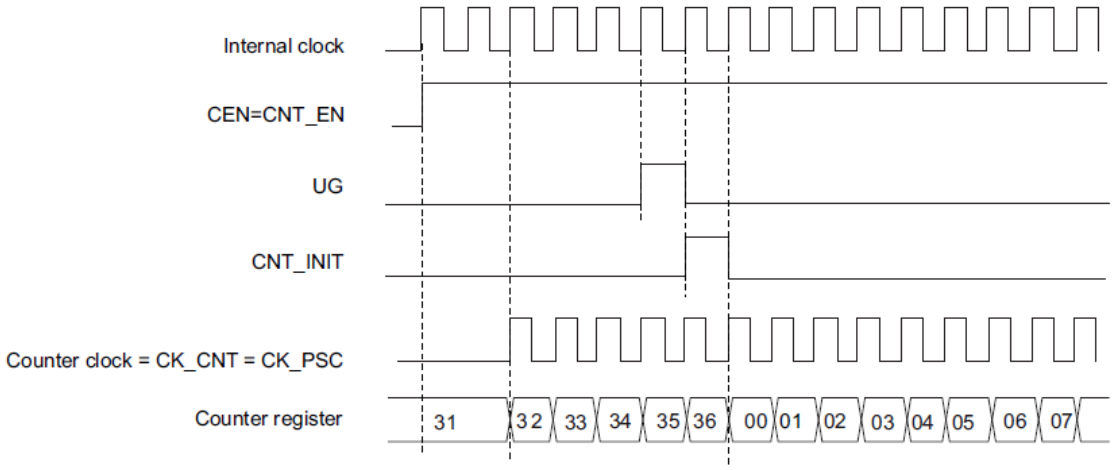


그림 3 Internal clock 사용시 동작

다음으로 external clock mode1을 사용하면 TI1또는 TI2핀으로 공급되는 외부 신호가 타이머의 clock신호가 된다. 이 모드를 사용하기 위해서는 Tix핀을 입력으로 사용하고, TIMx_SMCR의 TS비트를 통해서 Tix를 트리거 입력으로 선택하고, TIMx_SMCR의 SMS비트를 111로 설정하고, TIMx_CCER의 CCEx를 1로 설정하면 완료된다. 아래 [그림 4]는 external clock mode1 사용시 관련 신호들의 동작이다.

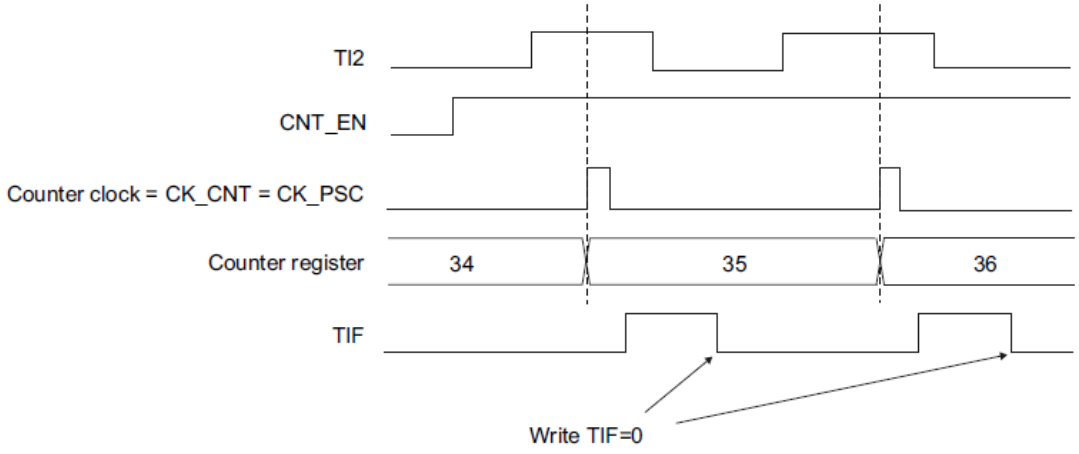


그림 4 External clock mode 1 사용시 동작

다음으로 external clock mode2를 사용하면 ETR핀이 타이머의 입력신호로 사용된다. 이 모드를 사용하기 위해서는 TIMx_SMCR의 ECE를 1로 설정하면 된다. 이때 필요에 따라 TIMx_SMCR의 설정을 통해 prescaler, filter, 신호의 극성을 설정할 수 있다. 아래 [그림 5]는 external clock mode2 사용시 관련 신호들의 동작이다.

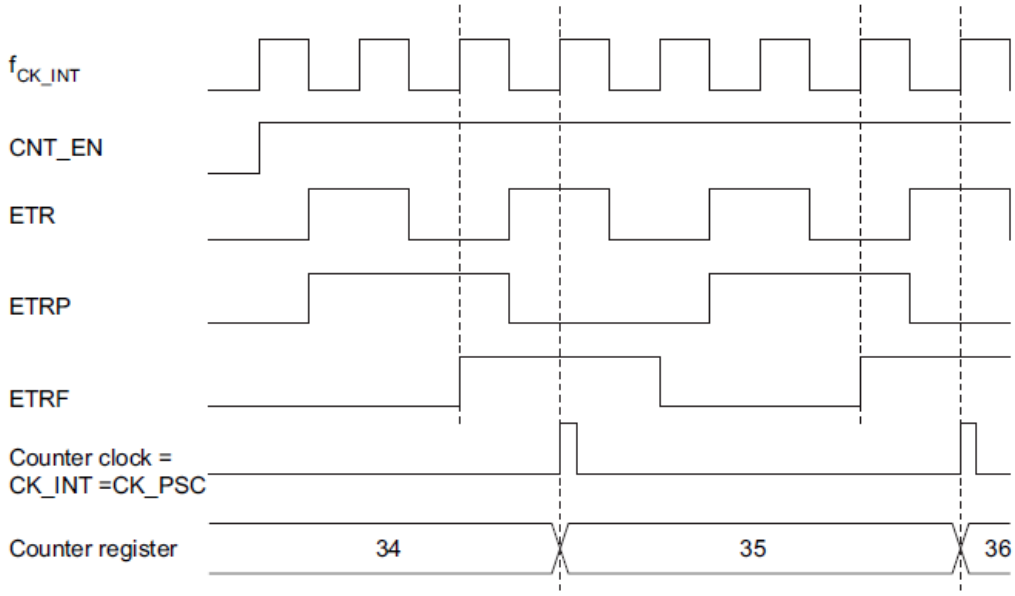


그림 5 External clock mode 2 사용시 동작

마지막으로 한 개의 타이머가 다른 타이머의 prescaler로 사용되기 위한 모드인 internal trigger inputs를 사용하기 위해서는 신호를 제공하는 타이머의 TIMx_CR2의 MMS비트를 010으로 하고, 제공받는 타이머의 TIMx_SMCR의 TS비트를 0으로 설정하고, SMS를 111로 설정하고, 두 타이머 모두 TIMx_CR1의 CEN비트를 1로 설정한다.

3) Time base generator

Time-base 유닛은 Counter register인 TIMx_CNT, Prescaler register인 TIMx_PSC, Auto-reload register인 TIMx_ARR로 이루어져 있다. TIMx_CR1의 CEN비트를 1로 설정하면 prescaler의 출력인 CK_CNT가 동작하게 된다. 또한 공급되는 clock 신호가 TIMx_PSC의 값에 따라 다음번 이벤트 업데이트 이후에 분주되어 사용된다. 마지막으로 TIMx_ARR는 이벤트 발생 후 타이머를 초기화 할 때 사용되는 값으로 TIMx_CR1의 ARPE레지스터의 값에 따라 이벤트 후에 사용될지 말지 결정된다. 이때 clock신호, prescaler, auto-reload 값을 통해서 이벤트의 발생 주기를 구할 수 있다. 먼저 우리가 사용하는 보드의 기본 TIM_CLK는 72Mhz이다. 이벤트의 주기를 구하는 식은 아래와 같다.

$$Update\_event = TIM_{CLK}/((PSC + 1) * (ARR + 1) * (RCR + 1))$$

4) Counter modes

카운터 모드에는 upcounter mode, downcounter mode, center-aligned mode가 존재한다. 먼저 upcounter mode는 0부터 시작해서 auto-reload된 값까지 숫자가 증가하다가 해당 값에 도달하면 overflow event를 발생시키며 다시 0부터 시작한다. 만약 event 발생을 원하지 않는다면 TIMx_CR1의 UDIS비트를 1로 설정한다. 또한 TIMx_CR1의 URS비트가 1이고 UG도 1이라면 인터럽트가 발생하지는 않고 이벤트만 발생한다. 아래 [그림 6]은 TIMx_ARR = 0x36일 때의 동작 예시이다.

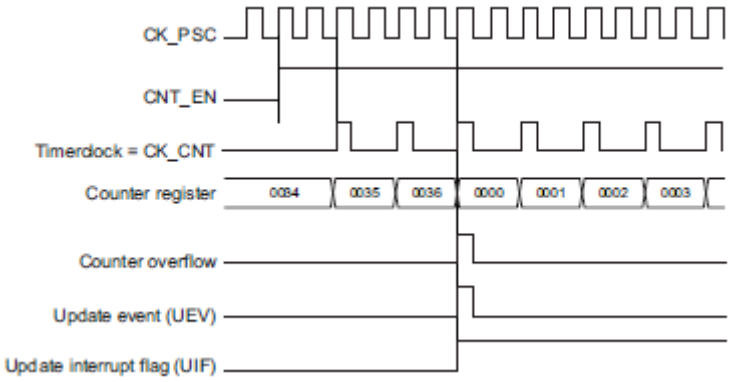


그림 6 Upcounter mode 동작 예시

다음으로 downcounter mode는 atuo-reload되는 값에서부터 0까지 숫자가 감소하다가 0이 되면 underflow even를 발생시키고 auto-reload값부터 다시 카운트를 시작한다. 나머지 동작은 upcounter mode와 비슷한다. 아래 [그림 7]은 TIMx_ARR=0x36일 때의 동작 예시이다.

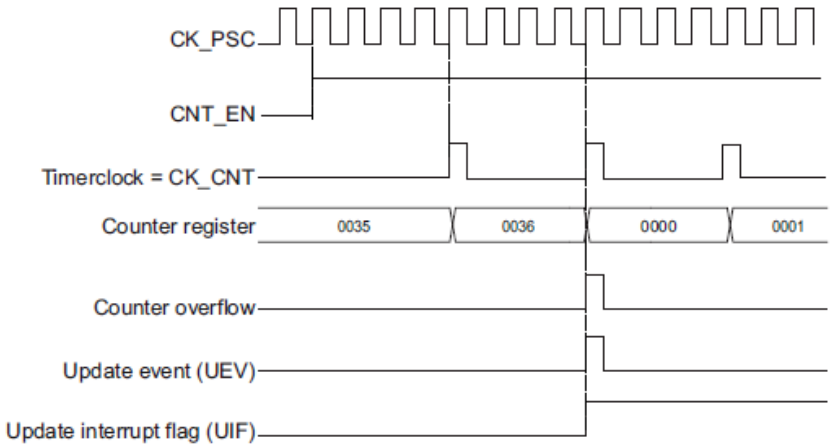


그림 7 Downcounter mode 동작 예시

마지막으로 center-aligned mode는 0에서부터 auto-reload되는 값 - 1까지 upcounting하고

overflow event를 발생시킨 후 다시 auto-reload된 값부터 1까지 다시 downcounting하고 underflow even를 발생시킨다. 이후 위 과정을 동일하게 반복한다. 이 모드를 위해서는 TIMx_CR1의 CMS비트를 01, 10, 11 중 하나로 설정해야 한다. 아래 [그림 8]은 TIMx_ARR=0x06일 때의 동작 예시이다.

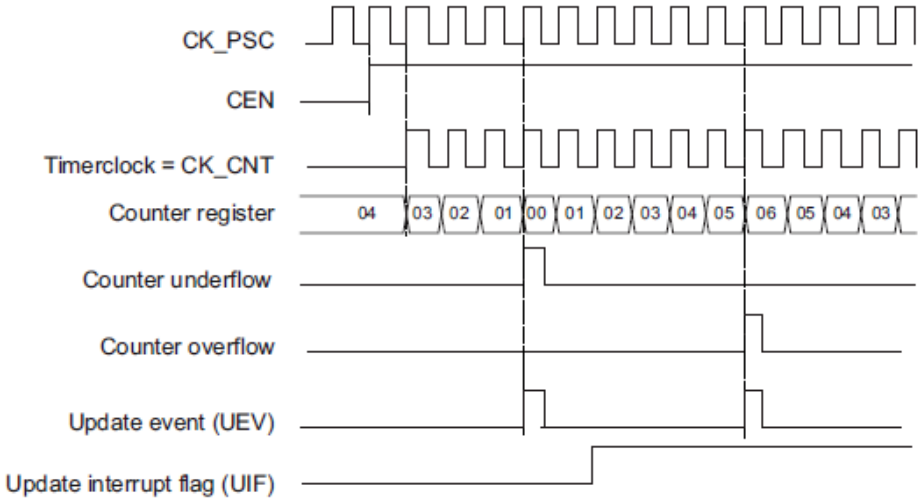


그림 8 Center-aligned mode 동작 예시

5) Capture/compare channels

각 타이머의 capture/compare channel은 digital filter, multiplexing, prescaler를 포함하는 input stage와 capture/compare 레지스터, comparator과 output control을 포함하는 output stage로 구성되어 있다. 각 stage의 구성은 아래 [그림 9], [그림 10], [그림 11]을 통해서 확인 가능하다.

