

Microprocessor Lab Report

Lab 14: Device Driver

소속: 컴퓨터정보공학부

학번: 2019202053

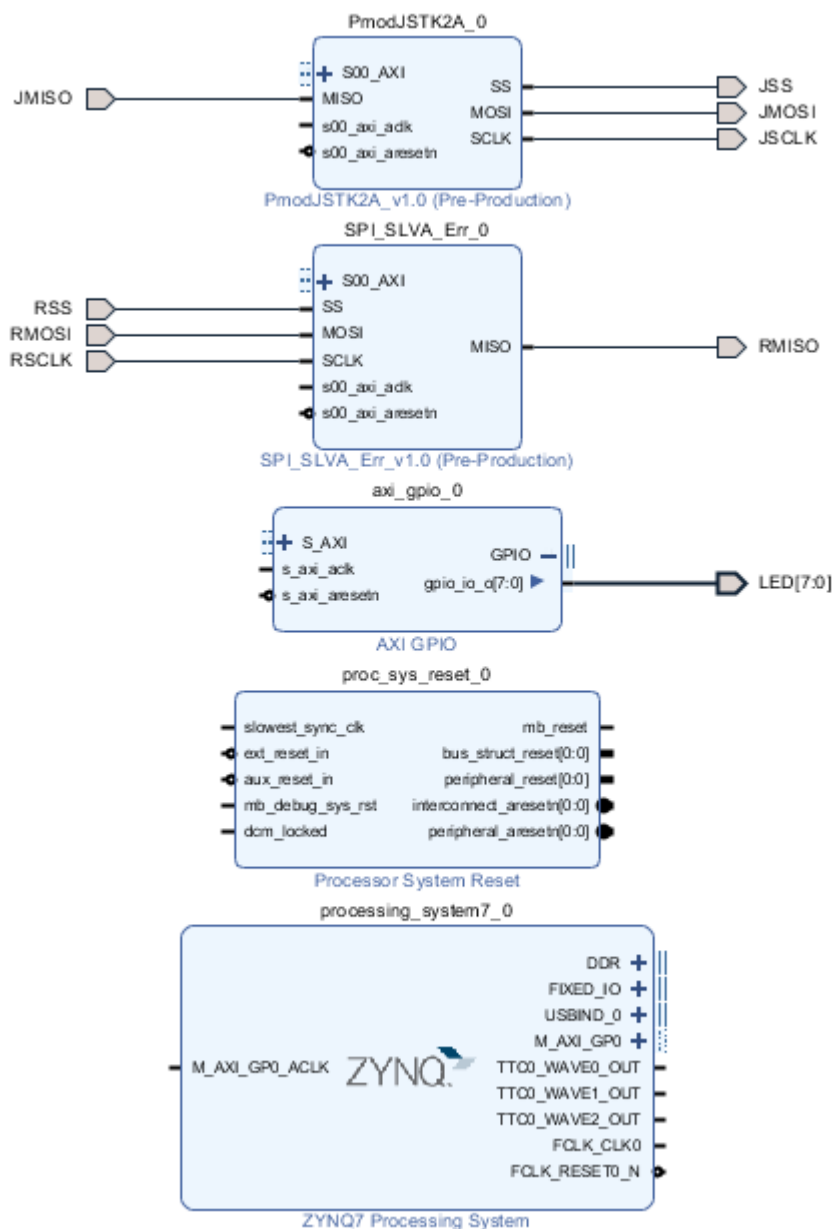
이름: 신윤석



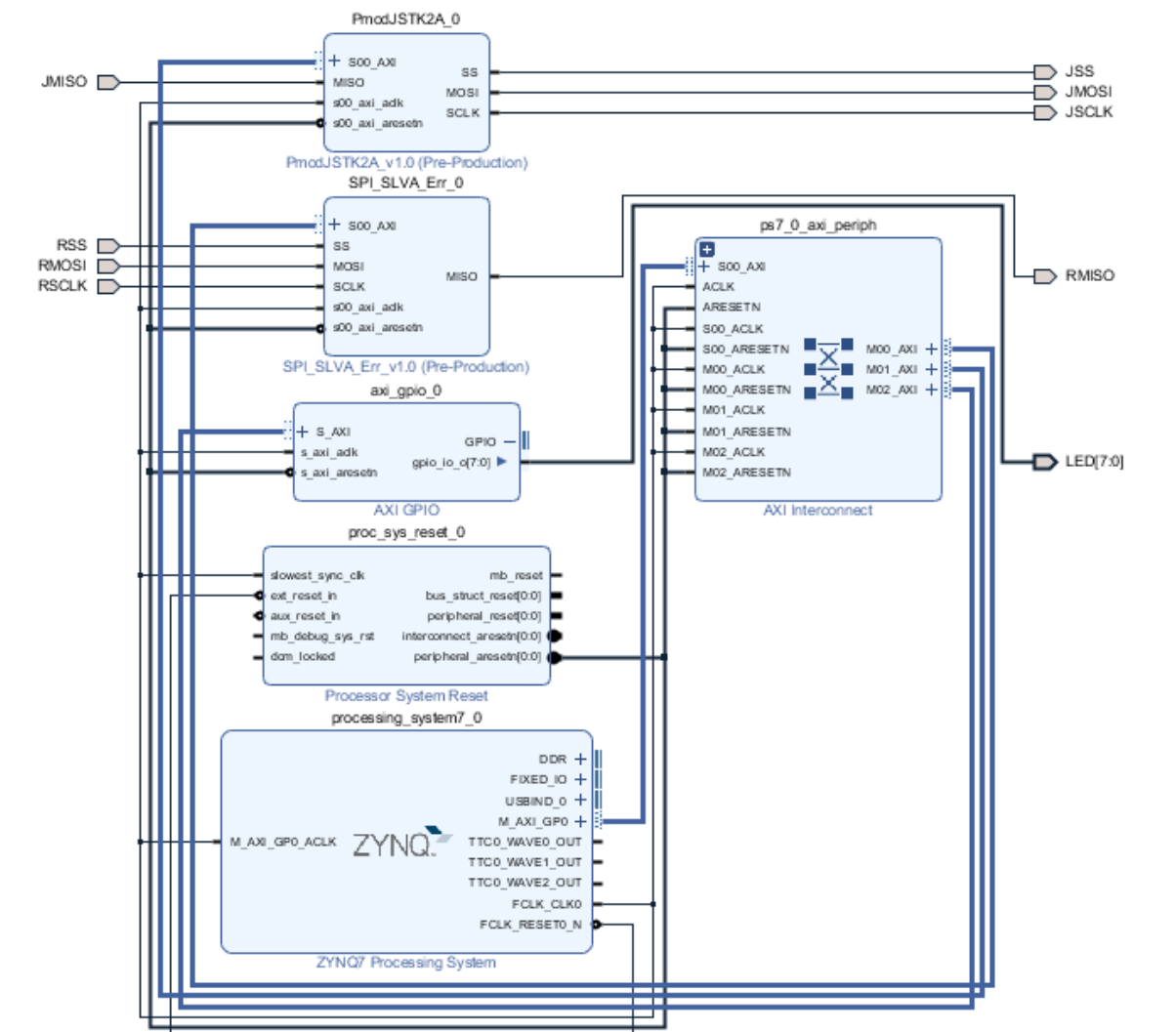
1. 문제의 해석 및 해결 방향

이번 과제에서는 ZYNQ starter board 를 이용하여 AXI bus 에 조이스틱과 nano board 를 붙이고, GPIO 로 보드 내에 있는 LED 에 연결하여 조이스틱에서 얻은 사용자 입력을 nano board 로 옮기고 nano board 에서 다시 신호를 처리해서 nano board 가 board 의 AXI 버스를 통해 ZYNQ 보드 위에 있는 LED 와 조이스틱에 있는 LED 를 제어하고 nano board 에 SPI 로 붙어있는 7-segment 까지 제어해 본다.

우선 다음과 같이 vivado 에 모듈을 작성해준다.



그리고 connection automation 을 실행하면 다음과 같아진다.



