임베디드 설계 및 실험 11 주차 실험 보고서

목요일 분반 3 조

202011503 강찬미

201606151 김민승

201724425 김민종

201824451 김정호

201724485 배재홍

목 표

1. 임베디드 시스템의 기본 원리 습득
2. Timer 이해 및 실습

1. 배경 지식

1.1 Timer

타이머는 주기적 시간 처리에 사용되는 디지털 카운터 회로 모듈이다. 펄스폭 계측과 주기적인 interrupt 발생 등에 사용하기 위해 쓰인다. 우선 우리는 보드의 주파수가 높기 때문에 prescaler 를 사용하여 주파수를 낮춘 후 낮아진 주파수로 8,16bit 등의 카운터 회로를 사용하여 주기를 얻는다.

타이머의 종류로는 Systick, Watchdog, Advanced-control, General, Basic 카운터 등이 있다.

1. Systick : Real-time operating System 전용으로 쓰이지만 standard down counter 로도 사용할 수 있다. 특징으로는 24bit down counter 이고, autoreload capability, counter 가 0 이 되면 설정에 따라 interrupt 발생이 가능하다. 또한 programmable clock source 를 사용하게 된다.
2. Watchdog(WDG): 임베디드 시스템 등의 특수한 상황에서 CPU 가 올바르게 작동하지 않을 시 강제로 리셋 시키는 기능을 의미한다. 특징에 따라 크게 IWDG 와 WWDG 로 나뉘는데, IWDG 는 LSI 클럭 기반으로 메인 클럭 고장에도 활성 상태 유지가 가능하고, 타이밍 정확도 제약이 낮은 어플리케이션에 적합하며 counter 가 0 이 되면 reset 이 된다.

WWDG 는 7-bit down counter 이며 WWDG 의 clock 은 APB1 clock 을 prescale 해서 정의가 가능하다. 비정상적 어플리케이션 동작 감지를 위해 설정 가능한 time-window 기능도 제공한다. 또한 counter 가 0x40 보다 작을 경우 또는 카운터가 Time-window 밖에서 Reload 됐을 경우 reset 이 가능하다. 이러한 기능들로 time-window 내에서 반응하도록 요구하는 애플리케이션에 적합하다.

3. Advanced-control: advanced control timer 는 prescaler 를 이용해 설정 가능한 16-bit auto-reload counter 를 포함하고 있다. 입력 신호 펄스 길이를 측정 또는 출력 파형 생성 등에 사용 가능하다. 또한 자원을 공유하지 않는 독립적 구조이며, 동기화가 가능하다.
4. Basic timer: 16bit -auto-reload up counter 이며 설정 가능한 16-bit prescaler 를 이용해 counter clock 주파수를 나눠서 설정이 가능하다. 지난 주 사용한 DAC 트리거로 사용한다. 또한 counter overflow 발생 시 interrupt/DMA 를 생성 한다.
5. General-purpose: 이 timer 역시 prescaler 를 이용해 16-bit up, down/down auto-reload counter 를 조절할 수 있다. 또한 Advanced-control timer 와 같이 펄스 길이 측정 또는 출력 파형 생성 등의 용도로 사용할 수 있다. 이때 펄스 길이와 파형 주기는 timer prescaler 와 RCC clock controller prescaler 를 사용하여 μs 에서 ms 까지 변조 할 수 있으며, 타이머들은 완전히 독립이고 어떠한 자원도 공유하지 않으나 동기화는 가능하다는 특징이 있다.

$$\frac{1}{f_{clk}} \times prescaler \times period$$

$$\frac{1}{72Mhz} \times 7200 \times 10000 = 1[s]$$

- 분주란 MCU에서 제공하는 Frequency를 우리가 사용하기 쉬운 값으로 바꾸어 주는 것을 말합니다.
- Counter clock frequency 를 1~65536의 값으로 나누기 위해 16-bit programmable prescaler 사용
- period 로 몇 번 count하는지 설정

$$f_{clk} * \frac{1}{prescaler} * \frac{1}{period} = \text{주파수}[Hz]$$

<General-purpose timer 의 분주 계산 공식>

1.2 PWM

PWM(Pulse Width Modulation)은 펄스 변조의 일종으로 신호의 크기에 따라 펄스의 폭을 변조하는 방식이다. 펄스 파형의 high 상태와 low 상태 파형의 비율을 duty 사이클이라고 부르는데, PWM 은 이 duty 사이클을 조정해서 변조하는 방식이다.