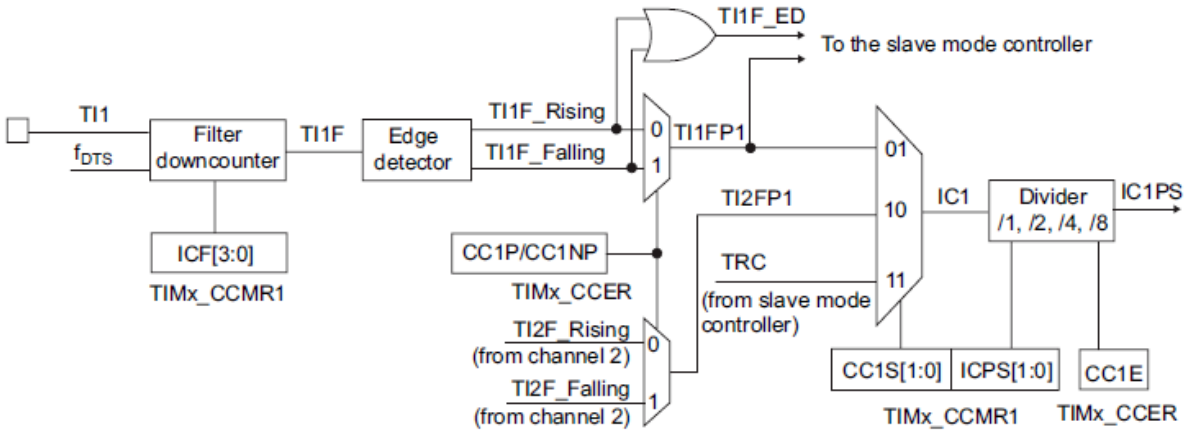


그림 9 Capture/compare channel input stage

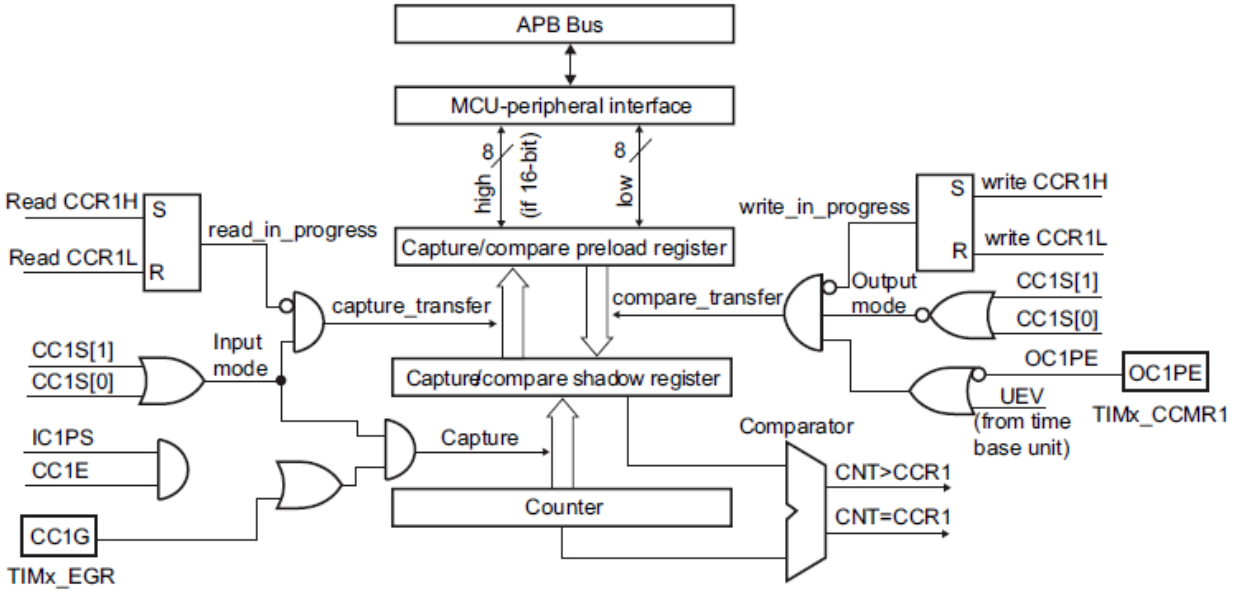


그림 10 Capture/compare channel main stage

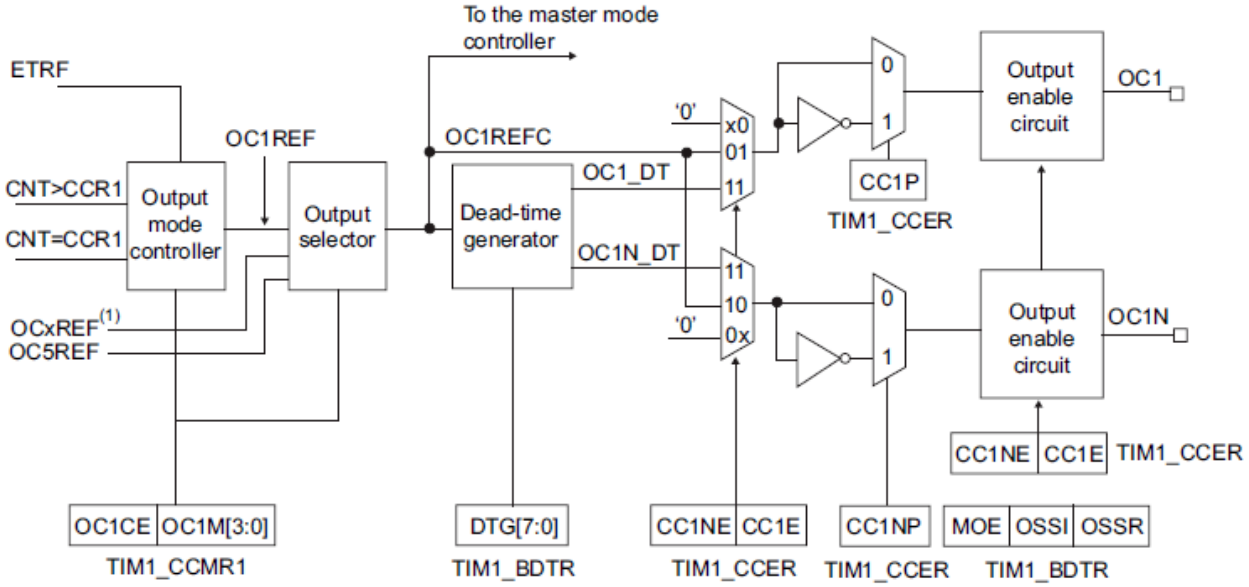


그림 11 Capture/compare channel output stage

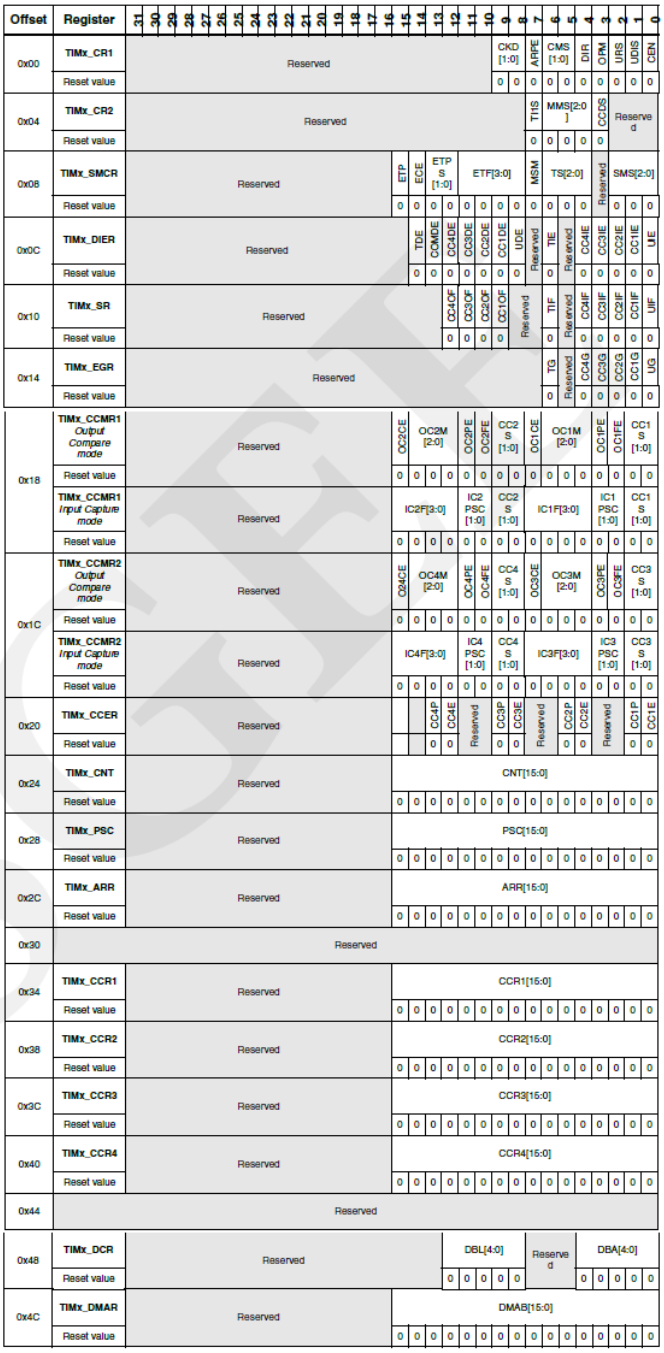
6) Some other modes

Input capture mode에서는 ICx신호의 변화가 감지된 이후의 카운터 값을 TIMx_CCRx에 저장한다. 그 이후 해당 CCxIF flag가 1로 set되며 인터럽트나 DMA신호가 발생할 수 있게 된다. 만약 capture가 발생하였을 때 이미 CCxIF flag가 1로 되어있었다면 CCxOF flag가 1로 set된다. 또한 이러한 flag들은 레지스터에 0을 저장해서 software적으로 reset해줄 수 있다.

다음으로 forced output mode가 존재하며 TIMx_CCMRx의 CCxS비트가 00인 아웃풋 모드일 때소

프트웨어에 의해 각각의 출력이 조절될 수 있는 모드이다. 또한 output compare mode가 존재하며 이는 일정 시간 경과 후 output waveform을 조절하거나 나타내기 위해 사용되는 모드이다. 마지막으로 one-pulse mode로 이는 input capture mode와 output compare 모드의 특정 경우이다. 사용자가 설정한 만큼의 딜레이와 폭으로 pulse를 생성할 수 있는 모드이다.

7) TIMx register map



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
0x00	TIMx_CR1	Reserved																						CKD [1:0]		ARPE	CMS [1:0]		DIR	OPM	URS	UDIS	CEN
	Reset value																							0	0	0	0	0	0	0	0	0	0
0x04	TIMx_CR2	Reserved																								TI1S	MMS[2:0]			CCDS	Reserved		
	Reset value																									0	0	0	0	0			
0x08	TIMx_SMCR	Reserved																ETP	ECE	ETPS [1:0]			ETF[3:0]			MSM	TS[2:0]			Reserved	SMS[2:0]		
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0
0x0C	TIMx_DIER	Reserved																	TDE	COMDE	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Reserved	TIE	Reserved	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	Reset value																		0	0	0	0	0	0	0		0		0	0	0	0	0
0x10	TIMx_SR	Reserved																			CC4OF	CC3OF	CC2OF	CC1OF	Reserved	TIF	Reserved	CC4IF	CC3IF	CC2IF	CC1IF	UIF	
	Reset value																				0	0	0	0		0		0	0	0	0	0	
0x14	TIMx_EGR	Reserved																								TG	Reserved	CC4G	CC3G	CC2G	CC1G	UG	
	Reset value																									0		0	0	0	0	0	
0x18	TIMx_CCMR1 *Output Compare mode*	Reserved																	OC2CE	OC2M [2:0]			OC2PE	OC2FE	CC2S [1:0]		OC1CE	OC1M [2:0]			OC1PE	OC1FE	CC1S [1:0]	
	Reset value																		0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
	TIMx_CCMR1 *Input Capture mode*	Reserved																	IC2F[3:0]				IC2PSC [1:0]		CC2S [1:0]		IC1F[3:0]				IC1PSC [1:0]		CC1S [1:0]	
	Reset value																		0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
0x1C	TIMx_CCMR2 *Output Compare mode*	Reserved																	OC4CE	OC4M [2:0]			OC4PE	OC4FE	CC4S [1:0]		OC3CE	OC3M [2:0]			OC3PE	OC3FE	CC3S [1:0]	
	Reset value																		0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
	TIMx_CCMR2 *Input Capture mode*	Reserved																	IC4F[3:0]				IC4PSC [1:0]		CC4S [1:0]		IC3F[3:0]				IC3PSC [1:0]		CC3S [1:0]	
	Reset value																		0	0	0	0	0	0	0		0	0	0	0	0	0	0	0
0x20	TIMx_CCER	Reserved																			CC4P	CC4E	Reserved		CC3P	CC3E	Reserved		CC2P	CC2E	Reserved		CC1P	CC1E
	Reset value																				0	0			0	0			0	0			0	0
0x24	TIMx_CNT	Reserved																	CNT[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x28	TIMx_PSC	Reserved																	PSC[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x2C	TIMx_ARR	Reserved																	ARR[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x30	Reserved																																
0x34	TIMx_CCR1	Reserved																	CCR1[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x38	TIMx_CCR2	Reserved																	CCR2[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x3C	TIMx_CCR3	Reserved																	CCR3[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x40	TIMx_CCR4	Reserved																	CCR4[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x44	Reserved																																
0x48	TIMx_DCR	Reserved																					DBL[4:0]			Reserved		DBA[4:0]					
	Reset value																					0	0	0	0	0		0	0	0	0	0	0
0x4C	TIMx_DMAR	Reserved																	DMAB[15:0]															
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

그림 12 TIMx 관련 레지스터 및 초기값

위 [그림 12]는 TIMx의 관련 레지스터 목록과 그 초기값이다. 이 레지스터들을 통해 TIMx의 동작을 설정하거나 원하는 값을 읽어들이고 수정할 수 있다.

3. 실험 과정

1) 실험 1

STEP 1: Program 8.2와 그림 8.7를 참조하여 Program 9.1의 내용을 분석한다.

이 과정에서 dot matrix display와 GPIO port와의 연결을 170페이지의 Step 8달리 다음 표와 같은 연결을 가정한다. Column에 연결되는 PB 신호들의 순서를 뒤집는 셈인데, 이진수를 표기할 때 일반적으로 우측에 LSB를 배치하는데 반해 dot matrix의 column 번호는 우측부터 번호가 부여되기 때문이다.

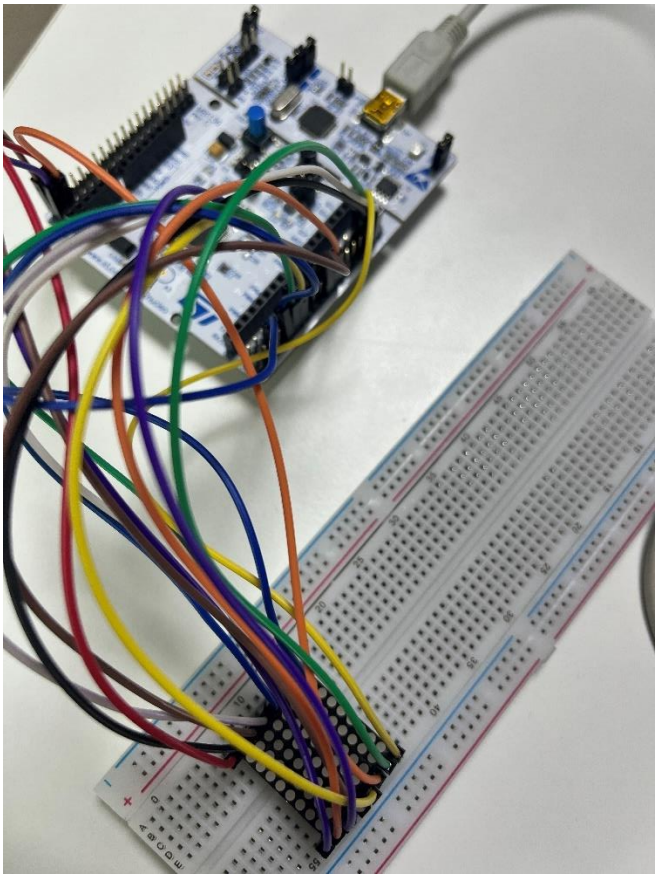


그림 13 새로 연결한 보드 모습

STEP 2: 인터럽트를 이용하여 dot matrix display를 구동하는 Program 9.1(lab91.c)를 이용하여 프로젝트를 생성한다.

```c
#include <stm32f10x.h>

u8 font8x8[16][8] = {
  {0x3c, 0x42, 0x46, 0x4a, 0x52, 0x62, 0x3c, 0x00}, //0
  {0x10, 0x30, 0x50, 0x10, 0x10, 0x10, 0x7c, 0x00}, //1
  {0x3c, 0x42, 0x02, 0x0c, 0x30, 0x42, 0x7e, 0x00}, //2
  {0x3c, 0x42, 0x02, 0x1c, 0x02, 0x42, 0x3c, 0x00}, //3
  {0x08, 0x18, 0x28, 0x48, 0xf3, 0x08, 0x1c, 0x00}, //4
  {0x7e, 0x40, 0x7c, 0x02, 0x02, 0x42, 0x3c, 0x00}, //5
  {0x1c, 0x20, 0x40, 0x7c, 0x42, 0x42, 0x3c, 0x00}, //6
  {0x7e, 0x42, 0x04, 0x08, 0x10, 0x10, 0x10, 0x00}, //7
  {0x3c, 0x42, 0x42, 0x3c, 0x42, 0x42, 0x3c, 0x00}, //8
  {0x3c, 0x42, 0x42, 0x3e, 0x02, 0x04, 0x38, 0x00}, //9
  {0x18, 0x24, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x00}, //A
  {0x7c, 0x22, 0x22, 0x3c, 0x22, 0x22, 0x7c, 0x00}, //B
  {0x1c, 0x22, 0x40, 0x40, 0x40, 0x22, 0x1c, 0x00}, //C
  {0x78, 0x24, 0x22, 0x22, 0x22, 0x24, 0x78, 0x00}, //D
  {0x7e, 0x22, 0x28, 0x38, 0x28, 0x22, 0x7e, 0x00}, //E
  {0x7e, 0x22, 0x28, 0x38, 0x28, 0x20, 0x70, 0x00}, //F
};
u8 data = 0xA;
u32 row, col, i;

  int main(void) {

  RCC->APB2ENR = 0x0000081D;
  GPIOC->CRL = 0x33333333;
  GPIOB->CRH = 0x33333333;

  TIM1->CR1 = 0x00;
  TIM1->CR2 = 0x00;
  TIM1->PSC = 0x07FF;
  TIM1->ARR = 0x2000;

  TIM1->DIER = 0x0001;
  NVIC->ISER[0] = 0x02000000;
  TIM1->CR1 |= 0x0001;

  row = 1;
  col = 0;
  i = 0;

  while(1){ ;}
} //end main

void TIM1_UP_IRQHandler(void) {
  if((TIM1->SR & 0x0001) != 0) {
    GPIOC->ODR = ~row;
    row = row << 1;
    col = font8x8[data][i];
    GPIOB->ODR = col << 8;
    i++;
    if(row == 0x100) {row = 1; i = 0;}
    TIM1->SR &= ~(1 << 0); //clear UIF
  }
}
```

그림 14 직접 작성한 lab9_1.c

STEP 3: Lines 3-20에서 선언된 font8x8[][]의 내용을 그림 8.7의 dot matrix에 표시될 수 있는 숫자들의 모양과 연관지어 해석해본다. 이 내용을 table lookup에 근거해서 설명해보자.

fontx8x8[][]의 내용은 dot matrix에서 각 row에서 켜야할 col에 대한 정보이다. 예를 들어 0에 대한 데이터를 살펴보면 0x3c, 0x42, 0x46, 0x4a, 0x52, 0x62, 0x3c, 0x00이고 이를 그려보면 아래 [표 1]과 같이 나타낼 수 있다. 이와 같이 0부터 F까지 dot matrix에서 어떻게 LED를 켜야 해당 문자처럼 보일지를 미리 table을 만든 것임을 알 수 있다. 나중에 프로그램에서 문자를 출력하고자 할 때 우리가 lines 3-20에서 선언한 배열을 참조하는 table lookup을 통해서 간편하게 문자를 dot matrix에 그릴 수 있다.

0x3C	0	0	1	1	1	1	0	0
0x42	0	1	0	0	0	0	1	0
0x46	0	1	0	0	0	1	1	0
0x4A	0	1	0	0	1	0	1	0
0x52	0	1	0	1	0	0	1	0
0x62	0	1	1	0	0	0	1	0
0x3C	0	0	1	1	1	1	0	0
0x00	0	0	0	0	0	0	0	0

그림 15 0에 대한 배열의 내용 풀이

STEP 4: Lines 26-28의 초기화 과정의 내용을 확인한다.

Lines 26-28의 코드를 한 줄 한 줄 해석하면 다음과 같다. 먼저 line 26의 RCC->APB2ENR = 0x0000081D; 는 TIM1, GPIOC, GPIOB, GPIOA, AFIO의 데이터가 APB2 버스를 통해서 이동할 수 있도록 enable해준 것이다. 다음으로 line 27의 GPIOC->CRL = 0x33333333; 은 PC0 – PC7핀을 push pull output으로 사용하도록 설정한 것이다. 마지막으로 line 28의 GPIOB->CRH = 0x33333333; 은 PB8 – PB15핀을 push pull output으로 사용하도록 설정한 것이다. 위의 3줄의 코드를 통해서 프로그램에서 사용할 장치를 APB2 bus에 연결해주고 dot matrix에 연결된 핀을 push pull output으로 연결해주었다.

STEP 5: Lines 48-53에서 수행하는 동작의 정체를 파악해보자. 이 과정에서 사용되는 row, col, i는 lines 39-41에서와 같이 초기화 된다. TIM1_UP_IRQHandler service routine이 수행될 때마다 dot matrix display에는 어떤 내용이 표시될지 알아보자.

Lines 48-53의 코드를 통해서 dot matrix에 font8x8[][]에 저장된 데이터를 불러와서 data에 해당하는 문자를 dot matrix에서 각 row별로 출력하는 동작을 수행한다. 하나씩 코드를 뜯어보면 다음과 같다. 먼저 line 48의 GPIOC->ODR = ~row; 를 통해서 현재의 row에 해당하는 핀에만 0을 출력하고 나머지 row에 연결된 핀에는 1을 출력한다. 그리고 line 49의 row = row << 1; 을 통해서 row의 비트를 한 칸 왼쪽으로 shift해서 다음 row를 선택한다. Line 50의 col = font8x8[data][i]; 는 font8x8에서 현재 출력하고자 하는 data와 몇 번째 row인지를 자연수로 저장하고 있는 i를 참고하여 현재 data와 i에 출력해야 할 col정보를 font8x8에서 table lookup을 통해서 가져와 저장한다. 다음으로 line 51의 GPIOB->ODR = col << 8; 을 통해서 line 50에서 가져온 col정보를 dot matrix의 col관련 핀에 저장해준다. 이때 col을 왼쪽으로 8칸 shift해서 저장하여 PB8 – PB15에 저장해준다. 다음으로 line 52의 i++; 을 통해서 col을 가져올 때 필요한 i를 하나 증가시켜준다. 마지막으로 line 53의 if(row == 0x100) {row = 1; i = 0}; 을 통해서 line 49에서 row를 증가하고 나서 만약 0x100이 되어 8번 row를 넘어가게 되면 row와 i를 모두 초기화해준다. 즉 요약하면 row를 바탕으로 dot matrix의 row핀에 output을 출력하고 i를 이용해 font8x8에서 table lookup을 통해 원하는 데이터를 col로 가져와 dot matrix의 col핀에 output을 출력하고 row와 i를 업데이트 하는 동작을 수행한다. 이를 통해 dot matrix의 row에 font8x8에 저장된 내용이 출력된다.

STEP 6: 다음 사항들에 대해 참고문헌 (STMicroelectronics, 2011)을 참조하면서 프로그램에 포함된 내용을 파악한다.

● **Line 26에는 APB2 bus에 연결된 TIM1을 사용하기 위해 clock을 enable하기 위한 조치가 포함되었다. 표 9.1을 통해 그 내용을 파악한다.**

● **Line 30-31은 TIM1의 counter register를 설정하는 과정이다. 프로세서가 제공하는 internal clock을 기본적인 방법으로 사용할 경우 별도의 설정이 필요하지 않다. TIM1과 연관된 다른 레지스터들의 초기화를 마친 후 line 37에서 볼 수 있는 것처럼 CEN bit를 1로 설정하여 TIM1을**

enable 시킨다.