다음은 constraint 를 통해 핀에 대한 정보를 입력한다.

```
1  # -----
2  # JA Pmod - Bank 13
3  # -----
4  set_property PACKAGE_PIN V11 [get_ports {JSS}]; # "JA1"
5  set_property PACKAGE_PIN AA11 [get_ports {JMOSI}]; # "JA2"
6  set_property PACKAGE_PIN V10 [get_ports {JMISO}]; # "JA3"
7  set_property PACKAGE_PIN AA9 [get_ports {JSCLK}]; # "JA4"
8  #set_property PACKAGE_PIN AB11 [get_ports {JA7}]; # "JA7"
9  #set_property PACKAGE_PIN AB10 [get_ports {JA8}]; # "JA8"
10 #set_property PACKAGE_PIN AB9 [get_ports {JA9}]; # "JA9"
11 #set_property PACKAGE_PIN AA8 [get_ports {JA10}]; # "JA10"
12
13
14 # -----
15 # JB Pmod - Bank 13
16 # -----
17 set_property PACKAGE_PIN W12 [get_ports {RSS}]; # "JB1"
18 set_property PACKAGE_PIN W11 [get_ports {RMOSI}]; # "JB2"
19 set_property PACKAGE_PIN V10 [get_ports {RMISO}]; # "JB3"
20 set_property PACKAGE_PIN W8 [get_ports {RCLK}]; # "JB4"
21 #set_property PACKAGE_PIN V12 [get_ports {JB7}]; # "JB7"
22 #set_property PACKAGE_PIN W10 [get_ports {JB8}]; # "JB8"
23 #set_property PACKAGE_PIN V9 [get_ports {JB9}]; # "JB9"
24 #set_property PACKAGE_PIN V8 [get_ports {JB10}]; # "JB10"
25
26 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets {RCLK}];
27
28
29 # -----
30 # User LEDs - Bank 33
31 # -----
32 set_property PACKAGE_PIN T22 [get_ports {LED[0]}]; # "LD0"
33 set_property PACKAGE_PIN T21 [get_ports {LED[1]}]; # "LD1"
34 set_property PACKAGE_PIN U22 [get_ports {LED[2]}]; # "LD2"
35 set_property PACKAGE_PIN U21 [get_ports {LED[3]}]; # "LD3"
36 set_property PACKAGE_PIN V22 [get_ports {LED[4]}]; # "LD4"
37 set_property PACKAGE_PIN W22 [get_ports {LED[5]}]; # "LD5"
38 set_property PACKAGE_PIN U19 [get_ports {LED[6]}]; # "LD6"
39 set_property PACKAGE_PIN U14 [get_ports {LED[7]}]; # "LD7"
40
41
42 # -----
43 # IOSTANDARD Constraints
44 # -----
```

```

40 |
41 |
42 | # -----
43 | # IOSTANDARD Constraints
44 | # -----
45 |
46 | # Note that the bank voltage for IO Bank 33 is fixed to 3.3V on ZedBoard.
47 | set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 33]];
48 |
49 | # Note that the bank voltage for IO Bank 13 is fixed to 3.3V on ZedBoard.
50 | set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 13]];
51 |

```

```

# -----
# JA Pmod - Bank 13
# -----

set_property PACKAGE_PIN Y11 [get_ports {JSS}]; # "JA1"
set_property PACKAGE_PIN AA11 [get_ports {JMOSI}]; # "JA2"
set_property PACKAGE_PIN Y10 [get_ports {JMISO}]; # "JA3"
set_property PACKAGE_PIN AA9 [get_ports {JSCLK}]; # "JA4"
#set_property PACKAGE_PIN AB11 [get_ports {JA7}]; # "JA7"
#set_property PACKAGE_PIN AB10 [get_ports {JA8}]; # "JA8"
#set_property PACKAGE_PIN AB9 [get_ports {JA9}]; # "JA9"
#set_property PACKAGE_PIN AA8 [get_ports {JA10}]; # "JA10"


# -----
# JB Pmod - Bank 13
# -----

set_property PACKAGE_PIN W12 [get_ports {RSS}]; # "JB1"
set_property PACKAGE_PIN W11 [get_ports {RMOSI}]; # "JB2"
set_property PACKAGE_PIN V10 [get_ports {RMISO}]; # "JB3"
set_property PACKAGE_PIN W8 [get_ports {RSCLK}]; # "JB4"
#set_property PACKAGE_PIN V12 [get_ports {JB7}]; # "JB7"
#set_property PACKAGE_PIN W10 [get_ports {JB8}]; # "JB8"
#set_property PACKAGE_PIN V9 [get_ports {JB9}]; # "JB9"
#set_property PACKAGE_PIN V8 [get_ports {JB10}]; # "JB10"


set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets {RSCLK}];