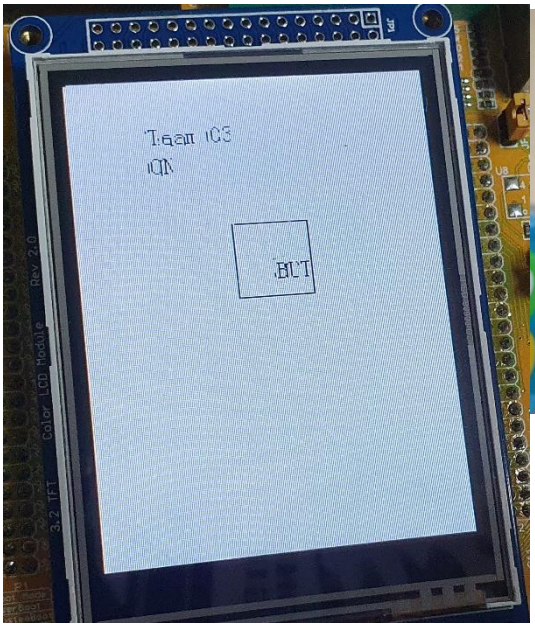
원래는 통신용으로 개발된 기술이었으나 전류,전압 제어용으로 탁월한 방식이었기 때문에 현재는 통신보다는 DC 쪽 전력 제어나 모터 제어 쪽에 쓰이는 기술이다.

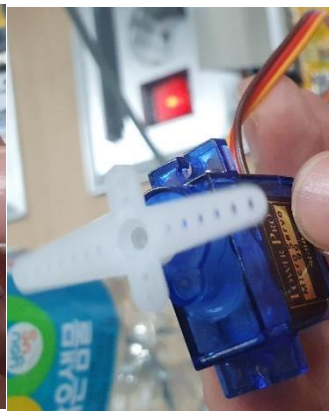
우리가 사용할 서보모터는 50Hz 의 주파수를 요구하고 해당 주파수에 맞게 Duty 사이클을 조정해주면 사용이 가능하다.

2. 실험 과정

2.1 세부 실험 내용

1. TFT-LCD 를 보드와 연결
2. 서보모터를 보드와 연결
3. TFT-LCD 에 Team 03, 토글 ON/OFF 상태, 사각형 버튼 출력
4. 사각형 버튼 터치 시 토글 ON/OFF 상태 변경
5. 토글 상태가 ON 이면, Timer2 인터럽트를 이용하여 LED1(1 초), LED2(5 초) 간격으로 Toggle, 토글 상태가 OFF 이면 Toggle 동작 멈춤
6. Timer3 를 통해 서보모터 제어