● Lines 32-33은 counter에 공급되는 clock 신호의 prescaler에 저장될 내용과 auto-reload 레지스터에 저장될 내용을 각각 지정한다. 182페이지의 관련식에 의해 counter가 update되는 빈도가 결정된다.

● Line 35-36은 TIM1의 인터럽트 요청을 위한 설정이다. Counter에 update가 일어날 경우에만 인터럽트가 발생하도록 설정한다. 또한 NVIC_ISER0에 TIM1의 update에 의한 인터럽트를 설정하는데, 표 7.2에 TIM1 Update interrupt에 해당한다. 표 7.5의 NVIC_ISER0에서 41번 인터럽트를 사용하도록 설정한다.

● Line 43은 무한 loop을 구성하고 있으나 loop 내부에서는 아무런 일도 수행되지 않음(단순 무한 loop)에 주목한다.

● Lines 46-56은 TIM1의 counter update에 의해 발생하는 인터럽트의 service routine으로서 TIM1에 의해 인터럽트가 발생할 때마다 수행된다. Line 47을 통해 수행하는 확인의 정체는 line 35에 의해 설정된 인터럽트만을 처리하기 위함이고, line 52는 TIM1에 의해 발생한 인터럽트 요청을 해제하기 위함이다.

Line 26의 코드는 RCC->APB2ENR = 0x0000081D; 는 STEP 4에서도 언급하였듯 TIM1을 APB2 bus에 연결하는 동작을 수행한다. 교재의 표 9.1을 보면 RCC->APB2ENR레지스터의 11번 bit는 TIM1EN이다. 즉 0x0000081D를 통해서 11번 bit를 1로 set하였음으로 APB2 bus에 연결된 TIM1을 사용하기 위해 clock을 enable하는 과정이다.

Line 30-31 코드는 각각 TIM1->CR1 = 0x00; , TIM1->CR2 = 0x00; 으로 [그림 12]에서 확인할 수 있듯이 원래의 초기값에서 어떠한 변화도 주지 않는다. 이렇게 기본설정의 경우의 동작을 reference manual을 통해 살펴보면 다음과 같다. CK_INT를 clock division없이 그대로 사용하며, TIMx_ARR의 auto-reload값이 buffer없이 그대로 적용되며, 일반적인 카운터가 증가하는 edge-aligned mode와 upcounter mode로 동작하며, update event에서 카운터가 멈추지 않으며 모든 경우에서 upade interrupt 또는 DMA 요청이 발생할 수 있으며, EUV가 enalbe되어있어 있다. 그리고 CEN bit가 0이기 때문에 아직은 counter가 disable되어 있다. 또한 각 output이 idle state이며 TIM1_CH1핀이 TI1의 입력으로 연결되어 있으며 TIM1_EGR레지스터의 UG bit가 trigger output으로 사용되고, CC1 event 발생시 CC1 DMA 요청이 보내지는 가장 기본적인 카운터로 설정되어 있

다. 그리고 line 37의 TIM1->CR1 |= 0x0001; 을 통해 그 이전까지의 CR1값을 유지하며 CR1의 0번 bit인 CEN을 1로 set해서 TIM1을 enable해준다.

Line 32-33의 TIM1->PSC = 0x07FF; , TIM1->ARR = 0x2000; 의 코드를 해석하면 다음과 같다. TIM1의 prescaler 값을 0x07FF로 설정하고 atuo-reload 값으로 0x2000으로 설정하였다. 그리고 이때 update 빈도를 구하기 위한 RCR값은 프로그램에서 건드리지 않았음으로 기본적인 0을 가지고 있다. 이 값을 바탕으로 TIM1 counter의 update 빈도를 구하면 72Mhz/((0x7FF + 1)*(0x2000 + 1)*(0 + 1)) = 4.29hz가 나오게 된다. 즉 update 빈도는 1초에 4.29번이다.

Line 35-36의 TIM1->DIER = 0x0001; , NVIC->ISER[0] = 0x02000000; 의 코드를 해석하면 다음과 같다. 먼저 DIER레지스터는 DMA/interrupt enable register로 0번 bit인 UIE를 1로 set하여 update interrupt만을 enable해주었다. 그리고 NVIC->ISER[0]의 25번 bit를 1로 set하여 41번 인터럽트인 TIM1_UP 인터럽트의 enable해주었다. 이를 통해서 TIM1의 update interrupt가 발생하였을 때 핸들러가 호출될 수 있도록 설정해주었다.

Line 43의 while(1) { ;} 코드는 프로그램에서 무한 루프를 돌고 있다. 루프만 내부에서는 아무 일도 하지 않고 계속해서 루프만 돌게 되는데 이는 더 이상 프로그램이 진행되지 못하게 막는 역할을 수행한다. 계속해서 무한 루프를 돌다가 인터럽트가 발생하면 인터럽트 핸들러를 수행하고 다시 돌아와서 계속해서 무한 루프를 도는 동작을 수행한다.

마지막으로 line 46-56의 코드는 TIM1_UP_IRQHandler함수의 내용으로 TIM1의 update interrupt가 발생하였을 때 수행하는 인터럽트 핸들러의 내용이다. Lines 48-53은 STEP5에서 확인하였듯 dot matrix에 LED를 켜는 과정이다. 그 외에 line 47의 if((TIM1_SR & 0x0001) != 0)을 통해서 인터럽트 핸들러로 들어왔을 때 TIM1의 SR레지서트의 0번 bit인 UIF를 확인해서 update interrupt가 pending되어 있는 것이 맞는지 확인한다. 확인 결과 update interrupt가 pending되어 있다면 dot matrix에 LED를 켜는 코드를 수행한다. 그리고 마지막에 line 54의 TIM1->SR &= ~(1 << 0); 을 통해서 TIM1의 0번 bit인 UIF에 1을 써서 update interrupt의 pending상태를 해제한다. 이러한 과정이 필요한 것은 NVIC의 interrupt의 pending이 해제되는 시간보다 빨리 인터럽트 핸들러를 종료하고 다시 인터럽트로 들어올 수 있기 때문에 원하는 인터럽트가 pending되어 있는지 확인하고 인터럽트 종료 후에는 pending을 풀어주는 과정이 필요하다.

STEP 7: 이 프로그램을 수행하면서 dot matrix display에 표시되는 내용을 확인한다. MDK의 Peripherals-System Viewer-TIM-TIM1 메뉴를 통해 창을 열고 해당 counter의 동작을 관찰해보자.

Lines 32-33의 내용을 변경하면서 Prescaler와 auto-reload 레지스터의 역할을 이해한다.

프로그램의 수행결과를 TIM1, 인터럽트, GPIO의 동작과 연관지어 설명해보자.

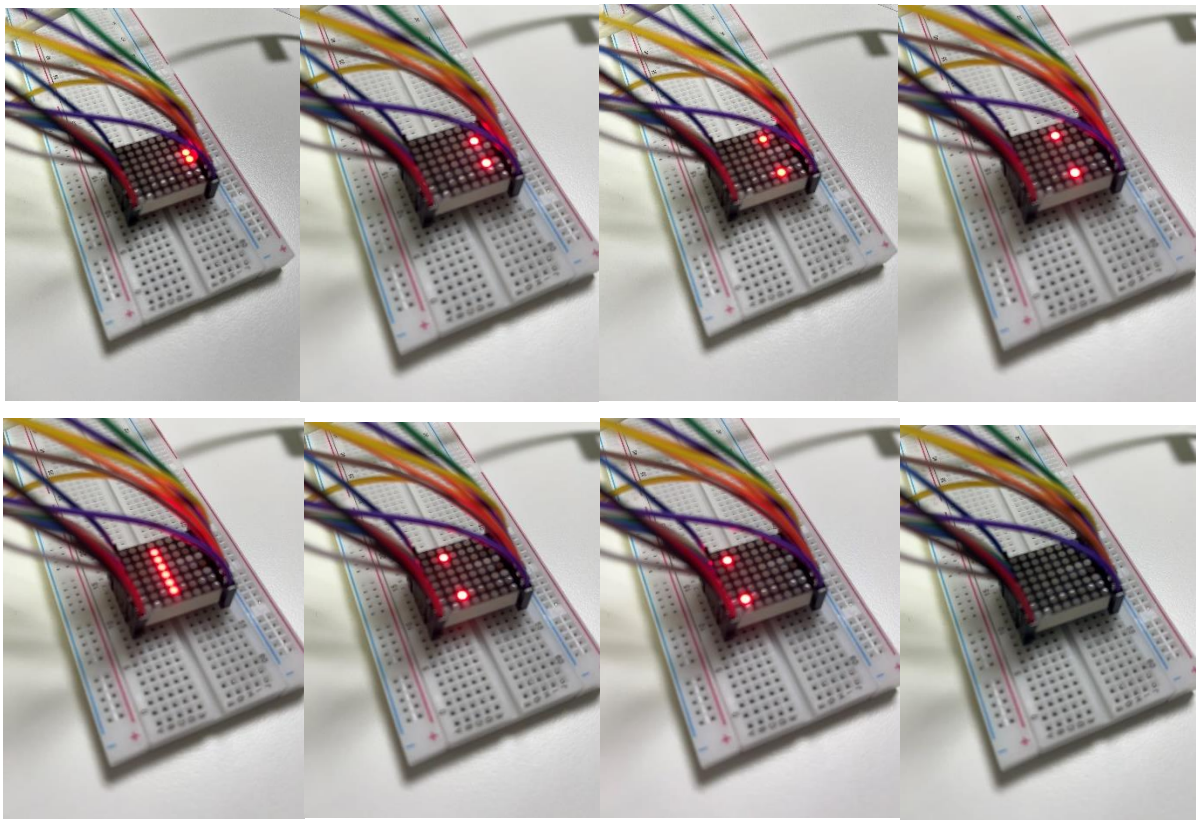
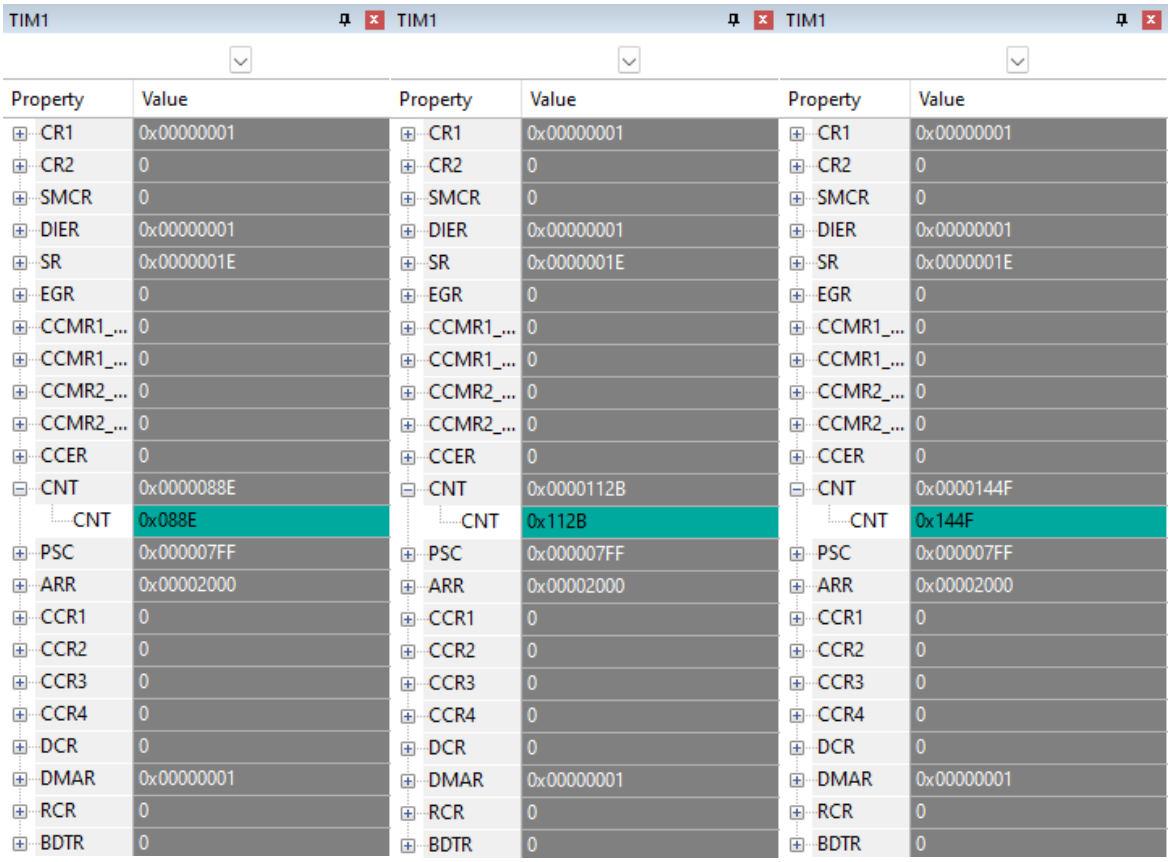


그림 16 lab9_1 동작 사진

우선 위 [그림 16]을 통해서 프로그램을 보드에서 수행시켰을 때의 동작을 차례대로 볼 수 있다. 한 번에 dot matrix에 A자 모양이 나오는 것이 아니라 한 row씩 움직이면서 A자의 모양대로 LED가 켜지는 것을 알 수 있다. 또한 디버거에서 TIM1의 정보를 보면 아래 [그림 17]처럼 CNT이 계속해서 바뀌고 있는 것을 알 수 있다.



TIM1

Property	Value
CR1	0x00000001
CR2	0
SMCR	0
DIER	0x00000001
SR	0x0000001E
EGR	0
CCMR1_...	0
CCMR1_...	0
CCMR2_...	0
CCMR2_...	0
CCER	0
CNT	0x0000088E
CNT	0x088E
PSC	0x000007FF
ARR	0x00002000
CCR1	0
CCR2	0
CCR3	0
CCR4	0
DCR	0
DMAR	0x00000001
RCR	0
BDTR	0

TIM1

Property	Value
CR1	0x00000001
CR2	0
SMCR	0
DIER	0x00000001
SR	0x0000001E
EGR	0
CCMR1_...	0
CCMR1_...	0
CCMR2_...	0
CCMR2_...	0
CCER	0
CNT	0x0000112B
CNT	0x112B
PSC	0x000007FF
ARR	0x00002000
CCR1	0
CCR2	0
CCR3	0
CCR4	0
DCR	0
DMAR	0x00000001
RCR	0
BDTR	0

TIM1

Property	Value
CR1	0x00000001
CR2	0
SMCR	0
DIER	0x00000001
SR	0x0000001E
EGR	0
CCMR1_...	0
CCMR1_...	0
CCMR2_...	0
CCMR2_...	0
CCER	0
CNT	0x0000144F
CNT	0x144F
PSC	0x000007FF
ARR	0x00002000
CCR1	0
CCR2	0
CCR3	0
CCR4	0
DCR	0
DMAR	0x00000001
RCR	0
BDTR	0

그림 17 TIM1의 레지스터 값

그리고 prescaler와 auto-reload 레지스터의 역할을 알아보기 위해서 line 32와 line 33에서 각각 할당하는 값을 바꿔보았다. 먼저 line 32에서 0x07FF대신 0x0001을 넣은 경우는 타이머의 event가 매우 빨리 발생하며 카운터가 동일하게 0x2000의 값까지 카운팅 되는 것을 디버거 모드에서 확인할 수 있었다. 즉, prescaler 레지스터에 저장된 값은 clock을 몇 분주 할것인지에 대한 값임을 알 수 있다. 또한 line 33에서 0x2000대신 0xF000을 넣은 경우 타이머의 event가 매우 느린 속도로 발생하였다. 그리고 디버거 모드로 타이머 레지스터의 CNT값을 살펴본 결과 최대 0x2000을 지나 0xF000까지 도달하는 것을 알 수 있었다. 즉 auto-reload 레지스터의 역할은 카운터에서 카운팅을 어디까지 할 것인지를 정해주는 역할을 수행한다.

이번 프로그램의 동작은 TIM1 타이머가 upcounting 하다가 auto-reload된 값까지 카운팅을 완료하면 update interrupt를 발생시키고 다시 카운팅을 시작한다. Update interrupt 발생시 인터럽트 핸들러 내부에서 dot matrix와 연결된 GPIOC와 GPIOB의 출력을 조절하여 dot matrix가 어떻게 켜질 것인지 컨트롤 하게 된다. 그 결과 prescaler와 auto-reload값에 의해 TIM1의 update interrupt가 발생하는 주기에 따라서 dot matrix에 한 row씩 미리 정해둔 문자 A가 출력된다.

STEP 8: (구현과제) Step 1에서와 같이 연결을 변경하지 않고(즉, 170페이지의 Step 8의 연결을 유지하고) 동작할 수 있도록 font8x8[][]의 내용을 변경해보자.

9주차 실험의 STEP1에서 8주차 실험과는 다르게 col을 역방향으로 연결하였다. 따라서 [그림 18]의 좌측처럼 dot matrix에서 문자가 좌우대칭되어 보인다. 이를 해결하기 위해서는 간단히 font8x8[][]의 값에서 bit값을 좌우대칭 하면 된다. 예를 들어 0x7C = 2b01111100를 좌우대칭 하면 2b001111100 = 0x3E가 되도록 모든 데이터를 수정해야 한다. 즉 font8x8[][]의 내용을 아래 코드와 같이 수정하면 [그림 18]의 우측과 같이 정상적인 출력이 나오도록 할 수 있다.

```
u8 font8x8[16][8] = {
   {0x3C, 0x42, 0x62, 0x58, 0x4A, 0x46, 0x3C, 0x00}, //0
   {0x08, 0x0C, 0x0A, 0x08, 0x08, 0x08, 0x3E, 0x00}, //1
   {0x3C, 0x42, 0x40, 0x30, 0x0C, 0x42, 0x7E, 0x00}, //2
   {0x3C, 0x42, 0x40, 0x38, 0x40, 0x42, 0x3C, 0x00}, //3
   {0x10, 0x18, 0x24, 0x12, 0xCF, 0x10, 0x38, 0x00}, //4
   {0x7E, 0x02, 0x3E, 0x40, 0x40, 0x42, 0x3C, 0x00}, //5
   {0x38, 0x04, 0x02, 0x3E, 0x42, 0x42, 0x3c, 0x00}, //6
   {0x7E, 0x42, 0x20, 0x10, 0x08, 0x08, 0x08, 0x00}, //7
   {0x3c, 0x42, 0x42, 0x3c, 0x42, 0x42, 0x3c, 0x00}, //8
   {0x3c, 0x42, 0x42, 0x7C, 0x40, 0x20, 0x1C, 0x00}, //9
   {0x18, 0x24, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x00}, //A
   {0x7C, 0x22, 0x22, 0x3c, 0x22, 0x22, 0x7C, 0x00}, //B
   {0x38, 0x44, 0x02, 0x02, 0x02, 0x44, 0x38, 0x00}, //C
   {0x1E, 0x24, 0x44, 0x44, 0x44, 0x24, 0x1E, 0x00}, //D
   {0x7e, 0x44, 0x14, 0x1C, 0x14, 0x44, 0x7e, 0x00}, //E
   {0x7e, 0x44, 0x14, 0x1C, 0x14, 0x04, 0x0E, 0x00}, //F
};
```

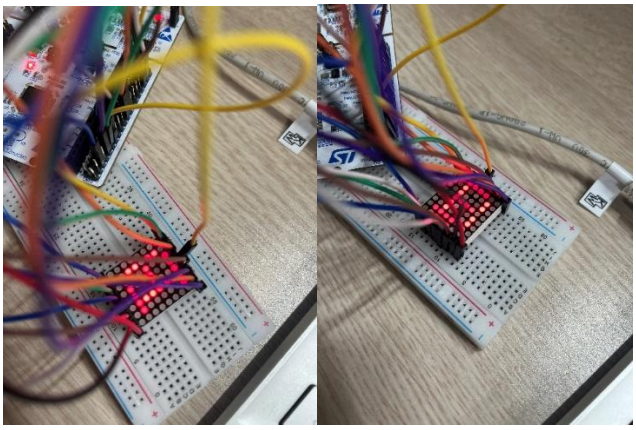


그림 18 코드 수정 전/후 출력

STEP 9: (구현과제) Program 8.3을 통해 이해한 내용을 바탕으로 key matrix의 scan이 TIM3에 의한 인터럽트 주기에 맞춰서 진행될 수 있도록 interrupt service routine으로 작성하여 Program 9.1에 추가해보자.