```

```

# -----
# User LEDs - Bank 33
# -----
set_property PACKAGE_PIN T22 [get_ports {LED[0]}}; # "LD0"
set_property PACKAGE_PIN T21 [get_ports {LED[1]}}; # "LD1"
set_property PACKAGE_PIN U22 [get_ports {LED[2]}}; # "LD2"
set_property PACKAGE_PIN U21 [get_ports {LED[3]}}; # "LD3"
set_property PACKAGE_PIN V22 [get_ports {LED[4]}}; # "LD4"
set_property PACKAGE_PIN W22 [get_ports {LED[5]}}; # "LD5"
set_property PACKAGE_PIN U19 [get_ports {LED[6]}}; # "LD6"
set_property PACKAGE_PIN U14 [get_ports {LED[7]}}; # "LD7"

# -----
# IOSTANDARD Constraints
# -----

# Note that the bank voltage for IO Bank 33 is fixed to 3.3V on ZedBoard.
set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 33]];

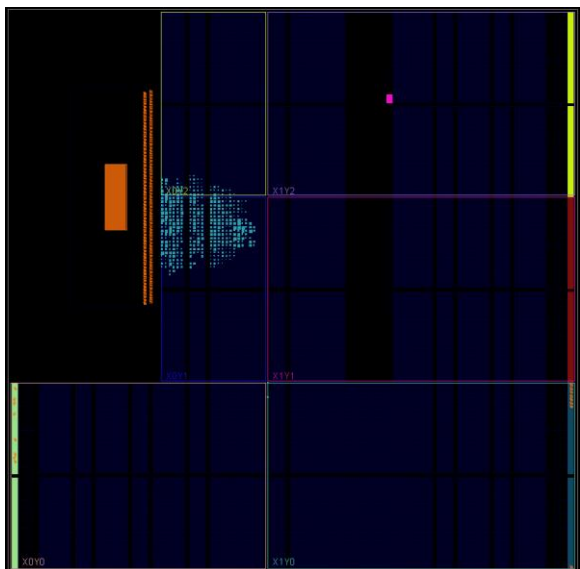
# Note that the bank voltage for IO Bank 13 is fixed to 3.3V on ZedBoard.
set_property IOSTANDARD LVCMOS33 [get_ports -of_objects [get_iobanks 13]];

```

과제 추가자료로 있는 자료를 그대로 사용하였음을 밝힌다.

그리고나서 synthesis, Implementation 을 하고 bitstream 을 만들어준다.


다음은 implementation 결과이다.



그 다음 hardware 를 export 하고 SDK 를 켜서 소프트웨어 코딩을 진행한다.

그에 앞서 address editor 를 확인하기 위해 AXI 연결 번호를 확인해 보면 위의 block automation 까지 완료된 block diagram 을 보면 AXI 0 번에 나노 보드와 연결된 모듈을 확인할 수 있고, AXI 1 번에 조이스틱과 연결된 모듈을 확인할 수 있다. 다음은 address editor 를 확인해 보면 다음과 같다.

🔧 processing_system7_0

✓  Data (32 address bits : 0x40000000 [1G])

▣ PmodJSTK2A_0	S00_AXI	S00_AXI_reg	0x43C1_0000	64K ▼	0x43C1_FFFF
▣ SPI_SLVA_Err_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K ▼	0x43C0_FFFF
▣ axi_gpio_0	S_AXI	Reg	0x4120_0000	64K ▼	0x4120_FFFF

조이스틱으로 가는 주소는 0x43C1_0000 이고 나노 보드로 가는 주소는 0x43C0_0000 임을 확인할 수 있다.

다음으로는 SDK 를 열어서 C 코드를 작성해준다. 이 코드는 조이스틱의 값을 받아서 나노 보드로 옮기고 나노 보드로부터 제어 값을 받아서 [7:0] bit 중 [0]bit 를 조이스틱 LED 제어값으로 보내고, [15:8]bit 는 ZYNQ 보드 내 LED 제어에 사용하는 값이다. 다음은 코드이다.


```

⊕ * rbpi_jstk.c: simple test application

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xil_io.h"

#define JSTK_AXI_IP_REG0 0x43C10000 // AXI address for joystick
#define JSTK_AXI_IP_REG1 0x43C10004 // AXI address + 4

#define NANO_AXI_IP_REG0 0x43C00000 // AXI address for Jetson Nano
#define NANO_AXI_IP_REG1 0x43C00004 // AXI address + 4

int main()
{
    // define GPIO object variables and user variables
    // GOIO is used for ZYNQ LEDs, and user variables are used for two AXI buses,
    // one for the joystick and the other for Jetson nano
    XGpio gpio_obj; // for ZYNQ LEDs
    unsigned int var; // for AXI read/write

    // initialize platform
    init_platform();

    // Initialize GPIOs. Should match to the instance
    XGpio_Initialize(&gpio_obj, XPAR_AXI_GPIO_0_DEVICE_ID);

    // Direction Register (gpio_obj, channel=1, i/o: input is 1, output is 0)
    XGpio_SetDataDirection(&gpio_obj, 1, 0);

    print("we are up\n");

    // Initialize your local data here if you have to

    while (1) { // do forever
        // your main code here.
        // If you keep the same data format as the joystick data from the NANO command,
        // you don't need to change data format in this program.

        // read joystick data, then send to Nano
        var = Xil_In32(JSTK_AXI_IP_REG0);
        Xil_Out32(NANO_AXI_IP_REG0, var);
        // read Nano data, then send to joystick
        var = Xil_In32(NANO_AXI_IP_REG1);
        if((var & 0xFFFF0000) != 0xFFFF0000) continue;
        Xil_Out32(JSTK_AXI_IP_REG1, var & 1);
        // read Nano data, then send to ZYNQ LEDs
        //var = Xil_In32(NANO_AXI_IP_REG0);
        var >>= 8;
        XGpio_DiscreteWrite(&gpio_obj, 1, var);
    }

    cleanup_platform();
    return 0;
}