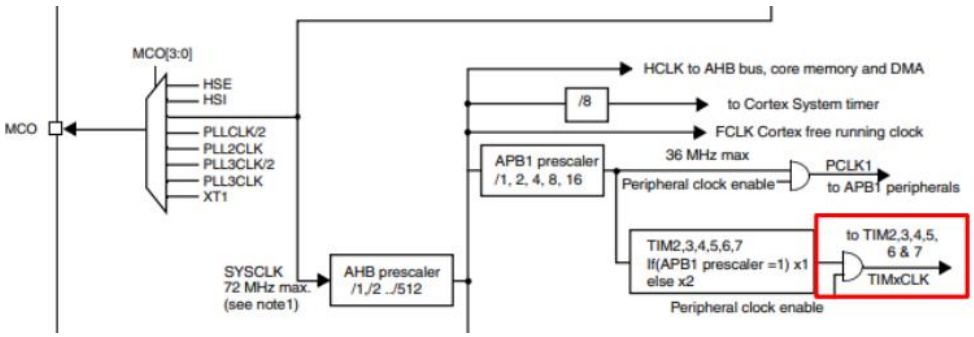
2.2 실험 방법

1. J-Link 와 보드를 컴퓨터에 연결한다.
2. IAR EW for Arm 을 실행해 project 생성 및 기본 설정을 한다.
3. 실험에 필요한 라이브러리 파일을 프로젝트에 넣는다.
4. 서보모터와 lcd 를 보드에 연결한다.
5. 10 주차 실험 코드와 stm32f10x_tim.h(.c) 파일의 내용을 바탕으로 하여 clock 인가, 타이머 설정 등 필요한 함수를 정의하고 구현한다.

- RCCInit() : Clock 를 인가해주는 함수
- GpioInit() : 사용되는 Gpio pin 의 핀번호, In/output Mode, Speed 를 설정해주는 함수
- NvicInit() : Timer2 의 인터럽트 우선순위를 설정해주는 함수
- TIM2_IRQHandler() : Timer2 의 인터럽트 발생 시 (1 초마다 발생) t1 을 증가시키고 서보모터의 각도를 변경하는 함수
- changeServoM() : 입력된 값으로 pulse 를 변경한 후 Init 해주어 서보모터의 각도를 바꿔주는 함수
- TIM_configure() : 사용되는 Timer 의 Period, Prescaler, in/out mode, pulse 등을 설정하고 사용할 수 있도록 설정해주는 함수, 이 때 아래의 식과 Reference Manual 의 Clock Tree 의 Timer clock frequency 참고하여 분주한다.

$$\frac{1}{f_{clk}} \times prescaler \times period \qquad f_{clk} * \frac{1}{prescaler} * \frac{1}{period} = \text{주파수}[Hz]$$

- 분주 계산을 위한 식 -



- Clock Tree -

6. 미션 동작을 수행할 수 있도록 main 함수의 while 문을 작성한다.

- TFT-LCD 에 Team 03, 토글 OFF 상태, 사각형 버튼 출력
- 사각형 버튼 터치 시 토글 ON/OFF 상태 변경 (ON -> OFF, OFF->ON)
- 토글 상태가 ON 이면, Timer2 인터럽트 이용하여 LED1(1 초), LED2(5 초) Toggle, lcd 에 ON 출력
- 토글 상태가 OFF 이면, LED Toggle 동작 멈춤, lcd 에 OFF 출력

3. 실험 결과 – 코드

3-1) lcd.c 설정

이번 주차에도 LCD 모듈을 사용한다. 따라서 저번 10 주차에 사용했던 lcd.c 수정을 해 주어야 한다.

```
static void LCD_WR_REG(uint16_t LCD_Reg)
{
   // TODO implement using GPIO_ResetBits/GPIO_SetBits
  GPIO_ResetBits(GPIOD, GPIO_Pin_13);   // LCD_RS(0);
  GPIO_ResetBits(GPIOC, GPIO_Pin_6);  // LCD_CS(0);

  GPIO_ResetBits(GPIOD, GPIO_Pin_14); // LCD_WR(0);

  GPIO_Write(GPIOE, LCD_Reg);

      // TODO implement using GPIO_ResetBits/GPIO_SetBits
  GPIO_SetBits(GPIOC, GPIO_Pin_6);  // LCD_CS(1);
  GPIO_SetBits(GPIOD, GPIO_Pin_14); // LCD_WR(1);
}
```