Program 9.1에서 display interval을 timer의 인터럽트에 의존하면서 CPU의 부담을 덜 수 있었던 것처럼 keypad의 작동을 위해 필요한 scan 과정을 어떻게 interrupt service routine에 수용할 수 있을지 생각해보자. 결과적으로 keypad에 속한 스위치를 누를 때마다 그에 상응하는 수를 dot matrix display에 표시하는 프로그램을 구현해보자.

```
#include <stm32f10x.h>


u8 font8x8[16][8] = {
   {0x3c, 0x42, 0x46, 0x4a, 0x52, 0x62, 0x3c, 0x00}, //0
   {0x10, 0x30, 0x50, 0x10, 0x10, 0x10, 0x7c, 0x00}, //1
   {0x3c, 0x42, 0x02, 0x0c, 0x30, 0x42, 0x7e, 0x00}, //2
   {0x3c, 0x42, 0x02, 0x1c, 0x02, 0x42, 0x3c, 0x00}, //3
   {0x08, 0x18, 0x28, 0x48, 0xf3, 0x08, 0x1c, 0x00}, //4
   {0x7e, 0x40, 0x7c, 0x02, 0x02, 0x42, 0x3c, 0x00}, //5
   {0x1c, 0x20, 0x40, 0x7c, 0x42, 0x42, 0x3c, 0x00}, //6
   {0x7e, 0x42, 0x04, 0x08, 0x10, 0x10, 0x10, 0x00}, //7
```

```
  {0x3c, 0x42, 0x42, 0x3c, 0x42, 0x42, 0x3c, 0x00}, //8
  {0x3c, 0x42, 0x42, 0x3e, 0x02, 0x04, 0x38, 0x00}, //9
  {0x18, 0x24, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x00}, //A
  {0x7c, 0x22, 0x22, 0x3c, 0x22, 0x22, 0x7c, 0x00}, //B
  {0x1c, 0x22, 0x40, 0x40, 0x40, 0x22, 0x1c, 0x00}, //C
  {0x78, 0x24, 0x22, 0x22, 0x22, 0x24, 0x78, 0x00}, //D
  {0x7e, 0x22, 0x28, 0x38, 0x28, 0x22, 0x7e, 0x00}, //E
  {0x7e, 0x22, 0x28, 0x38, 0x28, 0x20, 0x70, 0x00}, //F
};
u8 data = 0xA;
u32 row, col, i, j, key_index, key_row, key_col, col_scan;

        int main(void) {

  RCC->APB2ENR = 0x0000081D;
  RCC->APB1ENR = 0x00000002;
  GPIOC->CRH = 0x00003333;
  GPIOA->CRH = 0x00008888;
  GPIOA->ODR = 0x0F00;
  GPIOC->CRL = 0x33333333;
  GPIOB->CRH = 0x33333333;

  TIM1->CR1 = 0x00;
  TIM1->CR2 = 0x00;
  TIM1->PSC = 0x0001;
  TIM1->ARR = 0x2000;
```

```
   TIM3->CR1 = 0x00;
   TIM3->CR2 = 0x00;
   TIM3->PSC = 0x07FF;
   TIM3->ARR = 0x0100;

   TIM1->DIER = 0x0001;
   TIM3->DIER = 0x0001;
   NVIC->ISER[0] = 0x22000000;
   TIM1->CR1 |= 0x0001;
   TIM3->CR1 |= 0x0001;

   row = 1;
   col = 0;
   i = 0;
   j = 0;
   key_index = 0;
   key_row = 0x01;

   while(1){ ;}
} //end main

void TIM1_UP_IRQHandler(void) {
   if((TIM1->SR & 0x0001) != 0) {
      GPIOC->ODR = ~row;
      row = row << 1;
```

```
    col = font8x8[data][i];
    GPIOB->ODR = col << 8;
    i++;
    if(row == 0x100) {row = 1; i = 0;}
    TIM1->SR &= ~(1 << 0); //clear UIF
  }
}

void TIM3_IRQHandler(void) {
        if((TIM3->SR & 0x0001) != 0) {
    GPIOC->BSRR = (~(key_row << 8) & 0x0F00) | (key_row << 24);
    key_col = GPIOA->IDR;
    key_col = (key_col >> 8) & 0x0F;
    col_scan = 0x01;
    for (j = 0; j < 4; j++) {
      if((key_col & col_scan) == 0)
        data = key_index;
      col_scan = col_scan << 1;
      key_index = key_index + 1;
    }
    key_row = key_row << 1;
    if (key_row == 0x10) {
      key_row = 0x01;
      key_index = 0;
    }
```

```
                TIM3->SR &= ~(1 << 0);

        }

}
```

위 코드와 같이 keypad를 보드에 연결하고 초기화해준 뒤 TIM3에 관해 초기화 하고 TIM3에 대한 인터럽트 핸들러를 만들어 keypad를 주기적으로 스캔할 수 있도록 해주었다. 먼저 line 27에서 TIM3가 연결되어 있는 APB1 bus에 TIM3를 enable해주고 lines 28-30을 통해 8주차 실험처럼 GPIOC와 GPIOA를 keypad에 연결할 수 있도록 input, output을 설정하였다. 그리고 lines 39-42에서 TIM3에 대한 설정을 해주고, line 45,46,48을 통해서 TIM3의 인터럽트가 발생할 수 있도록 설정을 해주었다. 그리고 추가적으로 문자가 dot matrix에 표현되는 것을 한눈에 보기 위해서 line 36의 TIM1->PSC = 0x0001로 설정하여 update 빈도를 빠르게 설정해주었다. 그 결과 TIM3에서 update interrupt가 발생할 때 마다 keypad가 눌렸는지 확인하도록 하여 계속해서 루프를 돌면서 keypad를 확인하지 않고 인터럽트가 발생할 때 마다 keypad가 눌렸는지 확인하여 cpu의 부담을 줄여줄 수 있었다. 이 프로그램에 대한 출력 결과는 아래 [그림 19]와 같이 나타난다.

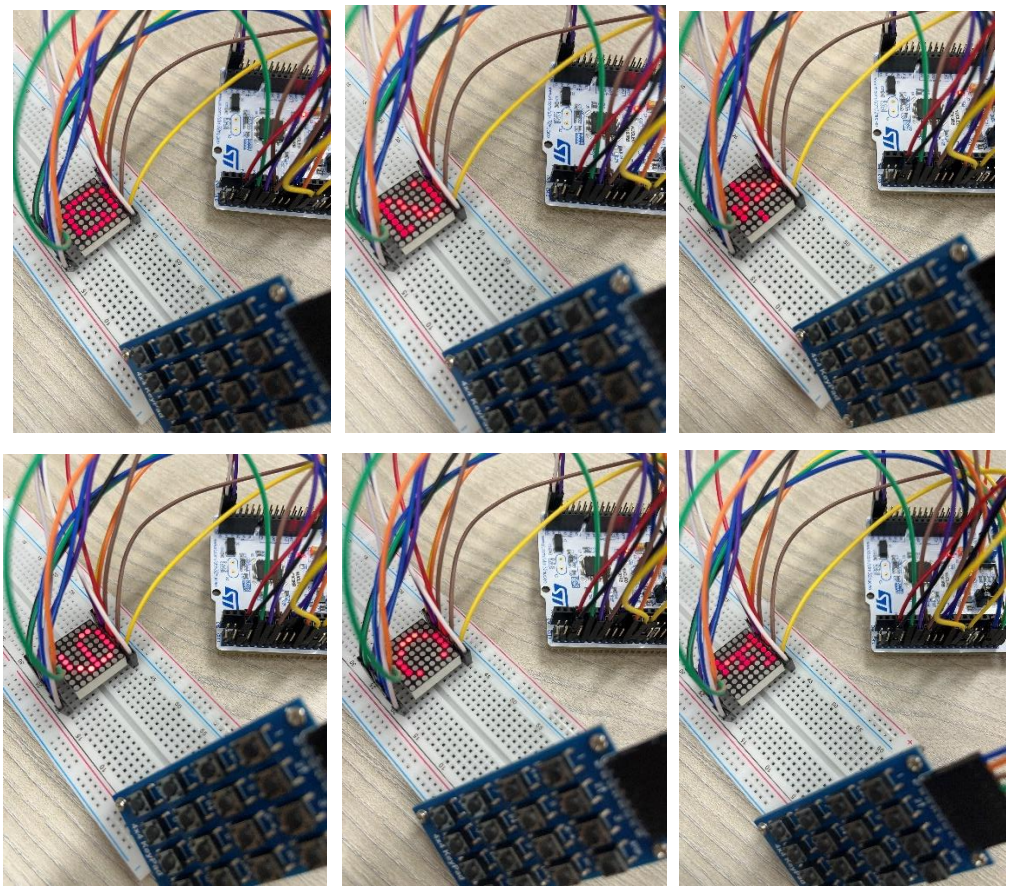


그림 19 STEP9 프로그램 수행 결과 예시

2) 실험 2

STEP 10: 이전 프로그램의 분석 및 수행경험을 바탕으로 Program 9.2(lab92.c)를 분석한다. 이 프로그램에서는 TIM2를 사용한다.

이 과정에서 9.2.8절의 reset mode에 대한 설명과 그림 9.14(a)를 참조한다.

Lines 6-7의 설정은 앞서 설명한 LED와 S0 스위치의 연결과 관련된다.

Line 10과 line 14는 이전 프로그램에서와 다르게 지정되는 것을 볼 수 있는데, 그 의미를 (STMicroelectronics, 2011)을 참조해서 확인해 보자.

이 프로그램에서는 line 16에서 볼 수 있는 것처럼 TIMx_SMCR을 사용하는데 프로그램에서 제시된 설정에 대한 의미를 (STMicroelectronics, 2011)를 통해 확인해보자. 또한 line 14에서 볼 수 있는 것처럼, overflow update에 의한 인터럽트(UIF)뿐만 아니라, trigeer에 의한 인터럽트(TIF)도 사용하도록 설정되었음도 확인한다.

Lines 6의 GPIOA->CRL = 0x00300008; 과 line 7의 GPIOA->ODR = 0x01; 을 통해서 PA0는 input pull-up으로 설정하고 PA5를 push-pull output으로 설정하였다. 그리고 PA0는 keypad의 K4와 연결하였고 PA5는 보드의 LD2와 내부적으로 연결되어 있다. 그리고 아래 [그림 20]과 같이 keypad와 보드를 연결해주었다.

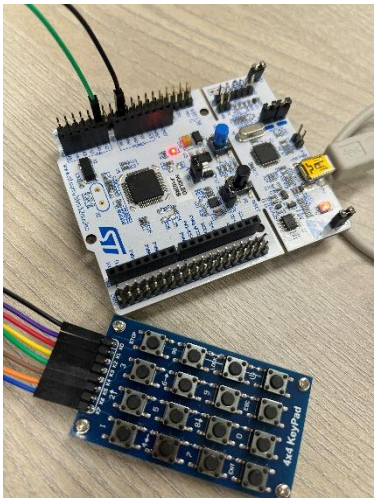


그림 20 실험 2를 위한 보드 연결

Line10의 TIM2->CR1 = 0x04; 와 line 11의 TIM2->DIER = 0x0041; 는 실험 1과는 다르게 지정하였다. 먼저 TIM2->CR1 레지스터의 2번 bit는 URS로 카운터의 overflow와 underflow에 의해서만 update interrupt와 DMA request가 발생하게 설정하는 것이다. 이렇게 설정하면 UG bit를 setting

할 때나 slave mode controller에 의해서 update interrupt와 DMA request가 발생하지 않도록 해준다. 또한 TIM2->DIER 레지스터의 0번 bit는 실험 1과 동일하게 update interrupt가 발생할 수 있도록 enable해준 것이고 또한 6번 bit에 1을 set하여 trigger interrupt또한 enable해주었다.

또한 line 16의 TIM2->SMCR = 0x8074; 를 통해서 SMCR레지스터에서 ETP를 1로 set하고 TS를 111로 set하고 SMS를 100으로 set하였다. 그 의미를 살펴보면 ETP를 1로 set하여 trigger가 active low로 동작하도록 설정하고, TS가 111로 set되어 external trigger input을 사용하도록 설정하고 SMS가 100으로 set되어 trigger가 reset mode로 동작하도록 설정하였다.

즉 요약하면 실험 1과 다르게 실험 2에서는 trigger interrupt도 사용하도록 설정하고 trigger가 reset mode로 동작하도록 하였다.

STEP 11: Lines 25-34는 TIM2의 언터럽트 service routine이다. Line 26, line 30과 같이 확인하는 이유를 line 14의 설정과 연관지어 해석해보자.

Lines 27, 31에서 GPIO의 BSRR register를 사용하는 것을 볼 수 있는데, 그 역할과 사용방법에 대해 reference manual을 통해 확인해보자. 각각은 board의 어떤 부분에 어떻게 영향을 미치도록 동작하는가? ODR register가 있음에도 불구하고 이 register를 이처럼 사용할 때 어떤 장점이 있는가?

Line 14를 통해서 TIM2에 대해서 update interrupt와 trigger interrupt가 모두 발생할 수 있도록 설정을 하였다. TIM2는 두 인터럽트를 모두 TIM2에 의해 관리되기 때문에 어떤 인터럽트가 발생하더라도 TIM2_IRQHandler가 호출되게 된다. 즉 ISR이 호출되었을 때 어떤 인터럽트에 의해서 호출된 것인지 확인해주어야 한다.

Line 27의 GPIOA->BSRR = 1 << 5; 는 PA5를 1로 set하게 된다. 다음으로 line 31의 GPIOA->BSRR = 1 << 21; 는 PA5를 0으로 reset하게 된다. 즉 update interrupt가 발생한 경우에는 line 31이 수행되어 PA5가 0으로 reset되어 LED가 꺼지고 trigger interrupt가 발생한 경우에는 line 27이 수행되어 PA5가 1로 set되어 LED가 켜지게 된다. GPIOA->ODR 레지스터에 접근하여 0이나 1을 써줄 수 있지만 BSRR레지스터를 사용하는 경우에는 atomic하게 ODR에 값을 써줄 수 있다. 즉 이번 실험 2처럼 인터럽트가 여러 개인 경우 인터럽트끼리 중복되어 호출될 수 있는데 이 때 ODR 레지스터라는 동일한 레지스터에 접근하기 때문에 atomic한 연산을 하지 않는 경우 레지스터의 동

기화가 되지 않아 오류가 발생할 수 있다는 문제가 있다. 따라서 하나의 값을 여러 인터럽트에서 접근하려 할 때는 atomic하게 동작하기 위해 BSRR 레지스터를 이용해서 set, reset하는 것이 좋다. 이에 대한 근거는 reference manual에서 가져온 아래 [그림 21]의 Note처럼 atomic한 동작을 위해서는 ODR을 통한 접근이 아닌 BSRR레지스터를 사용하는 것이 좋다.

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 21 GPIOx_ODR 레지스터 설명

STEP 12: Program 9.2(lab92.c)을 이용하여 프로젝트를 생성한다.

S0 스위치를 한 번만 눌러보고 board에 장착된 LED의 동작을 약 2-3초 동안 살펴보자.

프로그램을 수행하면서 MDK의 Peripherals-System Viewer-TIM-TIM2 메뉴를 통해 창을 열고 해당 counter의 동작을 관찰해보자.

```
1  #include <stm32f10x.h>
2
3  int main(void) {
4
5    RCC->APB2ENR |= 0x000005;
6    GPIOA->CRL = 0x00300008;
7    GPIOA->ODR = 0x01;
8
9    RCC->APB1ENR |= 0x00000001;
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);
19   TIM2->CR1 |= 0x0001;
20
21   while(1) {}
22 } //end main
23
24 void TIM2_IRQHandler(void) {
25   if((TIM2->SR & 0x0040) != 0) {
26     GPIOA->BSRR = 1 << 5;
27     TIM2->SR &= ~(1 << 6); //clear TIF
28   }
29   if((TIM2->SR & 0x0001) != 0) {
30     GPIOA->BSRR = 1 << 21;
31     TIM2->SR &= ~(1 << 0); //clar UIF
32   }
33 }
```

그림 22 직접 작성한 lab9_2.c 코드

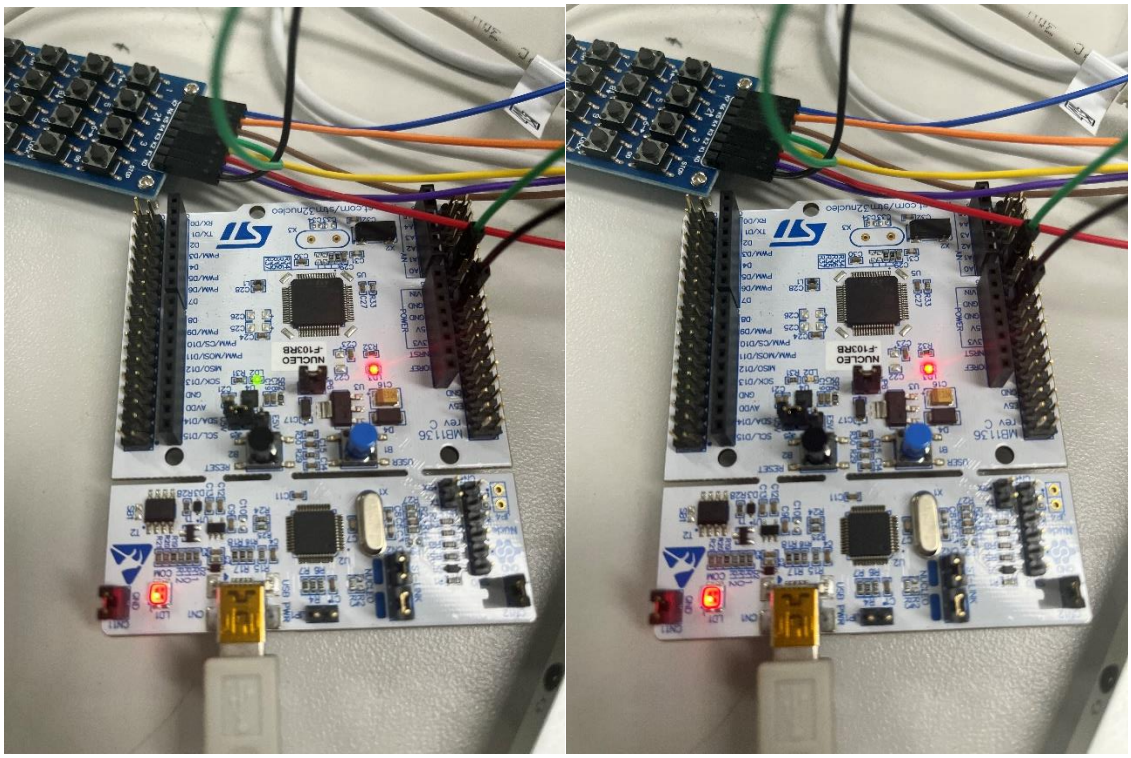


그림 23 실험 2의 동작

실험 2의 프로그램 동작은 위 [그림 23]과 같다. 스위치를 누르면 [그림 23]의 좌측처럼 LED가 켜졌다가 약간의 시간이 흐른 뒤 [그림 23]의 우측처럼 LED가 다시 꺼지게 된다. 약간의 시간이 어느정도인지 정확히 확인해 보기 위해서 PSC와 ARR의 값을 이용해서 update 빈도를 계산해 보면 우선 TIM2는 APB1을 이용함으로 TIM_CLK는 36Mhz이고 PSC = 0x5FFF, ARR = 1FFF이다. 이를 바탕으로 계산해보면 $\frac{72Mhz}{(0x5FFF+1)*(0x1FFF+1)}$ = 0.3576로 약 2.8초를 주기로 update event가 발생하는 것을 알 수 있다. 그리고 디버거 모드에서 TIM2의 레지스터을 확인해본 결과는 아래 [그림 24], [그림 25], [그림 26]과 같다.

