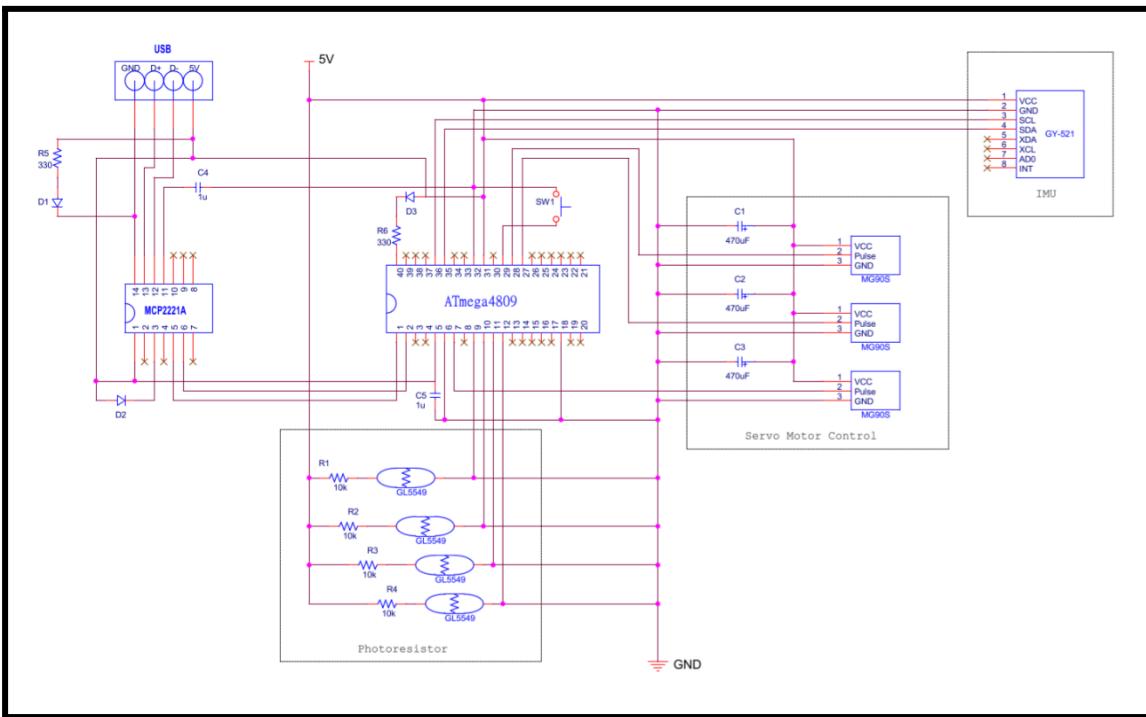


프로젝트 명(윤대민, 김태완, 이재영): 사물 tracking 3축 짐벌



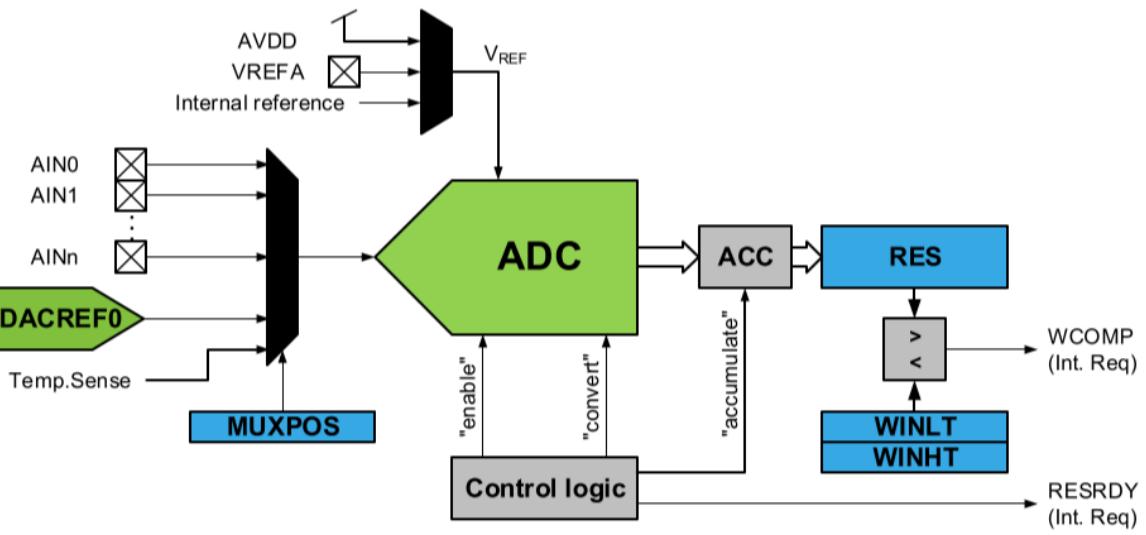
Orcad을 이용한 회로도 구성은 위와 같다. 실습 때 이용하던 Atmega4809와 MCP2221A의 회로를 바탕으로 내부의 풀업 저항을 이용할 수 있는 GY-521를 MCU와 I2C통신으로 연결하였고, 3개의 Servo Motor를 PWM Output으로 연결하여 안정적으로 사용하기 위해 각각에 전해 축전기를 연결해주었다. 4개의 Photoreistor는 풀업 저항을 이용하여 과전류를 방지한다. 측정되는 빛의 양이 많을수록 측정되는 전압의 크기는 작아진다.

사물 tracking 3축 짐벌			윤대민, 김태완, 이재영			
Bill Of Materials			May 14 2022			
Item	Quantity	Reference	Part	Part name	Package type	가격
1	3	C1,C2,C3	Electrolytic capacitor	470uF, 10V		320
2	2	C4,C5	Capacitor	1uF		-
3	1	D1	Red LED			-
4	1	D2	Green LED			-
5	1	D3	Yellow LED			-
6	1	SW1	Push button (Switch)			-
7	6	R1,R2,R3,R4,R5,R6	Resistor	10kΩ*4, 330Ω*2		-
8	1	U2	IMU	GY-521		3,500
9	3	U3,U4,U6	Servo motor	MG90S		12,600
10	4	U7,U8,U9,U10	Photoresistor	GL5549		1,200
11	1	U12	USB to Serial Convertor	MCP2221A	14-DIP	-
12	1	U11	MCU	ATmega4809	40-DIP	-
13	1		Battery	YJ603450		5,500
14	1		만능기판	100x100 사각 만능기판	100mm x 100mm	-
15	1		외형 제작	포맥스	(3T, 297x420)*2	15,000
			전체가격			38,120

회로도를 바탕으로 필요한 부품을 BOM으로 정리하였으며 외형 제작에 필요한 단단한 재료를 찾아본 결과 폴리스틸렌을 발포한 것을 고밀도로 압축한 포맥스가 적합해 보였다. 꽤 가격이 나가지만 단단한 물체를 유지하기 위해 최적의 재료인 듯하다.