Reg 를 LCD 에 전송하기 위해 RS, CS, WR 를 GPIO_ResetBit() 함수를 사용해 LOW 로 설정해준다. 다음으로 GPIO_Write() 함수를 사용하여 LCD_REG 를 전송한다. 전송이 끝나면 CS 와 WR 을 GPIO_SetBits() 함수를 HIGH 로 설정해준다.

```
static void LCD_WR_DATA(uint16_t LCD_Data)
{
   // TODO implement using GPIO_ResetBits/GPIO_SetBits
  GPIO_SetBits(GPIOD, GPIO_Pin_13);   // LCD_RS(1);
  GPIO_ResetBits(GPIOC, GPIO_Pin_6);  // LCD_CS(0);
  GPIO_ResetBits(GPIOD, GPIO_Pin_14); // LCD_WR(0);

  GPIO_Write(GPIOE, LCD_Data);

   // TODO implement using GPIO_ResetBits/GPIO_SetBits
  GPIO_SetBits(GPIOC, GPIO_Pin_6);  // LCD_CS(1);
  GPIO_SetBits(GPIOD, GPIO_Pin_14); // LCD_WR(1);
}
```

위와 같이 GPIO_ResetBits()/GPIO_SetBits()를 사용해 RS 는 HIGH 로 CS, WR 은 LOW 로 설정해준다. 그 후 GPIO_Write()함수를 사용하여 LCD_Data 를 전송한다. 전송이 끝나면 CS 와 WR 을 HIGH 로 설정해준다.

3-2) 전처리 및 RCC/GPIO Configure

stm 보드의 Timer 2 번 3 번을 사용하기 위해 RCCInit 에서 클럭 인가를 한다.
또한 LED 및 Timer3 의 채널 3 번을 사용하기 위해 GPIOB 및 D 에 클럭 인가를 한다.
1,2 번 LED 사용을 위해 GPIOInit 함수를 활용하여 D2,3 번 Port 를 초기화 한다.
Timer3 의 채널 3 번 사용을 위해 B0 Port 를 Alternative Mode 로 초기화 한다.

```
#include "stm32f10x.h"
#include "core_cm3.h"
#include "misc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_adc.h"
#include "lcd.h"
#include "touch.h"
#include "stm32f10x_tim.h"

void RCCInit() {
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

}

void GpioInit() {
  GPIO_InitTypeDef GPIO_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure_LED;

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //PIN B0 = ADC_Channel_8
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP ; //or OD
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  GPIO_InitStructure_LED.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
  GPIO_InitStructure_LED.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure_LED.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOD, &GPIO_InitStructure_LED);
}
```

3-3) Timer Configure

Timer Configure 함수를 이용하여 Timer2 와 3 의 설정을 조율한다.

MCU 에서 제공하는 72MHz 의 주파수를 적당히 분주하여 원하는 대로 조정하는데 방법은 다음과 같다.

Period 와 Prescaler 의 값만큼 -물론 범위가 존재하지만- 주파수를 나누어 줄 수 있다. 따라서 72MHz/10000*7200 으로 Timer2 의 경우 1Hz 의 주파수를 갖고 Timer3 의 경우 72MHz/20000*72 로 50Hz 의 주파수를 갖게 되는데 이는 서보모터의 권장 주파수가 50~1000Hz 이기 때문이다.

Timer2 의 경우 분주 된 주파수를 그대로 사용하기 때문에 TODO 의 Period 와 Prescaler 값만 조절해주면 되지만, Timer3 의 경우 추가적으로 Pulse 값을 조정해주어야 한다.

Pulse 값은 분주한 주파수의 Duty 값을 정해주기 위해 조정하는 값이다.

Pulse 값에 따라 서보모터의 날개 각도가 달라지게 된다.

이번 실험에서 초기 값을 1500 으로 두었다.

```
void TIM_configure() {
  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

  TIM_TimeBaseStructure.TIM_Period = 10000;
  TIM_TimeBaseStructure.TIM_Prescaler = 7200;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
  TIM_ARRPreloadConfig(TIM2, ENABLE);
  TIM_Cmd(TIM2, ENABLE);
  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  ////////////////////////////////////////////////

  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure_2;
  TIM_OCInitTypeDef TIM_OCInitStructure_2;

  TIM_TimeBaseStructure_2.TIM_Period = 20000;
  TIM_TimeBaseStructure_2.TIM_Prescaler = 72;                  //set 50Hz
  TIM_TimeBaseStructure_2.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure_2.TIM_CounterMode = TIM_CounterMode_Down;

  TIM_OCInitStructure_2.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure_2.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OCInitStructure_2.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure_2.TIM_Pulse = 1500;                          // set move 0
  TIM_OC3Init(TIM3, &TIM_OCInitStructure_2);

  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure_2);
  TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Disable);
  TIM_ARRPreloadConfig(TIM3,ENABLE);
  TIM_Cmd(TIM3,ENABLE);
}
```