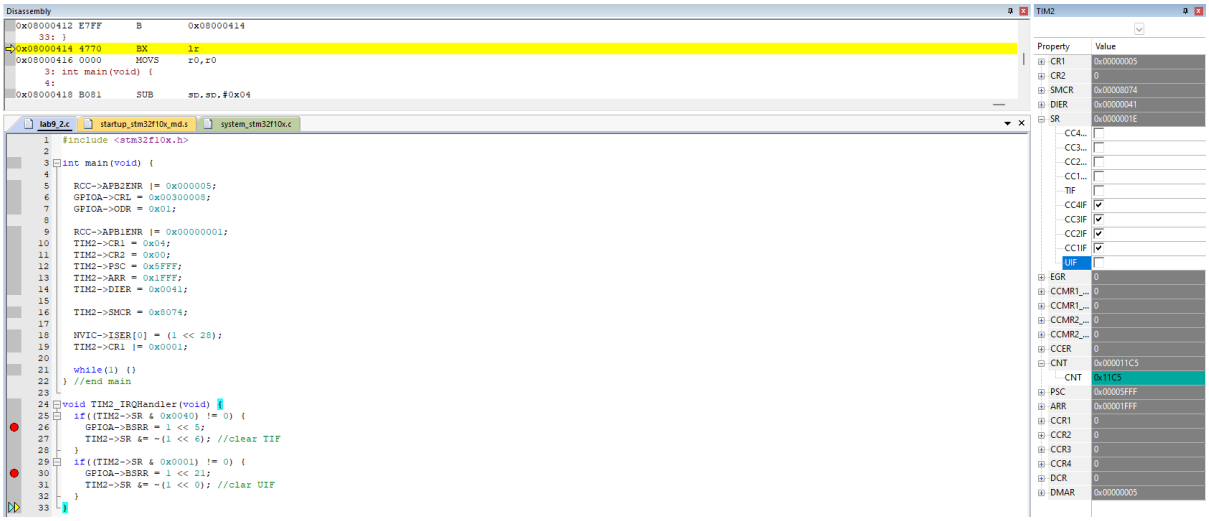


그림 24 보통의 TIM2 레지스터

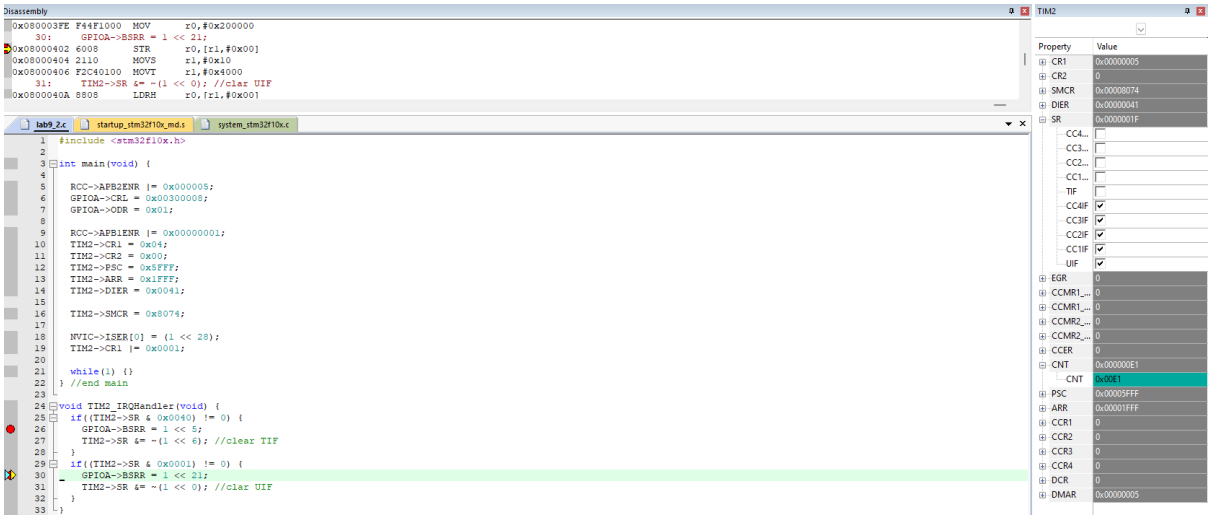


그림 25 일정 시간 후 TIM2 레지스터

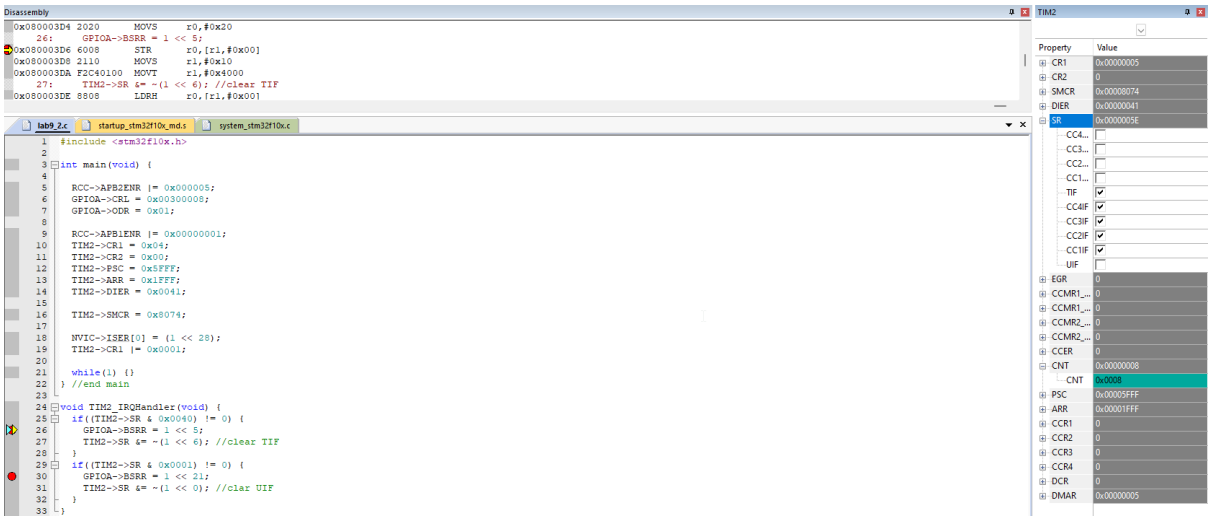


그림 26 스위치 누른 후 TIM2 레지스터

먼저 [그림 24]를 보면 아무런 인터럽트도 발생하지 않았을 때의 TIM2의 레지스터이다. TIF와 UIF 값 모두 0인 것을 알 수 있다. 그리고 [그림 25]에서 확인 가능 하듯 약 2.8초마다 counter가 overflow가 발생할 때 마다 매번 update interrupt가 발생한다. 인터럽트가 발생하면 UIF가 1로 set되고 타이머는 0부터 다시 counting 되며 TIM2 인터럽트 핸들러가 호출된다. 그림에서는 정확히 CNT값이 0은 아니지만 매우 낮은 값으로 0부터 다시 시작했다고 생각할 수 있다. 그리고 LED의 상태와 관련없이 LED를 끄게 된다. 마지막으로 [그림 26]은 스위치를 누른 직후 TIM2의 레지스터이다. Trigger가 발생하여 TIF가 1로 set되고 CNT값도 0부터 다시 counting되기 시작하였다. 그리고 핸들러로 들어가 LED의 상태와 관계없이 PA5를 1로 set해서 LED를 켜주었다. 즉 update와 trigger모두 count를 0부터 다시 시작하게 하며 각각 인터럽트는 LED를 켜고 꺼주는 역할을 수행한다.

STEP 13: 이번에는 S0 스위치를 한 번 누르고 약 0.5초 경과 후(그렇지만 LED가 꺼지기 전에) S0 스위치를 다시 눌러보자. 이상의 관찰을 바탕으로 현관의 자동문 또는 복도의 센서등의 감지 후 작동 및 시간지연과 같은 방식으로 동작한다고 말할 수 있는가?

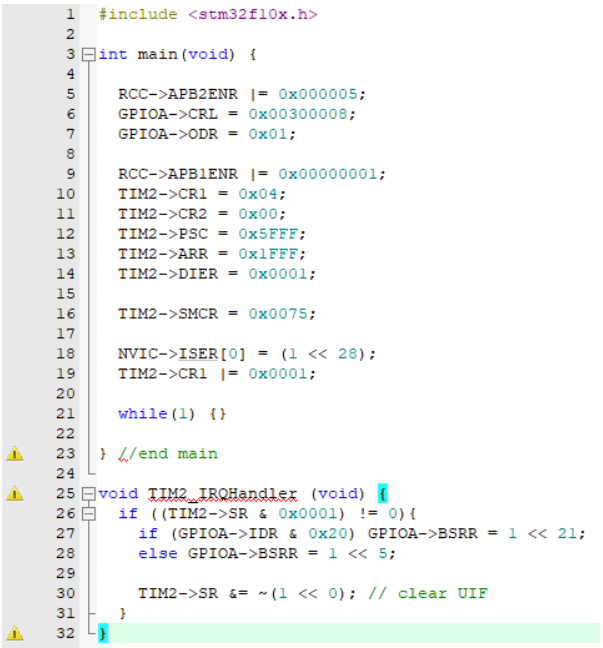
S0 스위치를 누르고 LED가 꺼지기 전에 다시 S0를 누른 경우에 마지막에 스위치를 누른 뒤부터 약 2.8초 이후에 LED가 꺼지는 동작을 확인할 수 있었다. 이는 일상생활에서 현관의 자동문이나 복도의 센서등처럼 마지막으로 신호를 받은 이후부터 정해진 시간이 지난 이후에 LED가 꺼지는 방식으로 작동한다고 말 할 수 있다.

STEP 14: Program 9.2를 다음과 같이 변경한다.

- **line 14의 0x0041을 0x0001으로.**
- **line 16의 0x8074를 0x0075로.**
- **line 25-34의 인터럽트 service routine을 다음 내용으로 교체.**

```
void TIM2_IRQHandler (void) {
        if ((TIM2->SR & 0x0001) != 0){
                if (GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
                else GPIOA->BSRR = 1 << 5;

                TIM2->SR &= ~(1 << 0); // clear UIF
        }
}
```



```
1  #include <stm32f10x.h>
2
3  int main(void) {
4
5    RCC->APB2ENR |= 0x000005;
6    GPIOA->CRL = 0x00300008;
7    GPIOA->ODR = 0x01;
8
9    RCC->APB1ENR |= 0x00000001;
10   TIM2->CR1 = 0x04;
11   TIM2->CR2 = 0x00;
12   TIM2->PSC = 0x5FFF;
13   TIM2->ARR = 0x1FFF;
14   TIM2->DIER = 0x0001;
15
16   TIM2->SMCR = 0x0075;
17
18   NVIC->ISER[0] = (1 << 28);
19   TIM2->CR1 |= 0x0001;
20
21   while(1) {}
22
23 } //end main
24
25 void TIM2_IRQHandler (void) {
26   if ((TIM2->SR & 0x0001) != 0){
27     if (GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
28     else GPIOA->BSRR = 1 << 5;
29
30     TIM2->SR &= ~(1 << 0); // clear UIF
31   }
32 }
```

그림 27 수정한 lab9_2.c 코드

STEP 15: 위 과정에서 변경한 내용을 중심으로 프로그램을 분석한다. 이 때 9.2.8절의 gated mode에 대한 설명과 그림 9.14(b)를 참조한다.

변경된 프로그램에 의해 LED는 어떤 식으로 작동할 것으로 예상되는가? 프로그램을 수행하면서 LED의 변화패턴을 관찰해보자. 관찰한 내용이 프로그램의 해석을 통해 예상했던 것처럼 동작하는가?

프로그램을 수행하면서 MDK의 Peripherals-System Viewer-TIM-TIM2 메뉴를 통해 창을 열고 해당 counter의 동작을 관찰해보자.

LED가 켜졌을 때 S0 스위치를 잠시 눌렀다 놓아보자. 눌렀던 시간만큼 켜지는 시간이 연장되었다고 볼 수 있는가? 마찬가지로 꺼졌을 때도 반복하면서 관찰해보자.

TIM2가 gated mode에서 동작했다고 볼 수 있는가? 근거(프로그램 해석과 관찰결과)를 통해 설명해보자.

먼저 수정한 부분을 중심으로 코드의 내용을 해석해보면 다음과 같다. Line 14의 TIM2->DIER = 0x0001; 은 trigger interrupt을 enable해주었던 것을 disable로 해주었다. 다음으로 line 16의 TIM->SMCR = 0x0075; 는 ETR bit를 다시 0으로 설정해 external trigger가 active high로 동작하게 해주었고 trigger가 reset mode대신 gated mode로 동작하게 설정하였다. 다음으로 TIM2 인터럽트 핸들러 내부에서는 update interrupt가 발생하였다면 GPIOA->IDR의 값을 확인해 PA5가 1이었다

면 LED를 꺼주고 PA5가 0이었다면 LED를 켜주고 update interrupt flag를 해제하고 핸들러를 탈출하는 동작을 수행한다. TIM2가 아래 [그림 28]과 같이 gated mode로 동작하는데 이 경우에는 trigger interrupt flag가 1로 set되어 있는 동안은 카운터가 증가하지 않는다. 즉 인터럽트 핸들러 코드를 바탕으로 LED의 동작을 예상해보면 버튼을 누르지 않은 경우에는 약 2.8초를 주기로 LED가 토글된다. 그리고 버튼을 누르면 TIF가 1이 되고 그 동안은 카운터가 동작하지 않아 LED의 현재상태를 계속해서 유지하게 된다.

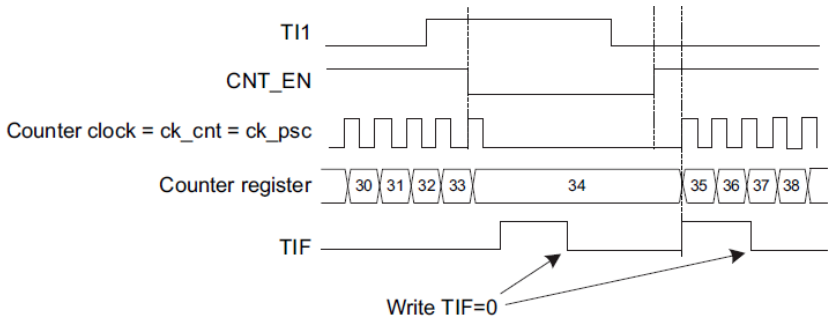


그림 28 Gated mode 동작 예시

프로그램을 보드에서 수행해본 결과 기본 동작으로는 2.8초 주기로 LED가 토글된다. 그리고 LED가 켜져있을 때 스위치를 누르면 계속해서 LED가 켜져있다. 그리고 버튼을 누르고 있지 않으면서 LED가 켜져있는 시간이 2.8초가 되면 LED가 다시 꺼진다. 동일하게 LED가 꺼져있을 때 스위치를 누르면 계속해서 LED가 꺼져있다. 즉 스위치는 카운터의 동작을 멈춰서 LED의 상태가 그대로 유지되고 일정 시간을 간격으로 LED가 토글하는 동작을 수행한다. 정확히 위에서 예상한 대로 동작하는 것을 알 수 있다.

그리고 카운터의 동작을 살펴보기 위해서 디버거 모드에서 TIM2 레지스터의 값을 살펴보면 아래 [그림 29], [그림 30], [그림 31]과 같다. 먼저 [그림 29]는 일반적인 시기의 TIM2 레지스터의 값이다. 특이한 점으로는 TIF가 1로 set되어있다는 점이다. 이는 우리가 코드에서 TIM2->SMCR의 ETP를 0으로 세팅했기 때문에 기본적으로 TIF가 1로 set되어 있다. 그리고 다음 [그림 30]에서처럼 카운터가 overflow가 발생하여 update interrupt가 발생하면 UIF가 1로 set되고 인터럽트 핸들러로 들어간다. 그리고 LED가 토글된다. 그리고 [그림 31]처럼 스위치를 누르게 되면 TIM2의 CNT값이 바뀌지 않고 그대로 유지되게 된다. 그리고 디버거상에서는 확인하기 힘들었지만 TIF가 순간적으로 0이 되었다가 1로 돌아가면서 카운터가 멈추었다가 스위치를 떼면 다시 TIF가 순간적으로 0이 되었다가 1로 돌아가며 다시 CNT가 증가하기 시작한다.