1. ADC

Figure 2-1. ADC Block Diagram



Bit	7	6	5	4	3	2	1	0
	MUXPOS[4:0]							
Access				R/W	R/W	R/W	R/W	R/W
Reset				0	0	0	0	0

bits 4:0 MUXPOS[4:0]: MUXPOS bits

This bit field selects which single-ended analog input is connected to the ADC. If these bits are changed during a conversion, the change will not take effect until this conversion is complete.

MUXPOS	Name	Input
0x00-0x0F	AIN0-AIN15	ADC input pin 0 - 15
0x10-0x1B	-	Reserved
0x1C	DACREF0	DAC reference in AC0
0x1D	-	Reserved
0x1E	TEMPSENSE	Temperature Sensor
0x1F	GND	GND
Other	-	Reserved

Atmega4809 ADC datasheet를 통해 MUXPOS Register로 해당하는 Analog input port와 연결할 수 있는 것을 확인하였고 이를 이용하여 4개의 CDS로 얻어 Result Register에 저장된 값을 시리얼 모니터로 확인해보고자 하였다.

```

void loop() {
    if( ADC0_INTFLAGS & PIN0 ) // Read Ready interrupt Enable
    {
        ADC0_MUXPOS = 0x0c;
        USART1_string_send("A0 : ");
        USART1_Transmit(ADC0_RESL); //ADC0_RESL 전송
        delayMicroseconds(100); // 데이터 밀림을 방지하기 위함

        ADC0_MUXPOS = 0x0d;
        USART1_string_send(" A1 : ");
        USART1_Transmit(ADC0_RESL);
        delayMicroseconds(100);

        ADC0_MUXPOS = 0x0e;
        USART1_string_send(" A2 : ");
        USART1_Transmit(ADC0_RESL);
        delayMicroseconds(100);

        ADC0_MUXPOS = 0x0f;
        USART1_string_send(" A3 : ");
        USART1_Transmit(ADC0_RESL);
        USART1_string_send("\n");
        delay(200);
    }
}

```

```

void loop() {
    if( ADC0_INTFLAGS & PIN0 )
    {
        ADC0_MUXPOS = 0x0c;
        mySerial.print("A0 : ");
        mySerial.print(ADC0_RESL);
        delayMicroseconds(100);

        ADC0_MUXPOS = 0x0d;
        mySerial.print(" A1 : ");
        mySerial.print(ADC0_RESL);
        delayMicroseconds(100);

        ADC0_MUXPOS = 0x0e;
        mySerial.print(" A2 : ");
        mySerial.print(ADC0_RESL);
        delayMicroseconds(100);

        ADC0_MUXPOS = 0x0f;
        mySerial.print(" A3 : ");
        mySerial.println(ADC0_RESL);
        delay(200);
    }
}

```

ADC0_CTRLA |= ADC_RESSEL_8BIT|ADC_ENABLE | ADC_FREERUN;

```

#define ADC_RESSEL_8BIT (0x01<<2)
#define ADC_ENABLE (0x01)
#define ADC_STCONV (0x01)
#define ADC_FREERUN (0x02)

#define PIN0 (1<<0)
#define PIN1 (1<<1)

```

PF4 ~ PF7 핀을 이용하여 MUXPOS를 변경해가며 각각의 CDS 값을 확인할 수 있었다. 8bit 분해능과 Free Running Mode를 이용하였으며 상단 좌측의 코드는 USART를 레지스터 코딩으로 구현하여 시리얼 모니터로 확인해보았는데 char 타입의 Extended ASCII 값으로 출력되는 것을 알 수 있었다. 우측의 softwareserial 라이브러리를 이용한 코드는 정수 타입으로 값을 확인할 수 있었다. 출력하는 PIN을 변경할 때 데이터가 밀리는 것을 방지하기 위해 100us의 delay를 넣어주었다.

```

A0 : # A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : # A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : ° A3 : ¶
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : L A3 : °
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : # A1 : ¶ A2 : ° A3 : °
A0 : # A1 : ¶ A2 : ° A3 : L
A0 : # A1 : ¶ A2 : ° A3 : °
A0 : # A1 : !! A2 : ° A3 : °
A0 : # A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : " A1 : ¶ A2 : ° A3 : °
A0 : " A1

```



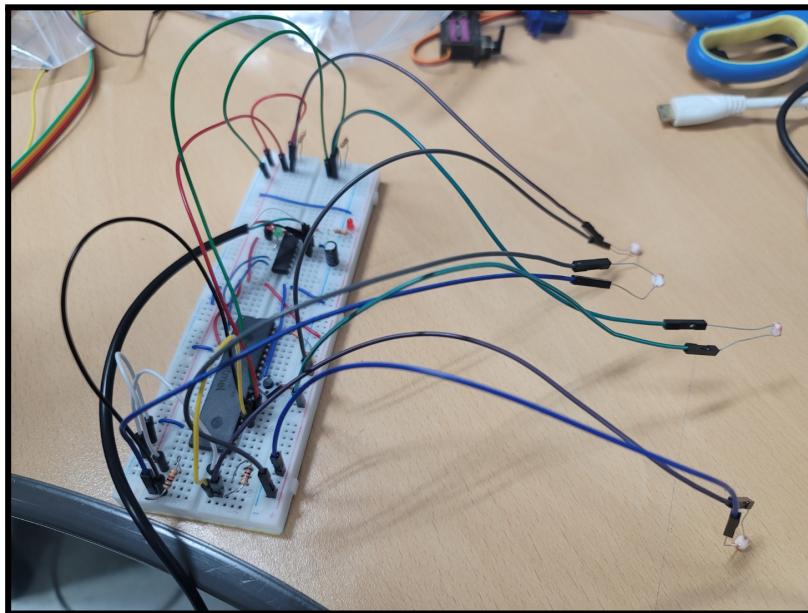
```

A0 : 14 A1 : 16 A2 : 13 A3 : 13
A0 : 14 A1 : 17 A2 : 14 A3 : 15
A0 : 13 A1 : 17 A2 : 15 A3 : 17
A0 : 14 A1 : 18 A2 : 15 A3 : 18
A0 : 13 A1 : 17 A2 : 15 A3 : 18
A0 : 15 A1 : 18 A2 : 255 A3 : 17
A0 : 40 A1 : 23 A2 : 35 A3 : 15
A0 : 29 A1 : 46 A2 : 33 A3 : 40
A0 : 27 A1 : 46 A2 : 33 A3 : 42
A0 : 29 A1 : 45 A2 : 31 A3 : 40
A0 : 29 A1 : 45 A2 : 31 A3 : 40
A0 : 30 A1 : 44 A2 : 31 A3 : 40
A0 : 31 A1 : 43 A2 : 30 A3 : 40
A0 : 31 A1 : 43 A2 : 30 A3 : 39
A0 : 32 A1 : 43 A2 : 30 A3 : 39
A0 : 32 A1 : 42 A2 : 29 A3 : 39
A0 : 33 A1 : 42 A2 : 29 A3 : 39