3-4) NVIC_Configure / TIM2_IRQHandler / changeServoM

NVIC 를 활용하여 인터럽트 설정을 해주고 Timer2 의 인터럽트 Handler 를 조정하여 시간을 계산한다. 또한 changeServoM 함수를 이용해 서보모터의 날개 각도를 조정하는데 이 때에 Handler 의 계산한 시간을 활용하여 1 초에 1 회씩 날개를 돌아가게 한다.

NvicInit 함수로 인터럽트 설정을 한다.

```
void NvicInit() {
  NVIC_InitTypeDef NVIC_InitStructure;
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 01;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

이번 실험에서 Timer2 번의 인터럽트 설정을 해 주되 우선순위는 고려하지 않는다.

changeServoM 함수에서 Timer3 의 펄스값을 조정하여 서보모터의 날개 각도를 변하게 한다.

정수형의 t1 변수를 0 으로 초기화 하고 전역으로 두어 1 초에 한번씩 호출되는 Timer2 번의 인터럽트 함수를 이용해 시간을 계산하고 거기에 더해 서보 모터 또한 1 초에 한번 씩 돌아가게 한다.

TIM_ClearITPendingBit 함수를 활용 해 인터럽트를 초기화 하여 핸들러가 다시 호출되게 한다.

```
void changeServoM(int percent){  //(percent,move) 3.5 (-90),  7.5(0), 10(90)

  int myPulse = percent;

  TIM_OCInitTypeDef TIM_OCInitStructure_2;

  TIM_OCInitStructure_2.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure_2.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OCInitStructure_2.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure_2.TIM_Pulse = myPulse;
  TIM_OC3Init(TIM3, &TIM_OCInitStructure_2);
}

int t1 = 0;

int arr[3] = {700, 1500, 2300};
void TIM2_IRQHandler(void) {
  t1++;
  if(t1==0xffff) t1 = 1;
  TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
  changeServoM(arr[t1%3]);
}
```

3-5) main 함수

```
uint16_t flag = 0;
uint16_t x, y;
uint16_t lcd_x, lcd_y;
int main() {
  SystemInit();
  RCCInit();
  GpioInit();
  TIM_configure();
  NvicInit();
//---------------------------------
  LCD_Init();
  Touch_Configuration();
  Touch_Adjust();
  LCD_Clear(WHITE);

  while(1){
    LCD_ShowString(50, 30, "Team 03", BLACK, WHITE);

    LCD_DrawRectangle(100, 100, 150, 150);
    LCD_ShowString(125, 125, "BUT", BLACK, WHITE);

    Touch_GetXY(&x, &y, 0);
    Convert_Pos(x, y, &lcd_x, &lcd_y);


    if (flag ^ (100 < lcd_x && lcd_x < 150 && 100 < lcd_y && lcd_y < 150)) {
      flag = 1;
    }
    else { flag = 0; }