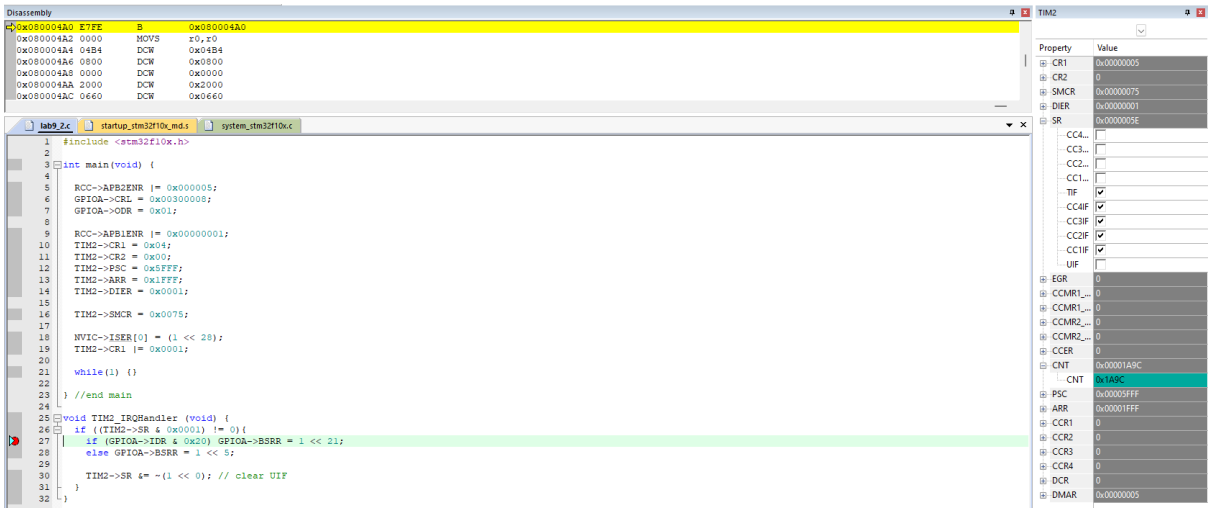


그림 29 보통의 TIM2 레지스터

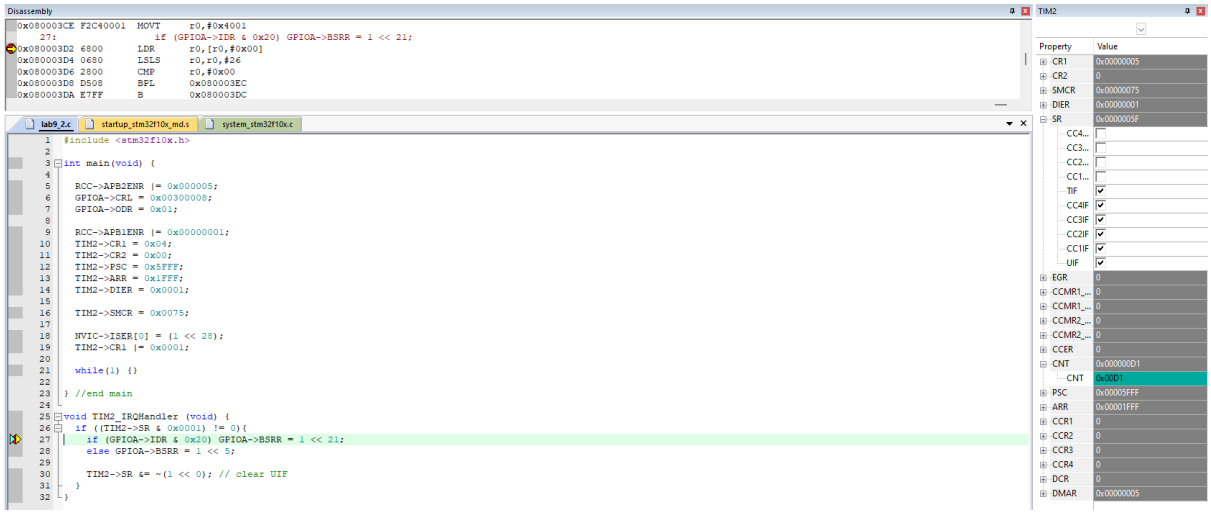


그림 30 일정 시간 후 TIM2 레지스터

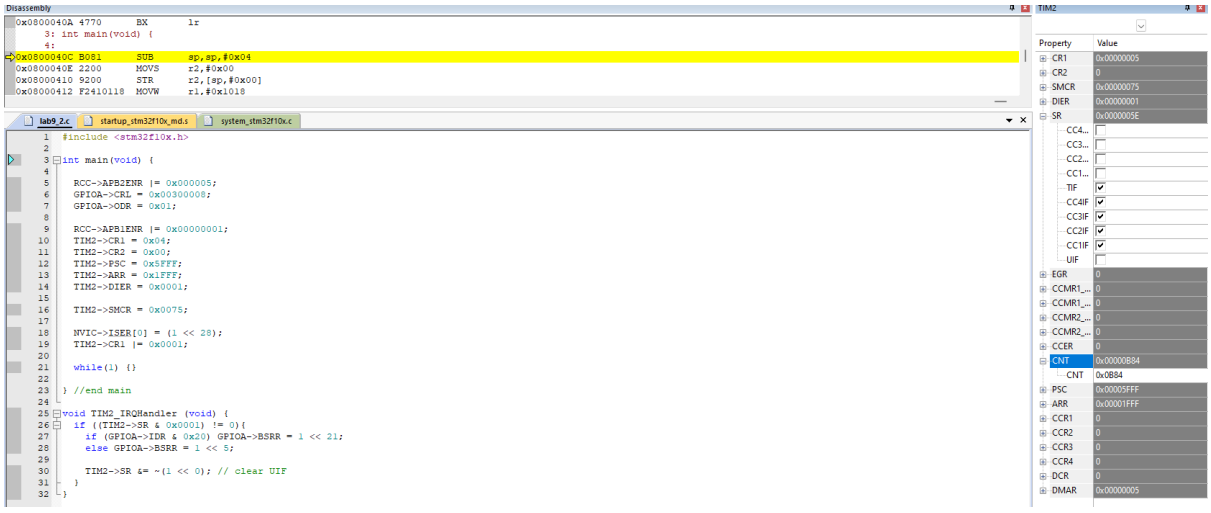


그림 31 스위치 누르는 중' TIM2 레지스터

LED가 켜졌을 때와 꺼졌을 때 S0 스위치를 눌렀다 때면 눌렀던 시간만큼 LED의 상태가 더 연장되어 유지되었다고 볼 수 있다. 실제로는 카운터의 범위가 증가한 것은 아니고 카운터의 동작이 잠시 일시정지하여 더 오랜시간 LED의 상태가 유지되었다. LED가 켜져있던 꺼져있던 상태와 관련

없이 모두 LED의 동작이 스위치를 누른 시간만큼 연장되어 동작하였다. 이로 보아 TIM2가 gated mode로 동작하였다고 생각할 수 있다. 왜냐하면 스위치를 누른 시간만큼 카운터의 update interrupt가 발생하는 시간이 증가하였는데 이로 보아 스위치를 누르는 동안은 타이머가 멈춰있었다고 생각할 수 있으며 gated mode로 동작하였다고 알 수 있다.

3) 실험 3

STEP 16: 이 프로그램은 9.2.1절 중, 180페이지의 External clock mode2: external trigger input (ETR)에서 설명하는 내용과 그림 9.5를 참조하여 구현한다. 즉, 타이머에 프로세서 내부에서 제공하는 clock 신호 대신 외부 핀을 통해 공급되는 신호를 연결하여 counter로 사용한다.

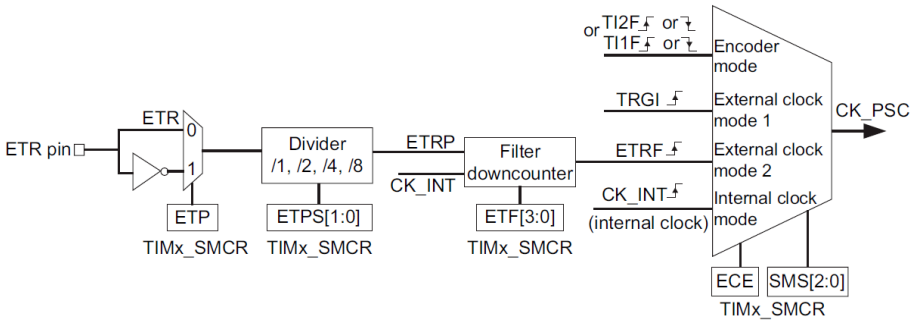


그림 32 External clock mode 2 input diagram

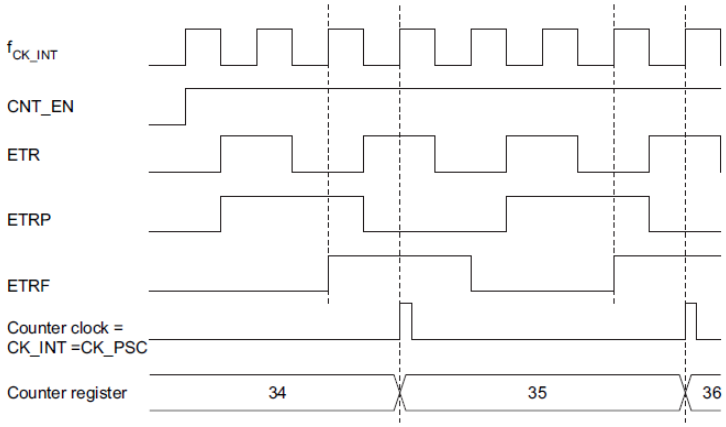


그림 33 External clock mode 2 동작 예시

위 [그림 32]는 이번 실험 3에서 사용할 external clock mode2에서의 input block diagram이다. 그리고 [그림 33]은 external clock mode2에서의 동작 예시이다. 외부 trigger인 ETR 신호의 rising edge마다 ETRP가 toggle되고 ETRP의 resynchronization delay에 의해 ETRF는 한 clock 뒤에 ETRF가 rising하고 카운터의 clock이 발생하게 된다. 이러한 동작으로 external clock mode2가 동작한다.

STEP 17: Program 9.3(lab93.c)를 분석한다. TIM2를 사용하며 clock source로 외부신호(USER 스위치)를 사용한다. Line 16에서 TIMx_SMCR을 이와 같이 초기화하는 근거를 (STMicroelectronics, 2011)을 통해 찾아본다.

```
1  #include <stm32f10x.h>
2
3  int main(void) {
4
5    RCC->APB2ENR = 0x00000005;
6    GPIOA->CRL = 0x00300008;
7    GPIOA->ODR = 0x01;
8
9    RCC->APB1ENR |= 0x00000001;
10   TIM2->CR1 = 0x04;
11   TIM2->CR2 = 0x00;
12   TIM2->PSC = 0x0;
13   TIM2->ARR = 10;
14   TIM2->DIER = 0x0001;
15
16   TIM2->SMCR = 0x4077;
17
18   NVIC->ISER[0] = (1 << 28);
19   TIM2->CR1 |= 0x0001;
20
21   while(1) {}
22
23 } //end main
24
25 void TIM2_IRQHandler(void) {
26   if((TIM2->SR & 0x0001) != 0) {
27     if(GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
28     else GPIOA->BSRR = 1 << 5;
29     TIM2->SR &= ~(1 << 0); //clear UIF
30   }
31 }
```

그림 34 직접 작성한 lab9_3.c 코드

Lines 5-19까지의 초기화 과정을 간단히 요약하면 다음과 같다. 먼저 keypad와 LD2와 연결되어 있는 PA0와 PA5를 input, output으로 설정해준다. 그리고 TIM2의 overflow/underflow에 의해서만 update interrupt가 발생하도록 하고 PSC는 0, ARR은 10으로 설정하고 update interrupt만 enable 해준다. TIMx_SMCR의 ECE를 1로 set해서 external clock mode 2를 enable해주고 TS를 111로 set 해서 external trigger input을 사용하도록 하고 SMS를 111로 set해서 slave mode에서 external clock mode 1에서 selected trigger의 rising edge를 카운터의 clock으로 사용하도록 설정한다.

사실 reference manual에서 가져온 [그림 35]에서 볼 수 있듯이 ECE bit를 1로 set하는 것은 뒤에서 TS를 111로 설정하고 SMS를 111로 설정한 것과 같은 효과를 낸다. 따라서 line 16에서 TIM2->SMCR = 0x4000; 으로 수정하여도 동일한 동작을 수행할 것이라고 예상할 수 있으며 코드 수정 후 확인 결과 동일한 동작을 수행하였다.

Bit 14 **ECE**: External clock enable

This bit enables External clock mode 2.
0: External clock mode 2 disabled
1: External clock mode 2 enabled. The counter is clocked by any active edge on the ETRF signal.
1: Setting the ECE bit has the same effect as selecting external clock mode 1 with TRGI connected to ETRF (SMS=111 and TS=111).
2: It is possible to simultaneously use external clock mode 2 with the following slave modes: reset mode, gated mode and trigger mode. Nevertheless, TRGI must not be connected to ETRF in this case (TS bits must not be 111).
3: If external clock mode 1 and external clock mode 2 are enabled at the same time, the external clock input is ETRF.

그림 35 ECE bit에 대한 설명

STEP 18: Program 9.3을 이용하여 프로젝트를 생성한다.

Lines 12-13의 설정을 참고하여 S0 스위치를 여러 번 눌러보고, overflow event 발생에 의한 인터럽트 요청을 LED의 변화를 통해 확인한다. 프로그램 분석에 따른 예측과 LED의 반응이 일치하는가?

MDK의 Peripherals-System Viewer-TIM-TIM2 메뉴를 통해 창을 열고 해당 counter의 동작을 관찰해보자.

```
1  #include <stm32f10x.h>
2
3  int main(void) {
4
5    RCC->APB2ENR = 0x00000005;
6    GPIOA->CRL = 0x00300008;
7    GPIOA->ODR = 0x01;
8
9    RCC->APB1ENR |= 0x00000001;
10   TIM2->CR1 = 0x04;
11   TIM2->CR2 = 0x00;
12   TIM2->PSC = 0x0;
13   TIM2->ARR = 10;
14   TIM2->DIER = 0x0001;
15
16   TIM2->SMCR = 0x4077;
17
18   NVIC->ISER[0] = (1 << 28);
19   TIM2->CR1 |= 0x0001;
20
21   while(1) {}
22
23 } //end main
24
25 void TIM2_IRQHandler(void) {
26   if((TIM2->SR & 0x0001) != 0) {
27     if(GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
28     else GPIOA->BSRR = 1 << 5;
29     TIM2->SR &= ~(1 << 0); //clear UIF
30   }
31 }
```

그림 36 직접 작성한 lab9_3.c 코드

Lines 12-13을 통해서 TIM2의 PSC와 ARR은 각각 0과 10이다. 따라서 이를 바탕으로 계산한 update event 주기는 0부터 10까지 count를 1분주 해서 사용하기 때문에 11번의 외부 clock마다 update interrupt가 발생하게 된다. 하지만 이를 고려하여 S0스위치를 눌러보면 11번 보다 적게 스위치를 눌렀는데도 LED가 toggle된다. 또한 8번, 9번 등등 경우에 따라 다른 횟수만큼 버튼을 눌러도 LED가 toggle되는 것을 확인할 수 있다. 즉 프로그램상으로 예측한 내용과 실제 동작에 약간의 오차가 존재한다. 사실 우리가 예상한 동작과 다르게 동작할 뿐 프로그램은 정상적으로 동작하고 있다. 왜냐하면 스위치를 누를 때 bouncing이 발생하여 우리가 스위치를 눌렀다고 생각하는 횟수보다 더 많이 타이머가 눌린 것처럼 동작하여 우리가 버튼을 눌렀다고 생각한 횟수보다 더 많이 인터럽트가 발생하여 카운터가 예상보다 빠르게 카운팅 한다. 이를 해결하기 위한 debouncing은 추가실험에서 다루어 보고자 한다.

카운터의 동작을 자세히 살펴보기 위해서 디버거 모드에서 TIM2의 레지스터 값을 살펴보면 아래 [그림 37]과 같다. 제일 왼쪽부터 진행상황을 보여준다. [그림 37]의 가장 왼쪽 그림처럼 CNT값이 0부터 시작해서 증가한다. [그림 37]의 두번째 그림은 스위치를 한 번 눌렀을 때의 상황인데 CNT가 3이나 증가한 것을 알 수 있다. 즉 스위치의 debouncing이 제대로 되지 않아 위에서 예상한 대로 버튼이 여러 번 눌렸다고 감지되었다. 다음으로 [그림 37]의 4번째 그림처럼 auto-reload된 값인 A에 도달하고 난 뒤 한 번 더 스위치가 눌리게 되면 [그림 37]의 마지막 그림처럼 UIF가 1로 set되면서 인터럽트 핸들러가 수행되고 LED의 상태가 toggle된다. 그 결과 실험 3에서의 프로그램의 동작은 아래 [그림 38]과 같이 LED가 계속해서 토글된다.

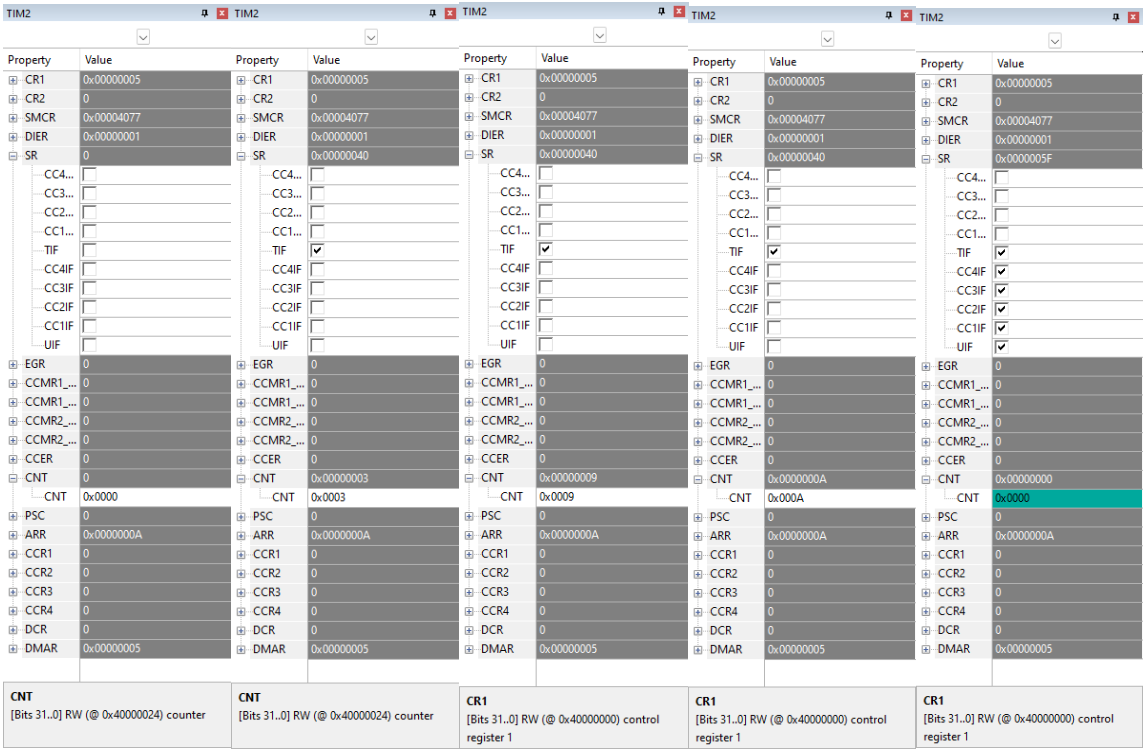


그림 37 TIM2 레지스터 값

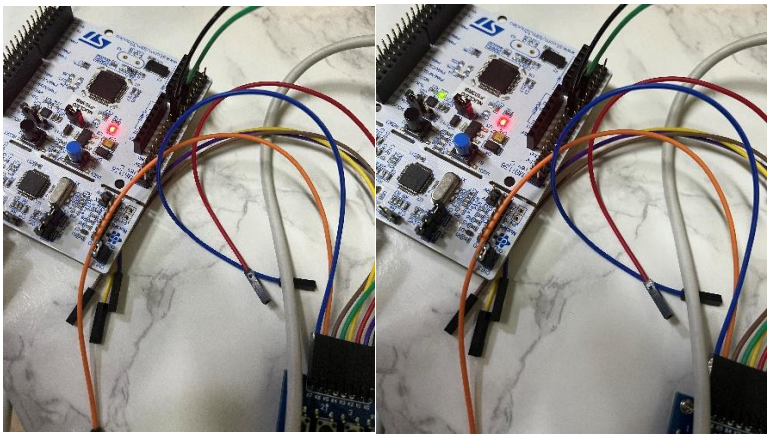


그림 38 실험 3의 LED 동작

4. Exercises

1) Timer와 counter는 hardware 관점에서 볼 때 본질적으로 동일하다. 활용에 따라 어떻게 구분할 수 있는가?