```

자동 스크롤 타임스탬프 표시

자동 스크롤 타임스탬프 표시



100us는 작은 delay여서 4개의 CDS로 얻는 값을 거의 동시에 확인할 수 있었으며 회로는 위의 사진과 같이 구현해보았다. 10k옴의 풀다운 저항을 추가하여 빛이 너무 밝아 CDS의 저항이 0이 되어도 과전류가 흐르는 것을 방지하였다.

4개의 CDS를 통해 값을 얻는 것을 확인하였고 이를 이용해 솔라 트래커와 같은 방식으로 Tracking 기능을 구현해 볼 예정이다.

2. PWM

TCA Split Mode를 사용하면, Data sheet 상에서 확인하였을 때, 2개의 8-bit Timer/Counter 를 통해 6개의 PWM을 구현할 수 있는 것으로 확인하였고, TCA Split Mode로 아래와 같이 구현하였다.

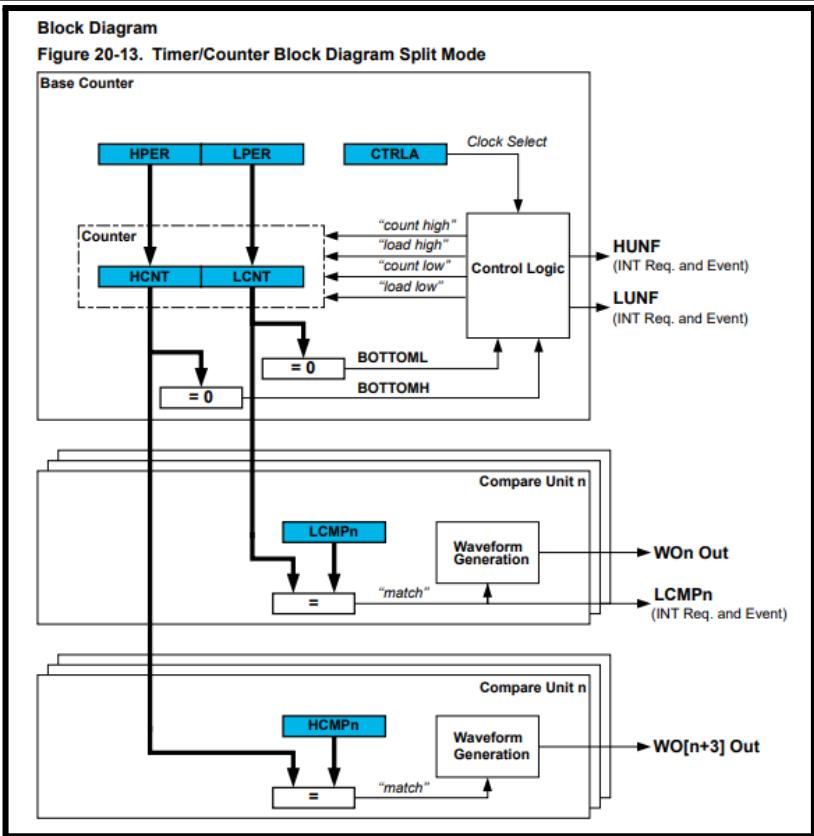
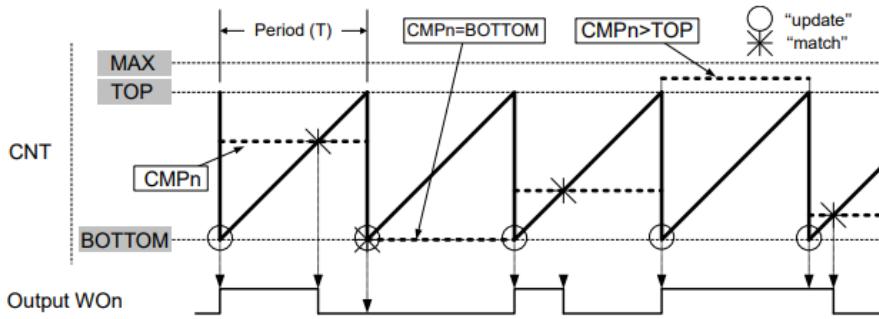


Figure 20-10. Single-Slope Pulse-Width Modulation



The TCA.n.PER register defines the PWM resolution. The minimum resolution is 2 bits (TCA.PER = 0x0002), and the maximum resolution is 16 bits (TCA.PER = MAX-1).

The following equation calculates the exact resolution in bits for single-slope PWM (R_{PWM_ss}):

$$R_{PWM_ss} = \frac{\log(PER+2)}{\log(2)}$$

The single-slope PWM frequency (f_{PWM_ss}) depends on the period setting (TCA.PER), the system's peripheral clock frequency f_{CLK_PER} and the TCA prescaler (CLKSEL in TCA CTRLA). It is calculated by the following equation where N represents the prescaler divider used:

$$f_{PWM_ss} = \frac{f_{CLK_PER}}{N(PER+1)}$$

Operation은 Single-Slope PWM 을 사용하였다. 20ms 주기로 동작하게 하기 위해 위 식을 통하여 PER를 312로 계산하고 사용하였는데, 8 bit Register 이기에 Overflow로 인해 56 이 PER 값으로 들어가게 되었는데, 처음 구현에서는 이 부분을 놓치고 구현하였다.