```

```
/*
```

```

* rbpi_jstk.c: simple test application
*/

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xil_io.h"

#define JSTK_AXI_IP_REG0 0x43C10000 // AXI address for joystick
#define JSTK_AXI_IP_REG1 0x43C10004 // AXI address + 4

#define NANO_AXI_IP_REG0 0x43C00000 // AXI address for Jetson Nano
#define NANO_AXI_IP_REG1 0x43C00004 // AXI address + 4

int main()
{
    // define GPIO object variables and user variables
    // GOIO is used for ZYNQ LEDs, and user variables are used for two AXI buses,
    // one for the joystick and the other for Jetson nano
    XGpio gpio_obj; // for ZYNQ LEDs
    unsigned int var; // for AXI read/write

    // initialize platform
    init_platform();

    // Initialize GPIOs. Should match to the instance
    XGpio_Initialize(&gpio_obj, XPAR_AXI_GPIO_0_DEVICE_ID);

    // Direction Register (gpio_obj, channel=1, i/o: input is 1, output is 0)
    XGpio_SetDataDirection(&gpio_obj, 1, 0);

    print("we are up\n");

    // Initialize your local data here if you have to

    while (1) { // do forever
        // your main code here.
    }
}

```

```

// If you keep the same data format as the joystick data from the NANO command,
// you don't need to change data format in this program.

// read joystick data, then send to Nano
var = Xil_In32(JSTK_AXI_IP_REG0);
Xil_Out32(NANO_AXI_IP_REG0, var);
// read Nano data, then send to joystick
var = Xil_In32(NANO_AXI_IP_REG1);
if((var&0xFFFF0000) != 0xFFFF0000) continue;
Xil_Out32(JSTK_AXI_IP_REG1, var & 1);
// read Nano data, then send to ZYNQ LEDs
var >>= 8;
XGpio_DiscreteWrite(&gpio_obj, 1, var);
}

cleanup_platform();
return 0;
}

```

위 코드에서 특징적인 부분은 나노 보드로부터 제어 값을 읽을 때 [31:16]부분이 FFFF 인지를 검사해서 FFFF 가 아니면 무시하는 행동을 한다 이 이유는 후에 문제 해결부분에서 후술한다.

다음으로는 나노 보드에서 돌아가는 파이썬 코드를 작성한다.

```

import spidev
import time

#initializing spi bus
spiToZync = spidev.SpiDev()
spiToZync.open(0,0)
spiToZync.max_speed_hz = 5000
spiToZync.mode = 0b00

spiToSeg = spidev.SpiDev()
spiToSeg.open(0,1)
spiToSeg.max_speed_hz = 5000
spiToSeg.mode = 0b00

refreshPeriod = 0.25    #chage this! the period to refresh the data printed
toggle = 1  #this represents the integer "0x00000001" byte by byte, so this list has 4
components

transferList = spiToZync.readbytes(4)  #read the 4 byte for int