타이머는 일반적으로 일정 시간이 지난 뒤 인터럽트를 발생시키는데 프로그램에서 delay을 주거나 반복적인 작업을 위한 시간을 체크할 때, pulse 생성 등 시간과 관련된 작업이 필요할 때 주로 사용된다. 이와 비슷하지만 카운터는 일정 횟수가 지난 뒤 인터럽트를 발생시키는데 입력의 개수나 주기를 확인할 때 사용된다. 둘 모두 상호 대체적으로 사용할 수 있지만 일반적으로 타이머는 시간과 관련되서 사용하고 카운터는 횟수와 관련되서 사용된다.

2) 제시된 프로그램들을 보면 TIMx->CR1을 두 번에 나누어 초기화(설정)하는 것을 볼 수 있다 (예를 들어, Program 9.1의 lines 30, 37). 이와 같이 설정하는 이유는 무엇인가?

TIMx_CR1의 0번 bit는 CEN bit로 1로 set되면 counter enable되어 카운터가 동작하기 시작한다. 따라서 해당 bit를 1로 set하기 전에 카운터와 관련된 동작에 대한 설정을 모두 마치고 enable을 시켜주기 위해서 두 번에 나누어 설정을 진행하고 있다. 즉 다른 레지스터에서 설정을 아직 정하지 않았음으로 한 번에 CR1 레지스터의 값을 초기화해 counter가 enable되면 원하는 동작과 다르게 카운터가 동작할 수 있음으로 모든 카운터 설정을 마치고 마지막에 CEN bit만 설정해주는 작업을 한번 더 하게 된다.

3) 이전 8장에서 작성한 프로그램들에서는 시간지연이 필요할 경우 반복문을 통해 해결하였으나, 본 장에서는 타이머와 인터럽트를 이용하여 해결하고 있다. 효용성과 정확도 측면에서 두 방식을 비교하여 보자.

반복문을 사용하여 시간지연을 주는 경우보다 타이머와 타이머 인터럽트를 이용해서 코드를 작성하는 경우 더 효율적이고 원하는 시간만큼 정확히 시간지연을 줄 수 있다. 물론 타이머를 사용하는 경우 타이머를 설정하는 과정이 필요하기 때문에 시간지연이 한두 군데서만 필요하고 정확한 시간을 지킬 필요가 없다면 반복문을 사용하는 것이 작성할 코드가 더 적어 편리할 수는 있지만

타이머를 여러 번 사용해야 하거나 정확한 시간만큼 지연시키기 위해서는 타이머를 사용하는 것이 효용성이나 정확도 측면에서 더 좋다.

4) 본 장의 시작부분에서 설명한 바와 같이, 실험에 사용된 일반용도의 타이머(TIM1, TIM2 등등)가 여러 개 있음에도 불구하고 별도의 타이머들(SysTick Timer, Watchdogs)이 제공되고 있다. 이처럼 여러 타이머들을 제공하는 이유에 대해 생각해 보자.

타이머를 사용하는 용도가 다양하고 시스템을 구성할 때 자주 사용하는 타이머가 존재하기 때문에 이를 일반용도의 타이머와 분리하여 별도의 타이머를 구성하여 제공한다. 예를 들어 SysTick Timer는 운영체제와 같이 핵심 시스템에서 주기적인 인터럽트의 발생이 필요할 때 사용하게 된다. 또한 Watchdog Timer는 시스템이 정상적으로 동작 중인지 확인하는 용도의 타이머로 시스템의 동작이 일정 시간 멈추게 되면 인터럽트를 발생시켜 시스템을 리셋 하는 등의 조치를 취하기 위한 용도로 사용된다. 이처럼 시스템에서 중요한 기능을 하는 타이머는 별도로 구성하여 사용할 수 있도록 제공해준다.

5) 정확히 1초마다 인터럽트를 요청할 수 있도록 관련 레지스터들을 설정해보자.

우리가 사용하는 보드에서 TIM1의 internal clock은 72Mhz로 동작한다. 이때 정확히 1초마다 인터럽트를 발생시키기 위해서는 update event의 주기를 1Hz로 만들어야 한다. 다양한 ARR과 PSC값을 이용해서 1초마다 인터럽트를 발생시킬 수 있지만 간단히 하기 위해 ARR = 35999, PSC = 1999로 하게 되면 $72Mhz/((36000)*(2000)) = 1Hz$이 된다. 1초마다 update interrupt를 발생시키기 위한 코드는 아래 [그림 39]와 같다.

```
RCC->APB2ENR = 0x00000800;

TIM1->CR1 = 0x00;
TIM1->CR2 = 0x00;
TIM1->PSC = 35999;
TIM1->ARR = 2000;

TIM1->DIER = 0x0001;
NVIC->ISER[0] = 0x02000000;
TIM1->CR1 |= 0x0001;
```

그림 39 1초마다 인터럽트를 위한 설정

6) Digital one-shot을 타이머를 통해 구현해 보았다. 그렇다면 analog one-shot은 어떻게 동작하는지 원리를 조사해보자.

Analog one-shot은 타이머를 이용하지 않고 RC회로를 기반으로 만들어져 동작한다. 아래 [그림]에서 버튼을 눌러 전압이 들어오면 캐패시터가 충전되다가 버튼을 떼면 캐패시터에 저장되었던 전압이 U1을 통과해 비반전 Schmitt trigger 회로의 입력으로 들어간다. 캐패시트에 저장된 전압이 VCC/4가 될 때까지 출력은 VCC로 유지된 채 출력이 나와 pulse를 만든다. 이때 pulse의 폭은 $\ln(4) * RC = 1.38RC$초가 된다.

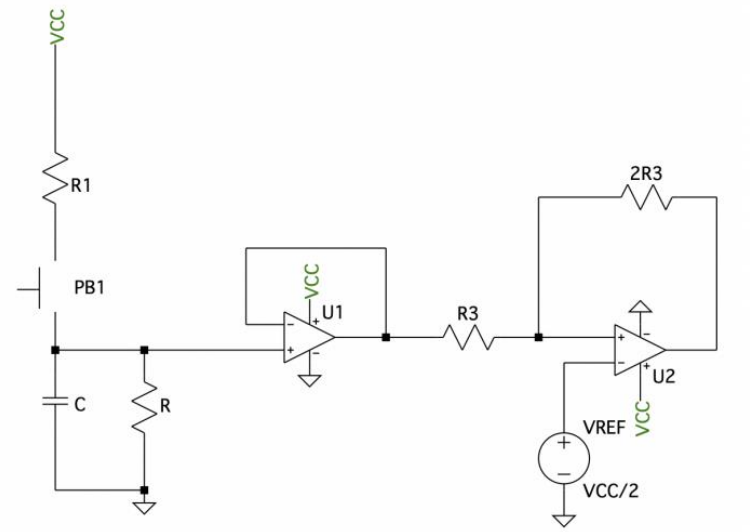


그림 40 기본적인 analog one-shot 회로

5. 추가 실험

1) TIMx Register Control And Structure

제안된 추가 실험을 진행해보기 위해서 stm32f10x.h 파일의 line 1196과 1198을 서로 바꿔보았다. 우선 2줄을 바꿔고 실험 3의 프로그램을 그대로 컴파일 한 결과 컴파일은 정상적으로 이루어졌다. 그러나 TIM2의 초기화 이후 [그림 41]의 TIM2 레지스터 값을 보면 예상대로 기존과 다르게 CR1과 CR2에 값이 정상적으로 할당되지 않은 것을 알 수 있다. 이때 CR2에 0x04가 저장되지 않은 것은 [사진 42]에서처럼 TIM2_CR2의 2번 bit는 reserved이기 때문에 우리가 쓴 값이 레지스터에 할당되지 않은 것이다.

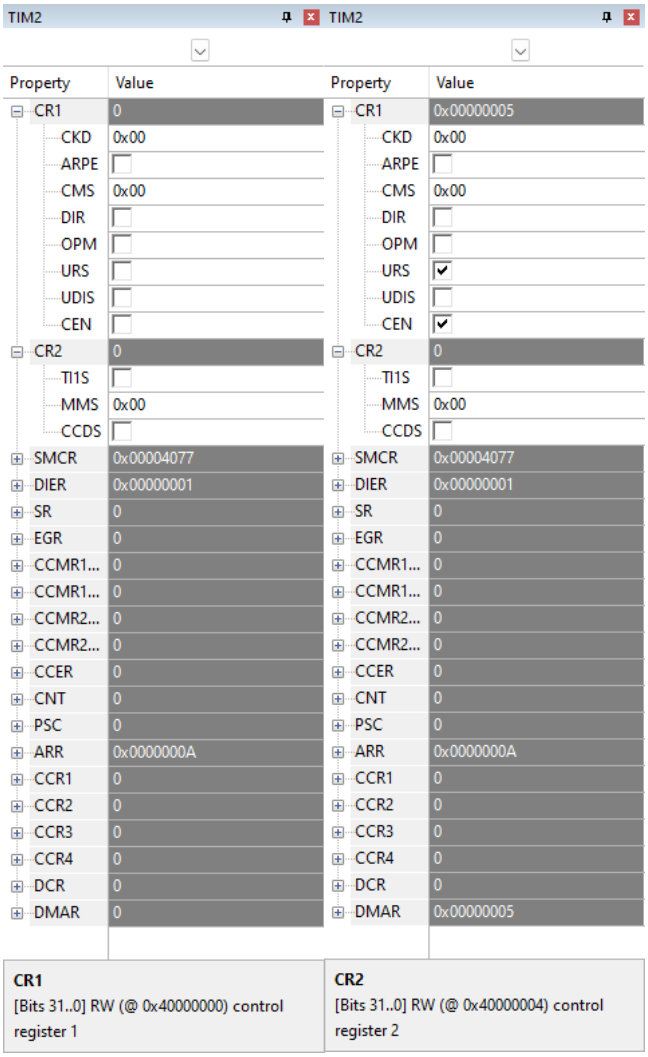


그림 41 TIM2 레지스터 (좌: 코드수정전, 우: 코드수정후)

15.4.2 TIMx control register 2 (TIMx_CR2)

Address offset: 0x04

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								TI1S	MMS[2:0]			CCDS	Reserved		
								rw	rw	rw	rw	rw			

그림 42 TIM2_CR2의 구조

이렇게 된 자세한 이유를 분석해보면 다음과 같다. 우선 line 1196, 1198은 TIM_TypeDef 구조체의 일부이다. __IO uint16_t CR1; 과 같이 선언되어 있다. 하나하나 풀어서 확인해보면 __IO는 core_cm3.h 파일 안에서 [그림 43]과 같이 volatile로 선언되어 있다. 그리고 uint16_t는 stdint.h 파일 안에서 [그림 44]와 같이 unsigned short int로 선언되어 있다.

```
169 #ifdef __cplusplus
170   #define   __I     volatile             /*!< Defines 'read only' permissions */
171 #else
172   #define   __I     volatile const       /*!< Defines 'read only' permissions */
173 #endif
174 #define     __O     volatile             /*!< Defines 'write only' permissions */
175 #define     __IO    volatile             /*!< Defines 'read / write' permissions */
```

그림 43 __IO 선언

```
61      /* exact-width unsigned integer types */
62  typedef unsigned           char uint8_t;
63  typedef unsigned short      int uint16_t;
64  typedef unsigned            int uint32_t;
65  typedef unsigned        __INT64 uint64_t;
```

그림 44 uint16_t 선언

즉 __IO uint16_t CR1; 는 바꿔서 말하면 volatile unsigned short int CR1; 이라고 할 수 있다. 먼저 volatile은 위키백과의 [그림 45]처럼 컴파일러가 최적화하지 못하게 막고 장치의 주소를 직접 접근할 수 있도록 해주는 C언어의 키워드이다. 그리고 unsiged short int는 C programming: A modern approach에서 가져온 자료인 [그림 46]에서 확인할 수 있듯이 16bit unsigned 데이터를 저장하는 자료형이다. 이와 유사하게 __IO uint16_t CR2; 도 동일하게 해석할 수 있다.

volatile 변수

문서 토론

위키백과, 우리 모두의 백과사전.

C/C++ 프로그래밍 언어에서 이 키워드는 최적화 등 컴파일러의 재량을 제한하는 역할을 한다. 개발자가 설정한 개념을 구현하기 위해 코딩된 프로그램을 온전히 컴파일되도록 한다. 주로 최적화와 관련하여 **volatile**가 선언된 변수는 최적화에서 제외된다. OS와 연관되어 장치제어를 위한 주소체계에서 지정한 주소를 직접 액세스하는 방식을 지정할 수도 있다. 리눅스 커널 등의 OS에서 메모리 주소는 MMU와 연관 된 주소체계로 논리주소와 물리주소 간의 변환이 이루어진다. 경우에 따라 이런 변환을 제거하는 역할을 한다. 또한 원거리 메모리 점프 기계어 코드 등의 제한을 푼다.

그림 45 volatile 변수 설명

Table 7.2
Integer Types on a 32-bit Machine

Type	*Smallest Value*	*Largest Value*
short int	–32,768	32,767
unsigned short int	0	65,535
int	–2,147,483,648	2,147,483,647
unsigned int	0	4,294,967,295
long int	–2,147,483,648	2,147,483,647
unsigned long int	0	4,294,967,295

그림 46 32-bit머신에서 unsigned short int

다음으로 [그림 47]의 line 1294와 [그림 48]의 line 1388로 보았을 때 TIM2->CR1과 같은 코드를 해석해 보면 (TIM_TypeDef *) (APB1PERIPH_BASE + 0x0000)이고 이는 다시 line 1290에 의해서 (TIM_TypeDef *) (PERIPH_BASE+ 0x0000)이고 다시 line 1282에 의해서 (TIM_TypeDef *) ((uint32_t)0x40000000+ 0x0000)처럼 해석할 수 있다. 즉 0x40000000의 주소를 접근하는데 이 주소는 TIM_TypeDef 포인터 타입의 정보를 가지고 있는 주소이고 이 주소에서 CR1에 접근하게 된다. 이때 만약 우리가 CR1과 CR2의 위치를 바꿔 코드가 [그림 49]와 같이 코드를 작성하면 TIM2->CR1을 접근할 때 0x40000000을 접근하는 것이 아니라 0x40000004를 접근해서 해당 주소에 값을 쓰기 때문에 실제로는 TIM2_CR2의 공간에 값이 쓰이게 된다. 마찬가지로 TIM2->CR2를 접근하면 0x40000000을 접근해서 데이터를 쓰기 때문에 실제로는 TIM2_CR1레지스터에 데이터를

쓰게 된다. 따라서 위에서 본 [그림 41]과 같이 레지스터에 원하는 값이 할당되지 않는 문제가 발생한다.

```
1280 #define FLASH_BASE            ((uint32_t)0x08000000) /*!< FLASH base address in the alias region */
1281 #define SRAM_BASE             ((uint32_t)0x20000000) /*!< SRAM base address in the alias region */
1282 #define PERIPH_BASE           ((uint32_t)0x40000000) /*!< Peripheral base address in the alias region */
1283
1284 #define SRAM_BB_BASE          ((uint32_t)0x22000000) /*!< SRAM base address in the bit-band region */
1285 #define PERIPH_BB_BASE        ((uint32_t)0x42000000) /*!< Peripheral base address in the bit-band region */
1286
1287 #define FSMC_R_BASE           ((uint32_t)0xA0000000) /*!< FSMC registers base address */
1288
1289 /*!< Peripheral memory map */
1290 #define APB1PERIPH_BASE       PERIPH_BASE
1291 #define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
1292 #define AHBPERIPH_BASE        (PERIPH_BASE + 0x20000)
1293
1294 #define TIM2_BASE             (APB1PERIPH_BASE + 0x0000)
1295 #define TIM3_BASE             (APB1PERIPH_BASE + 0x0400)
```

그림 47 Line 1294와 주변 코드

```
1388 #define TIM2                ((TIM_TypeDef *) TIM2_BASE)
1389 #define TIM3                ((TIM_TypeDef *) TIM3_BASE)
1390 #define TIM4                ((TIM_TypeDef *) TIM4_BASE)
```

그림 48 Line 1388과 주변 코드

```
1194 typedef struct
1195 {
1196   __IO uint16_t CR2;
1197   uint16_t  RESERVED0;
1198   __IO uint16_t CR1;
1199   uint16_t  RESERVED1;
1200   __IO uint16_t SMCR;
1201   uint16_t  RESERVED2;
1202   __IO uint16_t DIER;
1203   uint16_t  RESERVED3;
1204   __IO uint16_t SR;
1205   uint16_t  RESERVED4;
1206   __IO uint16_t EGR;
1207   uint16_t  RESERVED5;
1208   __IO uint16_t CCMR1;
1209   uint16_t  RESERVED6;
1210   __IO uint16_t CCMR2;
1211   uint16_t  RESERVED7;
1212   __IO uint16_t CCER;
1213   uint16_t  RESERVED8;
1214   __IO uint16_t CNT;
1215   uint16_t  RESERVED9;
1216   __IO uint16_t PSC;
1217   uint16_t  RESERVED10;
1218   __IO uint16_t ARR;
1219   uint16_t  RESERVED11;
1220   __IO uint16_t RCR;
1221   uint16_t  RESERVED12;
1222   __IO uint16_t CCR1;
1223   uint16_t  RESERVED13;
1224   __IO uint16_t CCR2;
1225   uint16_t  RESERVED14;
1226   __IO uint16_t CCR3;
1227   uint16_t  RESERVED15;
1228   __IO uint16_t CCR4;
1229   uint16_t  RESERVED16;
1230   __IO uint16_t BDTR;
1231   uint16_t  RESERVED17;
1232   __IO uint16_t DCR;
1233   uint16_t  RESERVED18;
1234   __IO uint16_t DMAR;
1235   uint16_t  RESERVED19;
1236 } TIM_TypeDef;
```