20.6 Register Summary - TCA in Split Mode

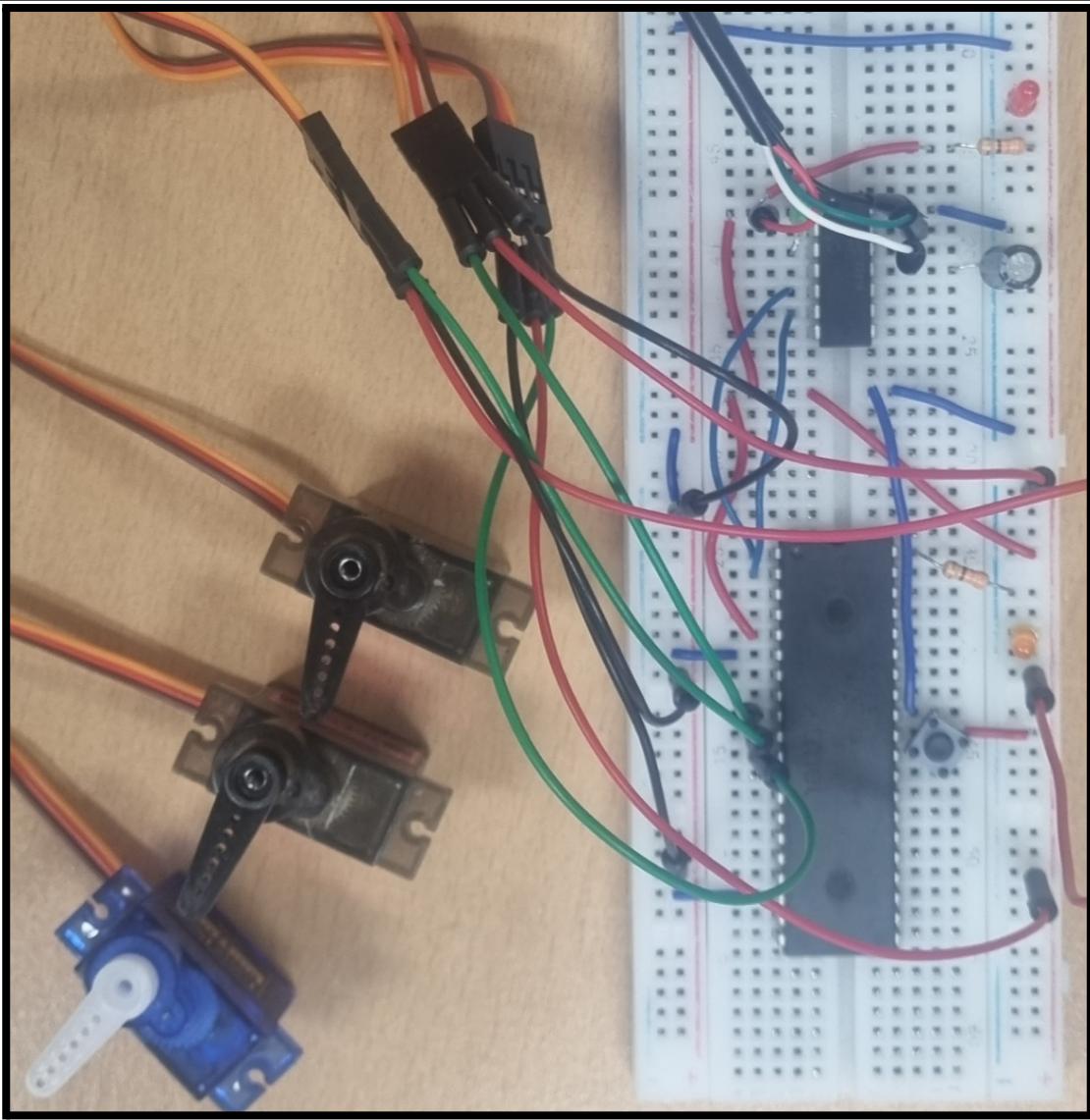
Offset	Name	Bit Pos.					CLKSEL[2:0]		ENABLE
0x00	CTRLA	7:0							
0x01	CTRLB	7:0		HCMP2EN	HCMP1EN	HCMP0EN		LCMP2EN	LCMP1EN
0x02	CTRLC	7:0		HCMP2OV	HCMP1OV	HCMP0OV		LCMP2OV	LCMP1OV
0x03	CTRLD	7:0							LCMP0OV
0x04	CTRLECLR	7:0					CMD[1:0]	CMDEN[1:0]	SPLITM
0x05	CTRLESET	7:0					CMD[1:0]	CMDEN[1:0]	
0x06	Reserved								
...									
0x09									
0x0A	INTCTRL	7:0		LCMP2	LCMP1	LCMP0		HUNF	LUNF
0x0B	INTFLAGS	7:0		LCMP2	LCMP1	LCMP0		HUNF	LUNF
0x0C	Reserved								
...									
0x0D									
0x0E	DBGCTRL	7:0							DBGRUN
0x0F	Reserved								
...									
0x1F									
0x20	LCNT	7:0					LCNT[7:0]		
0x21	HCNT	7:0					HCNT[7:0]		
0x22	Reserved								
...									
0x25									
0x26	LPER	7:0					LPER[7:0]		
0x27	HPER	7:0					HPER[7:0]		
0x28	LCMP0	7:0					LCMP[7:0]		
0x29	HCMP0	7:0					HCMP[7:0]		
0x2A	LCMP1	7:0					LCMP[7:0]		
0x2B	HCMP1	7:0					HCMP[7:0]		
0x2C	LCMP2	7:0					LCMP[7:0]		
0x2D	HCMP2	7:0					HCMP[7:0]		

```

void TCA_init() {
    *PORTMUX_TCA = 0x3;          // PWM Output : PD[0:5]
    *TCA_LPER   = B00111000;     // Low Byte PER
    *TCA_CTRLA  = B00001111;     // 0 | 0 | 0 | 0 | DIV1024 | EN
    *TCA_CTRLB  = B00000111;     // 0 | HCMP[0:2]EN | 0 | LCMP[0:2]EN
    *TCA_CTRLD  = B00000001;     // 0 | 0 | 0 | 0 | 0 | 0 | 0 | SPLITMODE EN
}

```

처음 구현에서 Overflow가 발생하여도 정상적으로 동작하여, PER 값을 56으로 설정하여 이와 같이 구현하였는데, 정상적으로 동작함을 확인하였다.



실제로 구현한 이후, PER 값을 잘못 계산하였기에, 주기는 우리가 원하는 20ms로 설정되지 않았다. 오실로스코프에서 측정한 결과 한 주기는 약 3.6us 정도로 측정되었는데, 이는 Data sheet 상에서 확인한 결과로는 동작이 되지 않아야 했으나, 정상적으로 동작되어 예상하지 못한 결과였다.