```

```

read = ((transferList[0] << 24)&0xFF000000) | ((transferList[1] << 16)&0xFF0000) |
((transferList[2] << 8)&0xFF00) | (transferList[3]&0xFF)    #make a int type data by four
byte data
while True:
    #parsing read data to x and y coordinate data and button click data
    y = read & 0x3FF
    x = (read >> 10) & 0x3FF
    btn = (read >> 20) & 0x3
    ToJSTK = read >> 20 #set ToJSTK variable with the data from joy stick's button bits
    #set ToJSTK by LED control signals
    if x > 640:
        ToJSTK |= 1 << 7

    if x < 384:
        ToJSTK |= 1 << 6

    if y > 640:
        ToJSTK |= 1 << 5

    if y < 384:
        ToJSTK |= 1 << 4

    #transfer the data with ZYNQ part
    transferList = [0xff,0xff,ToJSTK,toggle]    #0xff, 0xff at [31:16] is important part
    spiToZync.writebytes(transferList) #write the control data to ZYNQ board
    transferList = spiToZync.readbytes(4)    #read the data from ZYNQ board
    read = ((transferList[0] << 24)&0xFF000000) | ((transferList[1] << 16)&0xFF0000) |
((transferList[2] << 8)&0xFF00) | (transferList[3]&0xFF)    #make a int type data by four
byte data

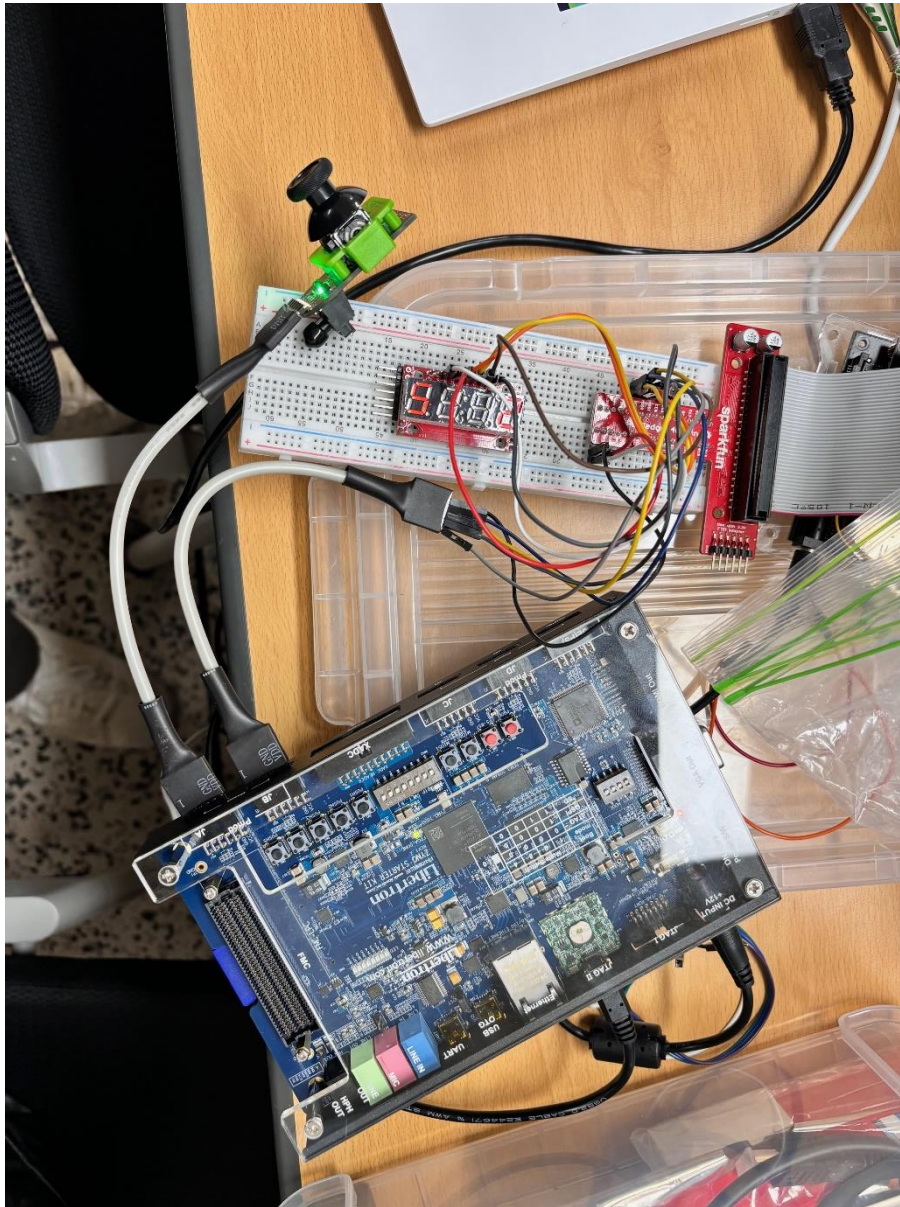
    #print X, Y coordinate and Button click info, toggle info, which means the lights at
jstk turn on or not
    print("X: %04dY: %04dB: %01d%01dttoggle:%d"%(x,y,btn>>1,btn&1,toggle))
    #change the toggle state
    if toggle == 1:
        toggle = 0
    else:
        toggle = 1
    #write the control data to blink the LED on jstk
    x = x / 10
    y = y / 10
    time.sleep(refreshPeriod)
    spiToSeg.writebytes([0x76,int(x/10),int(x%10),int(y/10),int(y%10)])
#    spiToSeg.writebytes([0x76,1,2,3,4])
    time.sleep(refreshPeriod)    #wait for the next read time