그림 49 수정한 코드

2) Timer Configuration Abstraction

```
#include <stm32f10x.h>



//TIM_timer
```

```
#define TIM11 1

#define TIM22 2

#define TIM33 3


//TIM_interrupt

#define TIM_UPDATE_INT 0x01

#define TIM_TRIGGER_INT 0x02


//TIM_flag

#define TIM_BASIC 1

#define TIM_RESET_MODE 2

#define TIM_GATED_MODE 3

#define TIM_EXTERNAL_CLOCK_MODE2 4


void my_TIM_configuration(int timer, int psc, int arr, int interrupt, int flag) {

   switch(timer) {

      case TIM11:

         RCC->APB2ENR |= 0x00000800;

         TIM1->PSC = psc;

         TIM1->ARR = arr;


         if(interrupt & TIM_UPDATE_INT) {

            TIM1->DIER |= 0x0001;

            NVIC->ISER[0] |= 0x02000000;

         }

         if(interrupt & TIM_TRIGGER_INT) {
```

```
    TIM1->DIER |= 0x0040;
    NVIC->ISER[0] |= 0x04000000;
  }

  switch(flag) {
    case TIM_BASIC:
      TIM1->CR1 = 0x00;
      TIM1->CR2 = 0x00;
      TIM1->SMCR = 0x00;
      break;
    case TIM_RESET_MODE:
      TIM1->CR1 = 0x04;
      TIM1->CR2 = 0x00;
      TIM1->SMCR = 0x8074;
      break;
    case TIM_GATED_MODE:
      TIM1->CR1 = 0x04;
      TIM1->CR2 = 0x00;
      TIM1->SMCR = 0x0075;
      break;
    case TIM_EXTERNAL_CLOCK_MODE2:
      TIM1->CR1 = 0x04;
      TIM1->CR2 = 0x00;
      TIM1->SMCR = 0x4077;
      break;
  }
```

```
    TIM1->CR1 |= 0x0001;
    break;



case TIM22:
    RCC->APB1ENR |= 0x00000001;
    TIM2->PSC = psc;
    TIM2->ARR = arr;

    if(interrupt & TIM_UPDATE_INT) {
       TIM2->DIER |= 0x0001;
       NVIC->ISER[0] |= 0x10000000;
    }
    if(interrupt & TIM_TRIGGER_INT) {
       TIM2->DIER |= 0x0040;
       NVIC->ISER[0] |= 0x10000000;
    }

    switch(flag) {
       case TIM_BASIC:
          TIM2->CR1 = 0x00;
          TIM2->CR2 = 0x00;
          TIM2->SMCR = 0x00;
          break;
       case TIM_RESET_MODE:
```

```
            TIM2->CR1 = 0x04;
            TIM2->CR2 = 0x00;
            TIM2->SMCR = 0x8074;
            break;
        case TIM_GATED_MODE:
            TIM2->CR1 = 0x04;
            TIM2->CR2 = 0x00;
            TIM2->SMCR = 0x0075;
            break;
        case TIM_EXTERNAL_CLOCK_MODE2:
            TIM2->CR1 = 0x04;
            TIM2->CR2 = 0x00;
            TIM2->SMCR = 0x4077;
            break;
    }

    TIM2->CR1 |= 0x0001;
    break;



case TIM33:
    RCC->APB1ENR |= 0x00000002;
    TIM3->PSC = psc;
    TIM3->ARR = arr;

    if(interrupt & TIM_UPDATE_INT) {
```

```
        TIM3->DIER |= 0x0001;
        NVIC->ISER[0] |= 0x20000000;
    }
    if(interrupt & TIM_TRIGGER_INT) {
        TIM3->DIER |= 0x0040;
        NVIC->ISER[0] |= 0x20000000;
    }

    switch(flag) {
        case TIM_BASIC:
            TIM3->CR1 = 0x00;
            TIM3->CR2 = 0x00;
            TIM3->SMCR = 0x00;
            break;
        case TIM_RESET_MODE:
            TIM3->CR1 = 0x04;
            TIM3->CR2 = 0x00;
            TIM3->SMCR = 0x8074;
            break;
        case TIM_GATED_MODE:
            TIM3->CR1 = 0x04;
            TIM3->CR2 = 0x00;
            TIM3->SMCR = 0x0075;
            break;
        case TIM_EXTERNAL_CLOCK_MODE2:
            TIM3->CR1 = 0x04;
```

```
            TIM3->CR2 = 0x00;
            TIM3->SMCR = 0x4077;
            break;
        }

        TIM3->CR1 |= 0x0001;
        break;
    }
}

int main(void) {
    RCC->APB2ENR = 0x00000005;
    GPIOA->CRL = 0x00300008;
    GPIOA->ODR = 0x01;

    my_TIM_configuration(TIM22, 0, 10, TIM_UPDATE_INT, TIM_EXTERNAL_CLOCK_MODE2);

    while(1) {}

} //end main

void TIM2_IRQHandler(void) {
    if((TIM2->SR & 0x0001) != 0) {
        if(GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
        else GPIOA->BSRR = 1 << 5;
        TIM2->SR &= ~(1 << 0); //clear UIF
```

```
  }
}
```

TIMx장치의 초기화 과정을 좀 더 추상화하기 위해 my_TIM_configuration()함수를 작성하였다. 해당 함수에는 파라미터로 어떤 타이머를 설정할지, prescaler값, auto-reload값, 어떤 인터럽트를 혀용할지, 어떤 방식으로 타이머를 설정할지 전부 초기화 해준다. 물론 타이머의 설정 방식은 매우 다양하지만 우선은 실험에서 사용한 방식은 전부 코드에 포함시켰다. 추후에 필요하다면 한 번만 함수내에서 초기화 하는 코드를 작성해주면 언제든지 동일한 설정으로 타미어를 세팅해줄 수 있다는 장점이 있다.

3) Peripheral Control Timing

STEP 8의 구현과제 코드를 기반으로 TIM1->PSC = 0xF; 와 TIM1->ARR = 0xF로 수정한 결과 [그림 50]과 같이 dot matrix에서 row가 하나씩 밀려서도 출력되는 것을 볼 수 있다. 기본 A문자보다 하나 다음 row에 연하게 A문자가 하나 더 그려져 있다. PSC와 ARR을 모두 0xF로 설정하였을 때 update interrupt가 발생하는 주기는 $\frac{72Mhz}{(0xF+1)*(0xF+1)} = 281250Hz$ 가 된다. 즉 매우 빠른 속도로 update interrupt가 발생한다.

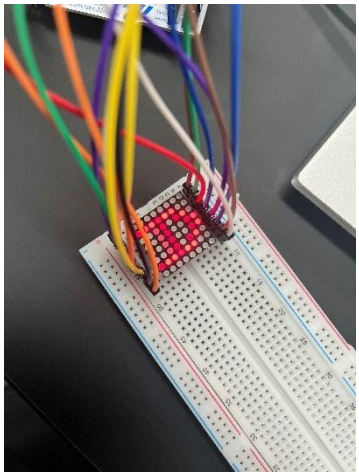


그림 50 0xF로 수정 후 결과

이때 다음 row에 연하게 A문자가 하나 더 그려지는 이유는 ISR이 수행될 때 코드상에서 row와 col을 순차적으로 내가 원하는 row와 col의 데이터로 바꾸게 되는데 row를 먼저 바꾸고 몇 개의 명령어를 지난 후에 col의 데이터를 바꾸게 된다. 따라서 아주 짧은 시간동안 이전의 col데이터로 다음 row에 LED를 켜게 된다. 이전의 col데이터를 다음 row에 쓰는 동작을 수행하는 횟수가

update interrupt가 수행되는 횟수와 동일한데 원래의 코드에서는 update interrupt가 발생하는 주기가 낮아서 아주 짧은 시간 동안만 켜지는 LED가 우리 눈에 보이지 않았는데 추가실험 3에서 바꾼 코드에서는 update interrupt가 매우 빠르게 많이 발생하기 때문에 이전의 col데이터로 다음 row에 불을 켠 LED가 우리 눈에 낮은 밝기로 보이게 된다.

이에 대해 좀 더 자세히 알아보기 위해서 PSC, ARR을 극단적으로 더 낮춰 0x0으로 수행해본 결과는 아래 [그림 51] 왼쪽과 같다. 또한 적당한 update 주기를 위해서 PSC = 0xF, ARR = 0xFF로 수행해본 결과는 [그림 51]의 가운데처럼 또렷하게 A그림이 나오는 것을 확인할 수 있었다. 그리고 조금 더 update 빈도를 줄이기 위해 PSC = 0xFF, ARR = 0xFF로 수행해본 결과 [그림 51]처럼 나왔다. 눈으로 보기에는 [그림 51]의 가운데처럼 나왔지만 카메라로 찍어보니 [그림 51]의 오른쪽처럼 dot matrix가 출력되는 과정이 보였다. 이 그림처럼 dot matrix에서 하나의 row만 나오는 게 아니라 다음 row에도 출력이 약간씩 나오는 것을 확인할 수 있었다. 이로 보아 [그림 50]에서 다음 row로 밀린 것이 이와 비슷한 이유라고 생각할 수 있다. 이처럼 dot matrix에서 또렷한 출력을 내기 위해서는 적당한 update 빈도를 가지도록 설정해주어야 하는 것을 알 수 있었다.

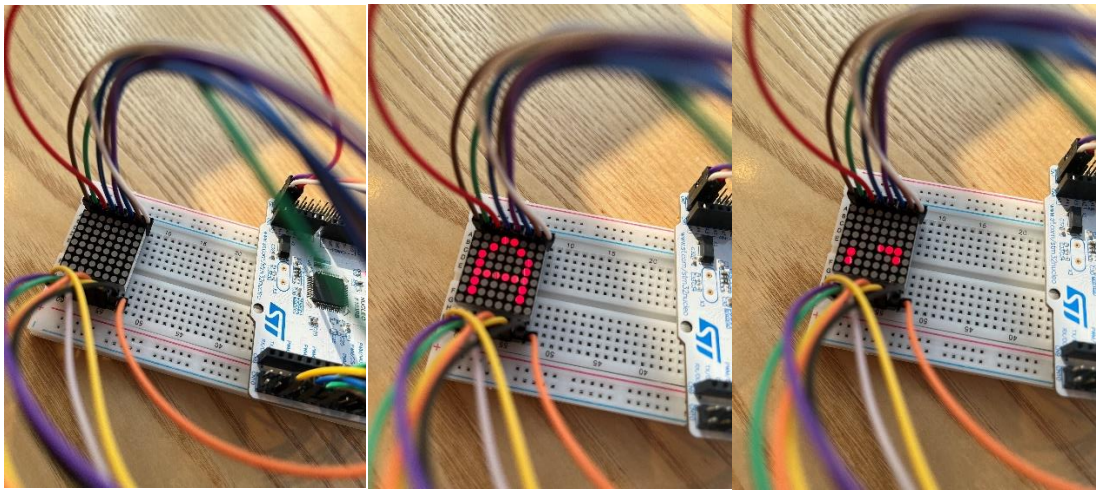


그림 51 여러 설정에서의 결과

4) PWM

```
#include <stm32f10x.h>


int main(void) {
```

```
   RCC->APB2ENR = 0x00000005;
   GPIOA->CRL = 0x00300008;
   GPIOA->ODR = 0x01;

   RCC->APB1ENR |= 0x00000001;
   TIM2->CR1 = 0x80;
   TIM2->CR2 = 0x00;
   TIM2->PSC = 7199;
   TIM2->ARR = 9999;
   TIM2->DIER = 0x0003;

   TIM2->CCMR1 = 0x0068;
   TIM2->CCER = 0x0001;
   TIM2->CCR1 = 9000;

   NVIC->ISER[0] = (1 << 28);
   TIM2->CR1 |= 0x0001;

   while(1) {}

} //end main

void TIM2_IRQHandler(void) {
   if((TIM2->SR & 0x0002) != 0) {
                        GPIOA->BSRR = 1 << 21;
```

```
    TIM2->SR &= ~(1 << 1); //clear UIF
  }
  if((TIM2->SR & 0x0001) != 0) {
                GPIOA->BSRR = 1 << 5;
    TIM2->SR &= ~(1 << 0); //clear UIF
  }
}
```

이번 실험에서는 타이머에서 중요한 기능 중 하나인 PWM을 구현해보았다. TIM2의 CH1을 이용해서 PWM을 구현하기 위해서 먼저 CR1, PSC, ARR을 구현해주었다. 그리고 update interrupt와 capture/compare 1 interrupt를 허용해주었다. 그리고 PWM에서 가장 중요한 CCR1을 설정해주는데 CCR1과 CNT를 비교해서 capture/compare 1 interrupt가 발생하기 때문에 CCR1값에 따라서 PWM의 비율이 정해진다. 인터럽트 핸들러에서는 capture/compare 1 interrupt가 발생하면 LD2를 끄고 pdate interrupt가 발생하면 LD2를 키도록 설정하였다. 즉 CCR1의 값이 증가하면 update interrupt이후 capture/compare 1 interrupt가 발생하는데 걸리는 시간이 길기 때문에 LD2가 켜져있는 시간의 비율이 크고 CCR1의 값이 작다면 LD2가 꺼져있는 시간이 커지게 된다.

그리고 이를 CCR1의 값을 1000, 5000, 9000으로 나누어서 수행해본 결과 사진상으로 꺼진 시간과 켜진 시간 차이를 알기는 어렵지만 [그림 52]와 같이 LD2가 켜졌다 꺼졌다 하는 것을 확인할 수 있었다. CCR1이 1000인 경우는 대부분 시간이 꺼져있다 잠깐 켜지는 식으로 동작하고 CCR1이 5000인 경우는 절반의 시간씩 꺼졌다 켜졌다 했으며, CCR1이 9000인 경우는 대부분 시간 켜져있다가 잠깐씩 꺼지는 동작을 수행하였다. 이처럼 PWM을 이용하여 다양한 비율로 신호를 제어할 수 있다.

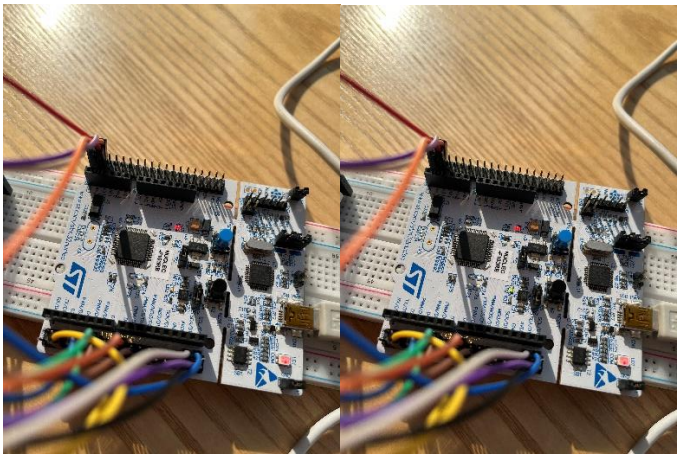


그림 52 PWM을 이용한 LD2 제어

5) Keypad Debouncing

```
#include <stm32f10x.h>


int pushed = 0;


int main(void) {


  RCC->APB2ENR |= 0x00000005;

  GPIOA->CRL = 0x00300008;

  GPIOA->ODR = 0x01;


  RCC->APB1ENR |= 0x00000001;

  TIM2->CR1 = 0x04;

  TIM2->CR2 = 0x00;

  TIM2->PSC = 7199;

  TIM2->ARR = 1000;

  TIM2->DIER = 0x0041;
```

```
    TIM2->SMCR = 0x8074;


    NVIC->ISER[0] = (1 << 28);
    TIM2->CR1 |= 0x0001;


    while(1) {}


} //end main


void TIM2_IRQHandler(void) {


    if((TIM2->SR & 0x0040) != 0) {
        pushed = 1;
        //GPIOA->BSRR = 1 << 5;
        TIM2->SR &= ~(1 << 6); //clear TIF
    }
    if((TIM2->SR & 0x0001) != 0) {
        if(pushed) {
            if(GPIOA->ODR & 0x0001) {
                if(GPIOA->IDR & 0x20) GPIOA->BSRR = 1 << 21;
                else GPIOA->BSRR = 1 << 5;
            }
        }
                        pushed = 0;
        TIM2->SR &= ~(1 << 0); //clar UIF
    }
```

```
}
```

Keypad를 누를 때 debouncing을 해주기 위해서 key가 입력되면 reset mode로 수행중인 TIM2의 trigger 인터럽트가 발생시키고 pushed를 1로 바꿔준다. 그리고 그로부터 100ms가 지난 후 TIM2의 update interrupt가 발생하여 현재까지도 버튼이 눌린 상태라면 LD2의 상태를 바꿔주도록 인터럽트 핸들러를 만들어 주었다. 그 결과 keypad의 버튼을 누를 때 여러 번 인식되지 않고 한 번씩만 인식하는 것을 확인할 수 있었다. 사진으로는 정확히 확인하기는 어렵지만 그 LD2의 동작은 아래 [그림 53]과 같다.

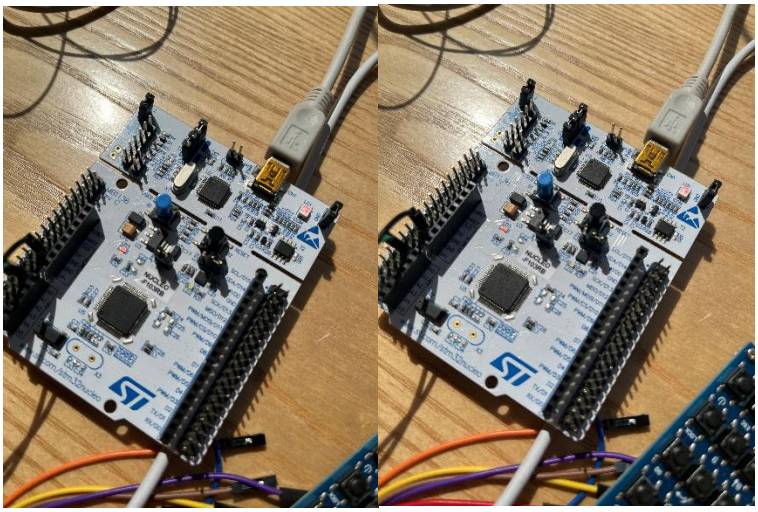


그림 53 디바운싱 후 LD2 동작

6. 결론

이번 실험을 통해서 시스템에서 매우 중요하고 사용되는 타이머의 사용법에 대해서 익혀볼 수 있었다. 또한 타이머를 사용하기 위해 설정해야 하는 레지스터의 의미와 해당 레지스터값의 의미를 이해할 수 있었다. 그리고 타이머를 다양한 방식으로 사용해볼 수 있는데 가장 기본적인 타이머의 형태부터, reset mode, gated mode, pwm 등 다양한 형태로 타이머를 동작시켜가며 내가 가진 소자를 통해서 활용해보았다.

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).

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

허언. (2021). EMBEDDED RECIPES.