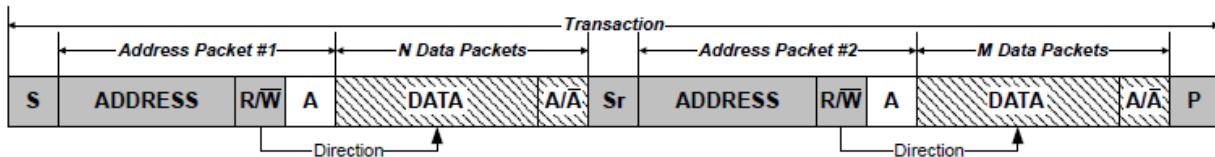
구현상에서 발견한 다른 문제점들은

1. 3개의 Servo motor 를 연결하였을 때, 전원이 불안정한 경우 motor 가 동작하지 않는 경우가 발생한다.
2. 같은 MG-90S가 같은 Input에서 다른 동작을 수행하는 경우가 발생한다. 제조사에 따라 다른 동작을 수행하는 경우가 발생하거나, 해당 Servo motor 자체의 고장으로 예상된다.

3. I2C

i2c 프로토콜의 구성은 다음과 같다.

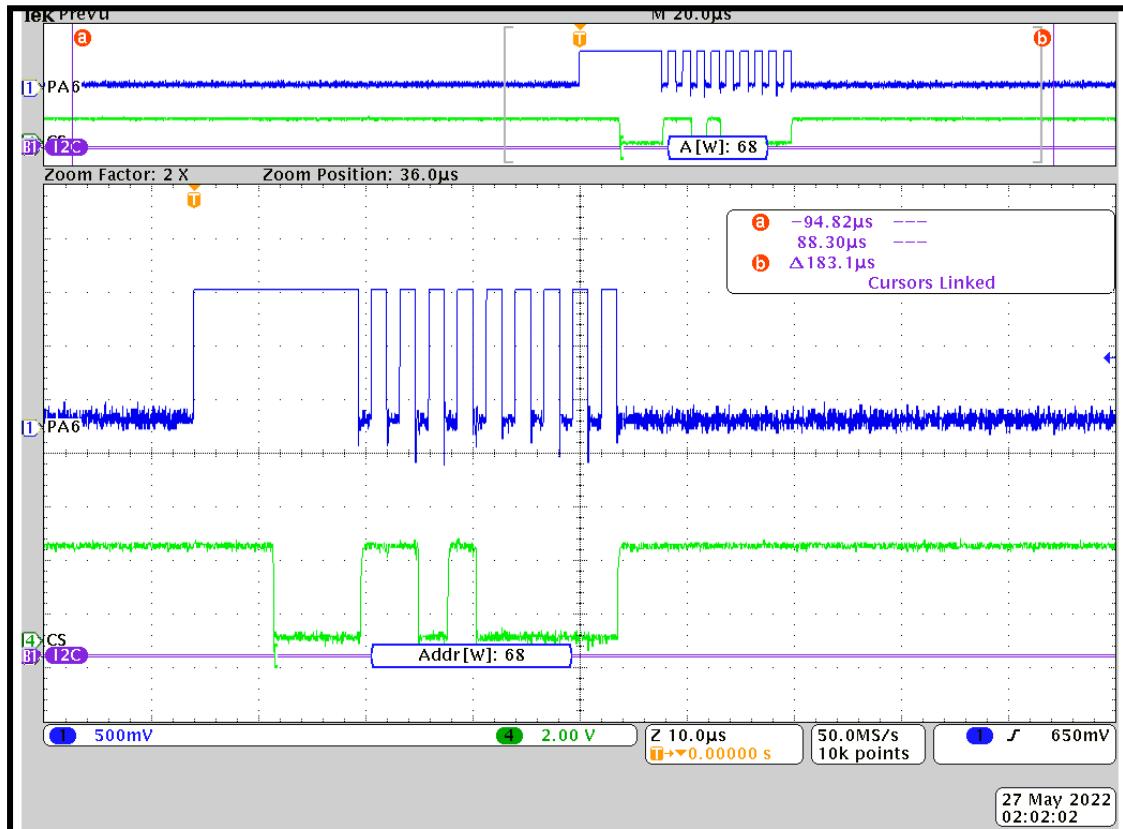
Figure 25-8. Combined Transaction



Read operation의 경우 Start+send slave address+write -> send register address -> restart + send slave

address+read -> read data 순으로 동작한다.

오실로스코프로 확인한 동작은 다음과 같다.



start signal 후에 slave address를 버스로 잘 전송하는 것을 확인할 수 있다.

i2c 프로토콜은 크게 5개의 함수를 사용한다.

I2C_0_init()

```
void I2C_0_init() {  
  
    TWI0_MBAUD = 30;  
    TWI0_MCTRLA = B00000001;  
    //TWI0_MSTATUS = B11000001;  
    TWI0.MCTRLB |= TWI_FLUSH_bm;  
    TWI0.MSTATUS |= (TWI_RIF_bm | TWI_WIF_bm);  
    TWI0_MSTATUS |= TWI_BUSSTATE_IDLE_gc;  
}
```

통신을 시작하기 전에 MBAUD 레지스터에 값을 할당해주어야 한다. MCTRLA 레지스터로 enable을 해준다.

MSTATUS 레지스터 값을 지정하여, flag enable을 하고, bus state를 idle로 설정해 준다.

I2C_0_start

```
uint8_t I2C_0_start(uint8_t baseAddress, uint8_t directionBit){  
    TWI0_MADDR = (baseAddress << 1) + directionBit;  
    while (!(TWI0_MSTATUS & (TWI_WIF_bm | TWI_RIF_bm))); //w  
    if (!(TWI0_MSTATUS & TWI_ARBLOST_bm)) return 0; //r  
    return !(TWI0_MSTATUS & TWI_RXACK_bm); //r  
}
```

MADDR에 값을 넣어주면 MDATA를 통해 bus로 전달된다. start를 위해서는 slave address와 write 를 start로 전송해주고, slave로부터 ack를 전달받은 경우에 while을 빠져나온다.

I2C_0_writingPacket

```
uint8_t I2C_0_writingPacket(uint8_t data)
{
    while (!(TWI0_MSTATUS & TWI_WIF_bm));
    TWI0_MDATA = data;
    TWI0_MCTRLB = TWI_MCMD_RECVTRANS_gc;
    return !(TWI0_MSTATUS & TWI_RXACK_bm);
}
```

write 명령은 write 동작이 실행되면 mdata에 값을 넣어줌으로써 bus로 전달된다.