```

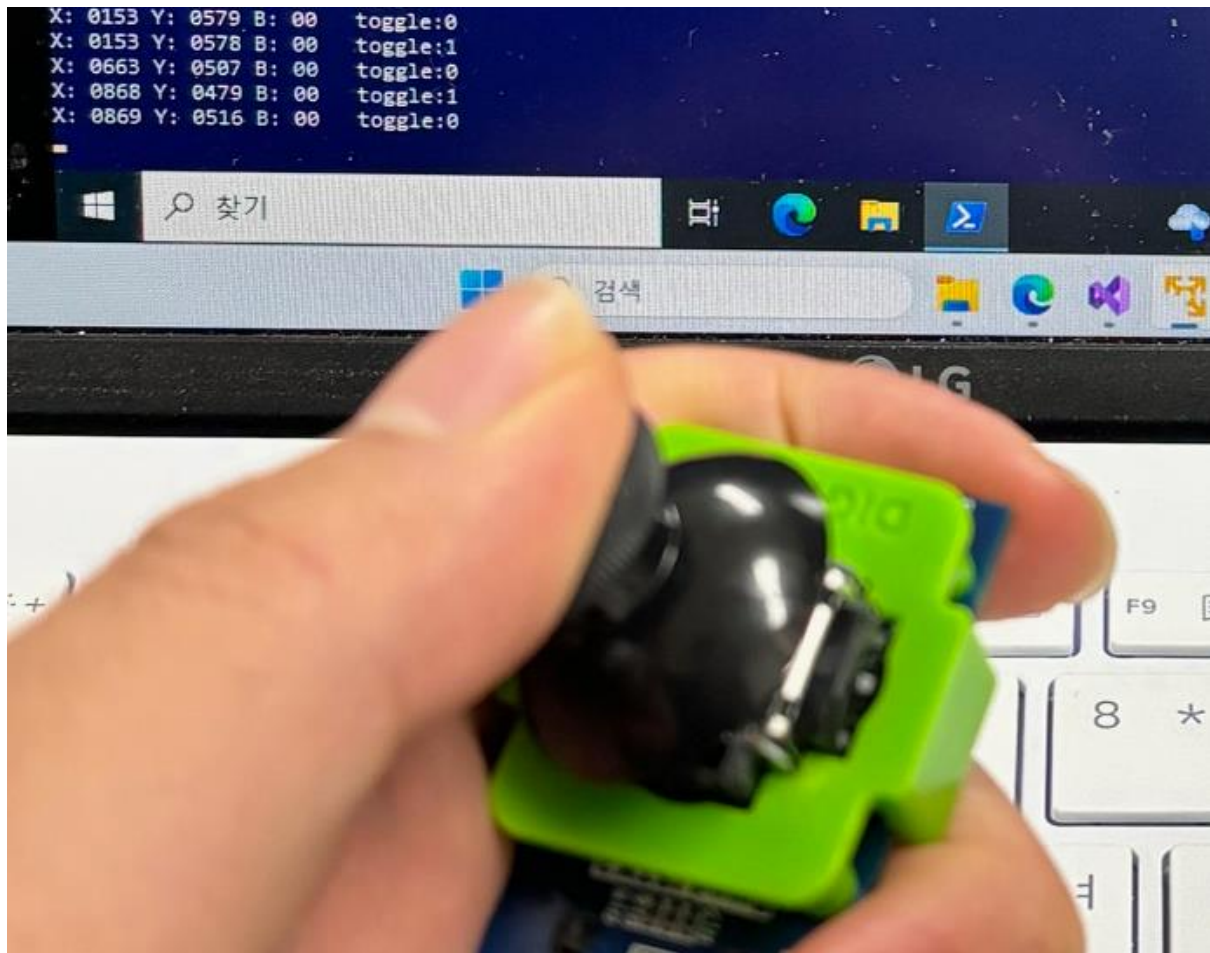
위 코드는 나노 보드를 반납한 이후라서 나노 보드에 ssh 로 접속해서 찍은 캡처본이 없다.