    if (flag == 1) {
      LCD_ShowString(50, 50, "ON  ", BLACK, WHITE);
      if (t1 % 2) {
        GPIO_SetBits(GPIOD, GPIO_Pin_2);
      }
      else {
        GPIO_ResetBits(GPIOD, GPIO_Pin_2);
      }
      if (t1 % 10 < 5) {
        GPIO_SetBits(GPIOD, GPIO_Pin_3);
      }
      else {
        GPIO_ResetBits(GPIOD, GPIO_Pin_3);
      }
    }
    else {
      GPIO_ResetBits(GPIOD, GPIO_Pin_2);
      GPIO_ResetBits(GPIOD, GPIO_Pin_3);
      LCD_ShowString(50, 50, "OFF", BLACK, WHITE);
    }
  }
}
```

앞서 작성한 함수를 통해 클럭 인가 및 초기화 작업을 하고 while 문을 활용하여 요구 동작을 구현한다.

먼저 LCD_ShowString 함수를 이용해 좌상단에 Team03 을 출력한다.

LCD_DrawRectangle 함수를 이용해 버튼의 역할을 할 사각형을 출력하고 버튼임을 표시하기 위해 LCD_ShowString 함수를 활용, But 라는 문자열을 사각형 내부에 출력한다.

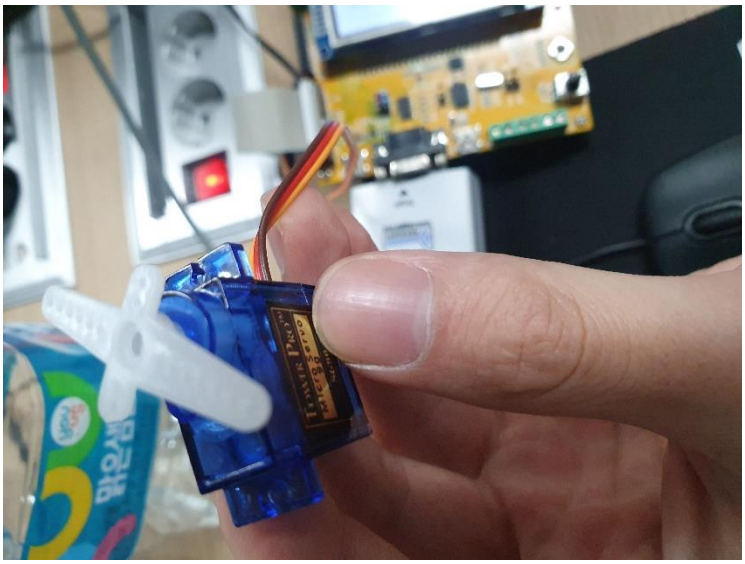
Touch_GetXY 함수를 활용 해 사용자가 클릭한 스크린의 위치를 알아내고 Convert_Pos 함수를 이용해 스크린의 좌표값을 직관적인 값으로 전환한다.

조건문을 활용 해 버튼을 클릭할 때 마다 토글 방식으로 ON OF 가 전환되게 하고 ON 상태일 시 ON 을 좌상단에 출력하고 LED1 번은 1 초마다 2 번은 5 초마다 토글되게 한다.

3-6) 동작 결과 및 회로 사진

ON 상태에서 LED 가 점등되는 모습

서보 모터가 돌아가는 모습



4. 결론

이번 실험에서는 타이머와 분주 계산, PWM 제어를 통해 서보모터를 제어하는 방법을 배웠다.

타이머 종류에는 Systick timer, IWDG/WWDG Timer-Watching timer, Advanced-control timer, Basic timer, General-purpose timer 가 있으며 이번 실험에서는 General-purpose timer 두 개를 사용했다. 한 타이머는 PWM 제어를 하는데 사용했고 나머지 한 개는 interrupt 를 통해 시간을 측정하는 용도로 사용했다.

분주는 prescaler 와 period 의 값을 설정해 frequency 를 조절하는 것인데 시간을 측정하는 타이머는 잘 분주 됐으나 PWM 제어를 하는 타이머는 분주에 어려움이 있었다. 처음 period 를 72, prescaler 를 20000 으로 설정해 frequency 를 50 으로 두고 서보모터를 제어했으나 서보모터가 원하는 대로 움직이지 않았다. 그리고 period 값을 20000, prescaler 값을 72 로 바꿔주었더니 정상적으로 작동했다. period 와 prescaler 값은 0~65535 사이의 값으로 바뀌어도 정상적으로 동작해야 한다고 생각했으나 그렇지 않았다. 이유는 아직 파악하지 못했다.