I2C_0_receivingPacket

```
uint8_t I2C_0_receivingPacket(uint8_t acknack) // 0 -> ack, else nack
{
    while (!(TWI0_MSTATUS & TWI_RIF_bm)); //wait for read interrupt flag
    uint8_t data = TWI0_MDATA;
    if (acknack == 0) {TWI0_MCTRLB = (TWI_ACKACT_ACK_gc | TWI_MCMD_RECVTRANS_gc); }
    else {TWI0_MCTRLB = (TWI_ACKACT_NACK_gc | TWI_MCMD_RECVTRANS_gc); }
    return data;
}
```

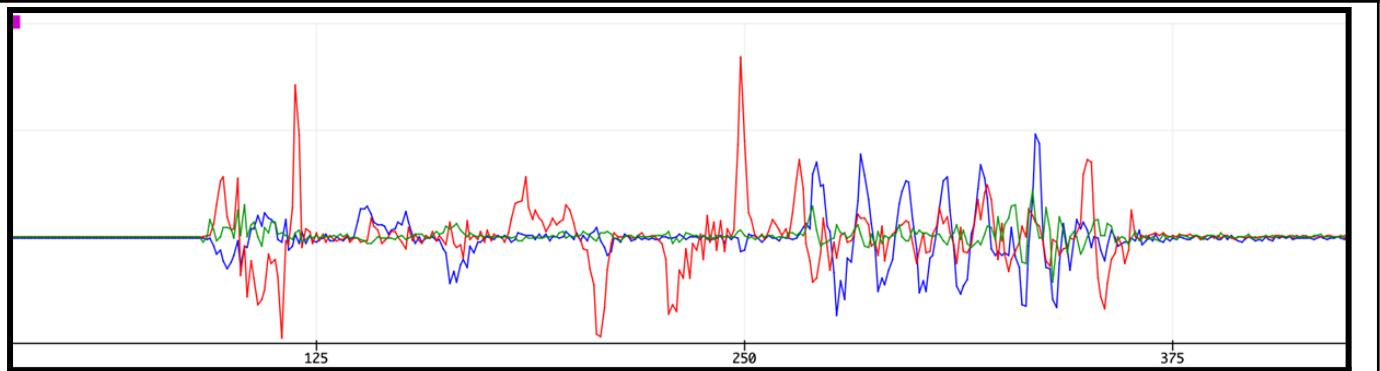
기능은 다음과 같다. MDATA 레지스터에 들어있는 값을 내부 변수 data에 입력한다. 이때 이 함수의 입력이 0인 경우에는 수신 후 버스에 ack를 전달한다. 이 함수의 입력이 1인 경우에 수신 후 버스에 nack을 전달한다.

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

위 사진은 accx, accy, accz, temp, gyrox, gyroy, gyroz 를 읽어오기 위한 i2c burst read sequence 프로토콜이다. 연달아 값을 읽어오기 위해 읽어온 후 master가 ack를 전송하고, 마지막 수신일 경우에만 nack을 전송한다. 해당 프로토콜에 맞춰 코드를 작성하였고, 값이 출력되는 것을 확인하였다.

```
I2C_0_start(0x68, I2C_DIRECTION_BIT_WRITE);
I2C_0_writingPacket(0x3B);
I2C_0_start(0x68, I2C_DIRECTION_BIT_READ);
raw_Acc_x = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_Acc_y = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_Acc_z = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_temp = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_Gyro_x = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_Gyro_y = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(0);
raw_Gyro_z = (I2C_0_receivingPacket(0) << 8) | I2C_0_receivingPacket(1);
```



다음은 각각의 **raw** 데이터를 전송받아서 그래프화 시킨 예시이다. 해당 그래프는 자이로센서의 **raw data**이다. mpu6050이 탑재되어 있는 보드를 움직였을 때, 해당 동작에 맞춰서 변화하는 것을 확인하였다.

4. Raw data 를 이용한 각도 산출

accelerometer의 데이터로 각도를 산출하는 방법은 다음과 같다.

중력방향을 기준으로 일정한 중력 가속도가 측정이 되고 있기 때문에, x, y축으로 이 중력가속도가 분배되는 정도를 계산하여, 보드가 기울어진 각도를 산출할 수 있다.

gyroscensor의 데이터로 각도를 산출하는 방법은 다음과 같다.

해당 축을 기준으로 회전한 정도에 대한 각속도가 측정되기 때문에, 이를 바탕으로 각 축을 기준으로 회전한 정도를 계산할 수 있다.

우리가 사용하는 장치의 경우, 장치의 움직임이 일정하므로, 가속도의 변화가 크지 않을 것으로 가정하고 상보필터의 비율을 정할 때 가속도센서로부터 측정한 각도를 주로 반영하도록 하였다. 이는 자이로센서로부터 각도를 산출하는 경우 가속도센서의 드리프트가 누적되기 때문에, 오차를 최소화하기 위함이다.

```
angle_x = (GyroCoef * (angle_x + angle_Gyro_x)) + ((1 - GyroCoef) * angle_Acc_x);
angle_y = (GyroCoef * (angle_y + angle_Gyro_y)) + ((1 - GyroCoef) * angle_Acc_y);
angle_z = angle_Gyro_z;
```

이 경우 gyrocoef는 0.02로 거의 반영하지 않았다.

다음으로는 자이로센서의 드리프트를 보정하는 알고리즘을 작성하였다.

Yaw 각도의 경우 자이로센서로부터 계산할 수밖에 없는데, 이는 yaw데이터의 **drift**가 발생할 수밖에 없음을 의미한다. 따라서 기기가 정지할 때마다, 드리프트를 보정해주는 알고리즘을 작성하였다.

```

if(driftnum == 0) {
    values[driftnum] = angle_Gyro_z;
    driftnum = 1;
}
else if(driftnum >= 1){
    values[driftnum] = angle_Gyro_z;
    diffs[driftnum-1] = values[driftnum] - values[driftnum-1];
    if(driftnum == 50) {
        float sum = 0;
        float avg = 0;
        float avgcal = 0;
        driftnum = 0;
        for(int k = 0; k<49; k++) {
            sum = sum + diffs[k];
        }
        avg = sum/49;
        avgcal = (diffs[0]+diffs[49])/2;
        if(((avgcal - avg)*(avgcal - avg))<=0.01) {
            mySerial.println("stop moving");
            drifterror = avg;
            mySerial.println(drifterror);
        }
    }
    else driftnum++;
}

```

동작은 다음과 같다. 매 50 개의 데이터를 비교하고 해당 데이터의 변화량을 저장한다. 변화량이 일정하다는 것은 기기의 움직임은 없고, 드리프트에 의한 증감만 일어나고 있음을 의미한다. 따라서 정지상태에서 일정한 변화량을 드리프트로 간주하고, 해당 드리프트를 보정해줄 수 있도록 하였다.

```

15:42:05.515 -> -2.47 aftercali -0.75
15:42:05.656 -> 0 // 0
15:42:05.656 -> -2.47 aftercali -0.75
15:42:05.798 -> 0 // 0
15:42:05.845 -> -2.47 aftercali -0.74
15:42:05.984 -> 0 // 0
15:42:05.984 -> -2.47 aftercali -0.75

```

```

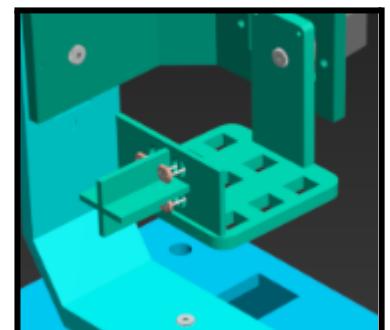
15:45:42.861 -> -6.48 aftercali -0.78
15:45:43.048 -> 0 // 0
15:45:43.048 -> -6.48 aftercali -0.78
15:45:43.188 -> 0 // 0
15:45:43.188 -> -6.49 aftercali -0.78
15:45:43.375 -> 0 // 0
15:45:43.375 -> -6.50 aftercali -0.78
15:45:43.518 -> 0 // 0
15:45:43.518 -> -6.50 aftercali -0.78
15:45:43.707 -> 0 // 0
15:45:43.707 -> -6.51 aftercali -0.79

```

위 스크린샷은 3분동안 정지상태에서 드리프트를 측정하고, 알고리즘 적용 여부에 따른 드리프트 양을 비교한 사진이다. 3분동안 알고리즘이 적용된 경우는 0.03의 드리프트가 측정되었고, 알고리즘을 적용하지 않은 경우 4 가량의 드리프트가 측정되었다. 드리프트의 경우, 센서 측정값이 크고, 동작이 많을 수록 커지는 경향을 보이므로, 실제 동작시에는 보정 여부가 결과에 큰 영향을 끼칠 것이라고 생각한다.

5. CDS의 Offset 조정

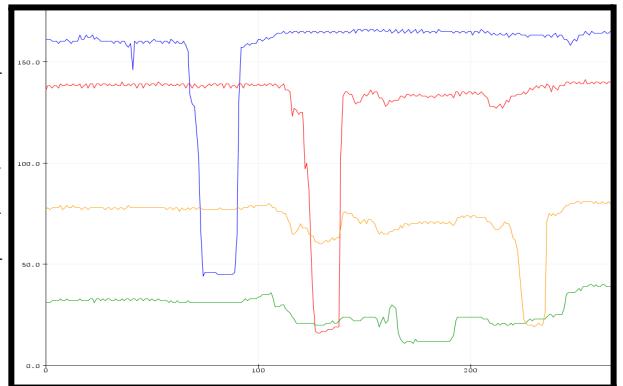
4개의 CDS가 하늘을 바라보는 것이 아닌 측면을 바라보도록 배치하였으며, 광원의 위치를 확실하게 인식하게 위해 2개의 가림막을 이용해 각 센서를 독립시키는 형태로 모델을 구현하였다.



광원에 특정 위치에 있다면 광원에 제일 근접한 CDS는 가장 큰 데이터를 얻을

것이고 나머지 CDS들은 가림막에 의해 그림자가 져서 낮은 데이터를 얻게 됨으로 데이터의 차이를 이용하여 광원을 따라 Tracking 하는 방식을 고민했다.

우측의 그림과 같은 배치를 하게 되어서 발생한 문제점이 설정한 광원이 없음에도 불구하고 형광등이나 자연광 등 환경적인 요인에 상단과 하단의 CDS들이 큰 차이를 유지하게 되는 것이었다. 우측의 시리얼 플로터로 CDS들의 raw data를 확인해본 결과이다. 파란색과 빨간색이 상단의 CDS들, 노란색과 초록색이 하단의 CDS들의 데이터를 보여준다.



이러한 문제점을 해결하기 위한 방법으로 생각한 것이 각 CDS로부터 초기 값들을 저장하여 이 값들의 평균을 내어 offset을 구하는 것이다. offset을 구한 이후에 받게 되는 raw data에서 offset을 빼서 어느 정도 data를 보정하는 것이다.

```

if (count < arrsize) {
    ADC_Read_Value();

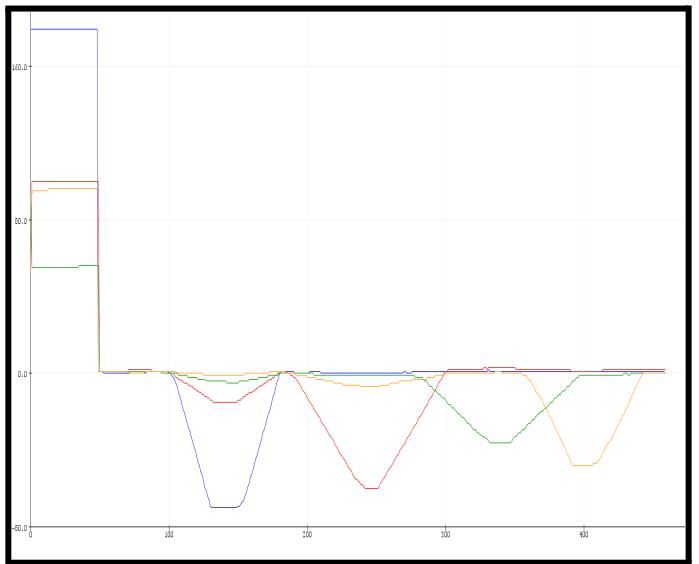
    sum1 += L_top;
    sum2 += R_top;
    sum3 += L_bottom;
    sum4 += R_bottom;

    count++;
}

if (count == arrsize) {
    L_top_offset = sum1 / arrsize;
    R_top_offset = sum2 / arrsize;
    L_bottom_offset = sum3 / arrsize;
    R_bottom_offset = sum4 / arrsize;
    offset_complete_flag = 1;
}

ADC_Read_Value();

```



offset을 보정하는 데 시간이 조금 걸린다는 단점이 있기는 하지만 설정한 광원이 아닌 주위의 빛에 의한 영향을 없앨 수 있고 이를 플로터로 확인할 수 있었다. 모든 CDS가 일정한 값을 보이는 것을 볼 수 있으며 각각의 CDS가 하나씩 가려보면서 그림자가 졌을 상황을 가정했을 때 데이터의 값도 정상적으로 얻을 수 있는지 확인해 보았다.

```

if (offset_complete_flag) {
    vert_status = 0;
    horiz_status = 0;
    int16_t top_avg = ((L_top - L_top_offset) + (R_top - R_top_offset)) / 2;
    int16_t bottom_avg = ((L_bottom - L_bottom_offset) + (R_bottom - R_bottom_offset)) / 2;
    int16_t left_avg = ((L_top - L_top_offset) + (L_bottom - L_bottom_offset)) / 2;
    int16_t right_avg = ((R_top - R_top_offset) + (R_bottom - R_bottom_offset)) / 2;

    bool mov_vert_flag = (abs(top_avg - bottom_avg) > threshold) ? true : false;
    bool mov_horiz_flag = (abs(left_avg - right_avg) > threshold) ? true : false;

    if (mov_vert_flag) {
        if (top_avg > bottom_avg) vert_status = 1;
        else vert_status = -1;
    }
    if (mov_horiz_flag) {
        if (left_avg > right_avg) horiz_status = 1;
        else horiz_status = -1;
    }
}

```

Offset 보정이 완료되면 flag가 생성되어 4개의 CDS를 상하좌우 방향에 따라 2개씩 묶어 각 방향에 대한 평균 값을 산출하였다. 산출 값을 이용하여 각 방향의 차이가 설정한 임계값을 초과하게 되면 수직/수평 방향의 움직임에 대한 flag를 생성하게 된다.

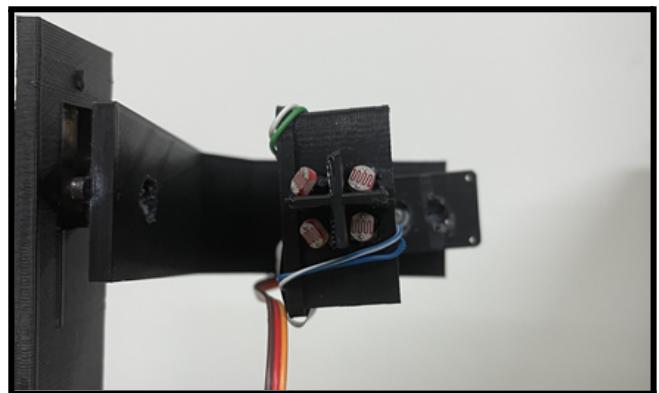
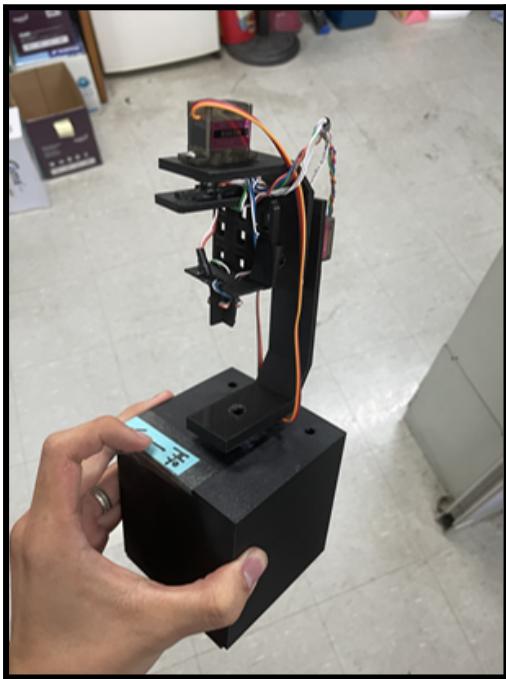
```
if (horiz_status == 1) {  
    fin_z = fin_z - 1;  
}  
else if (horiz_status == -1) {  
    fin_z = fin_z + 1;  
}  
else fin_z = fin_z;  
if (vert_status == 1) {  
    fin_x = fin_x - 1;  
}  
else if (vert_status == -1) {  
    fin_x = fin_x + 1;  
}  
else fin_x = fin_x;
```

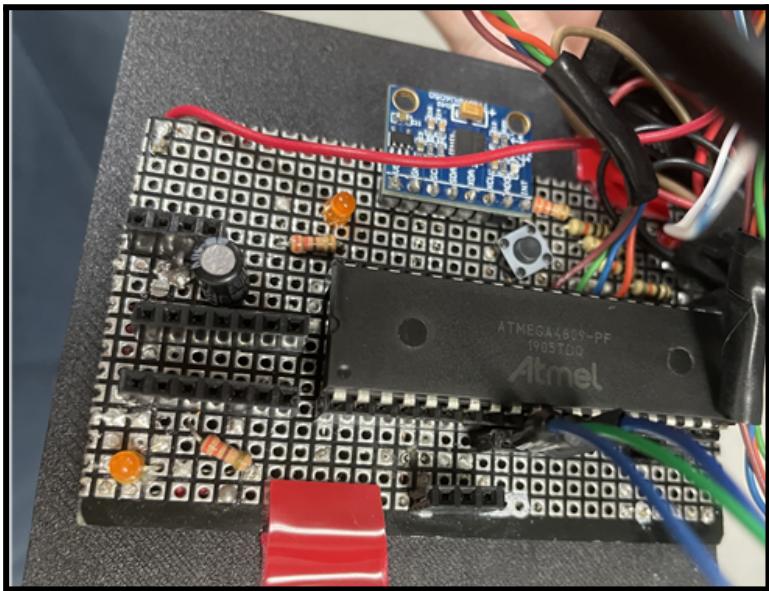
그래서 각 flag에 따라 광원이 있는 방향으로 1도씩 움직일 수 있도록 데이터를 누적시키는 코드를 구현했다.

```
*TCA_LCMP0 = move_Servo(-angle_z + fin_z);  
*TCA_LCMP1 = move_Servo(angle_x + fin_x);  
*TCA_LCMP2 = move_Servo(-angle_y);
```

이렇게 해서 산출한 tracking에 필요한 각도를 자세 제어를 위한 각도에 반영하여 두 가지 기능을 함께 구현하도록 했다.

6. 회로 제작 및 모델 조립





완성한 장치는 다음과 같다. 헤드 부분에 cds 4개를 부착하였고, 각각의 데이터 판단이 용이하도록 가림막을 통해 확실히 분리해 주었다.

만능기판은 장치 내부에 들어갈 수 있는 사이즈로 제작하였다.

최종적으로 구현 목표를 만족하도록 동작하는 것을 확인하였다.

작성자	일자 2022년 5월 29일	확인자	일자 년 월 일
	서명 윤대민 김태완 이재영		서명