문제는 Joystick 으로부터 읽은 data 가 비트가 반전되어서 들어온다는 점이다. 이를 해결하는 방법은 이후에 문제 해결에서 서술한다. 다음은 나노 보드에서 위의 코드를 실행하고 ZYNQ 보드에서 앞서 서술한 C 코드를 FPGA 를 굽고 실행하면 다음과 같은 결과를 볼 수 있다.

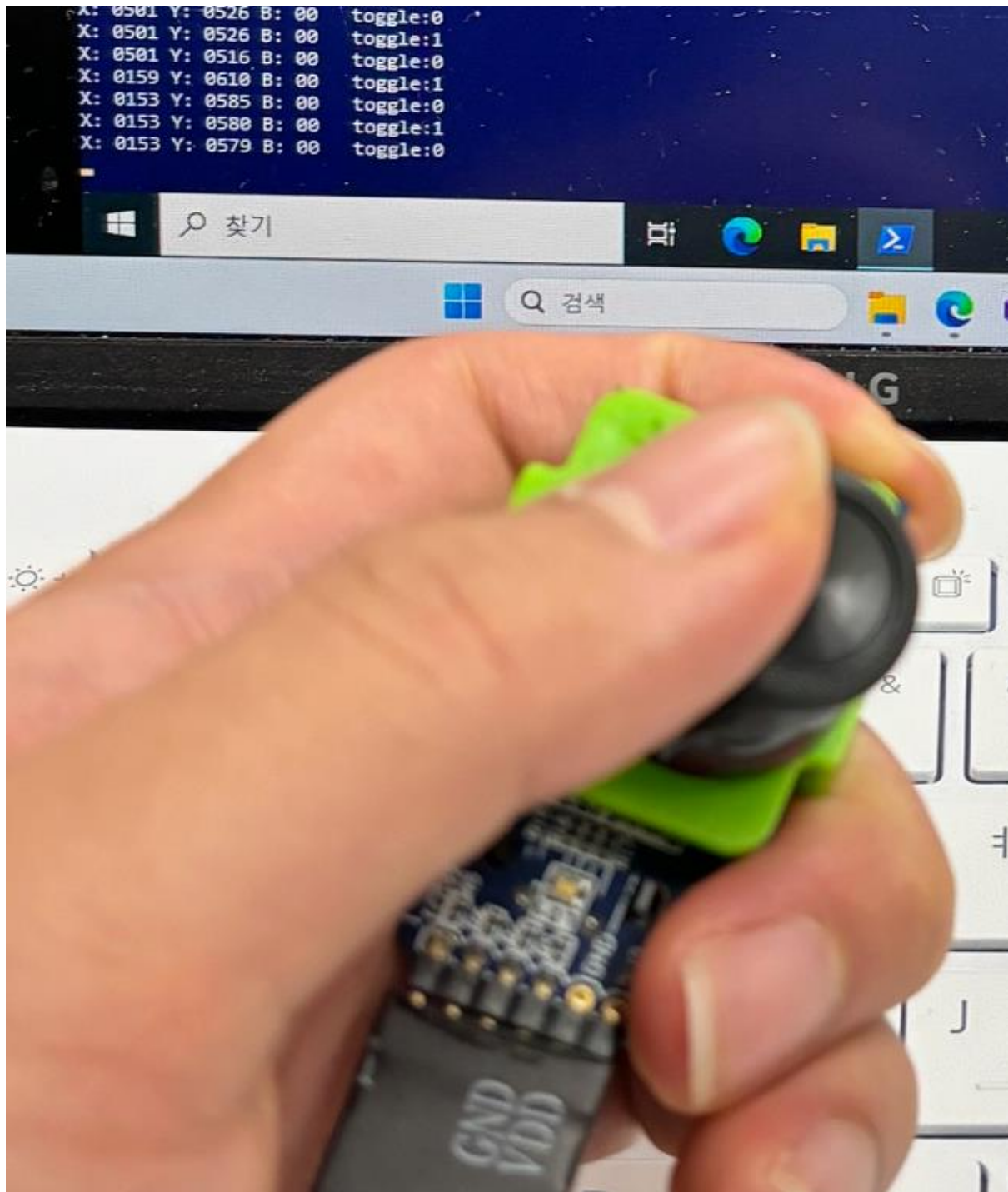
다음은 위 코드를 실행하기 위해 zync 보드와 조이스틱과 나노보드를 연결한 모습이다.



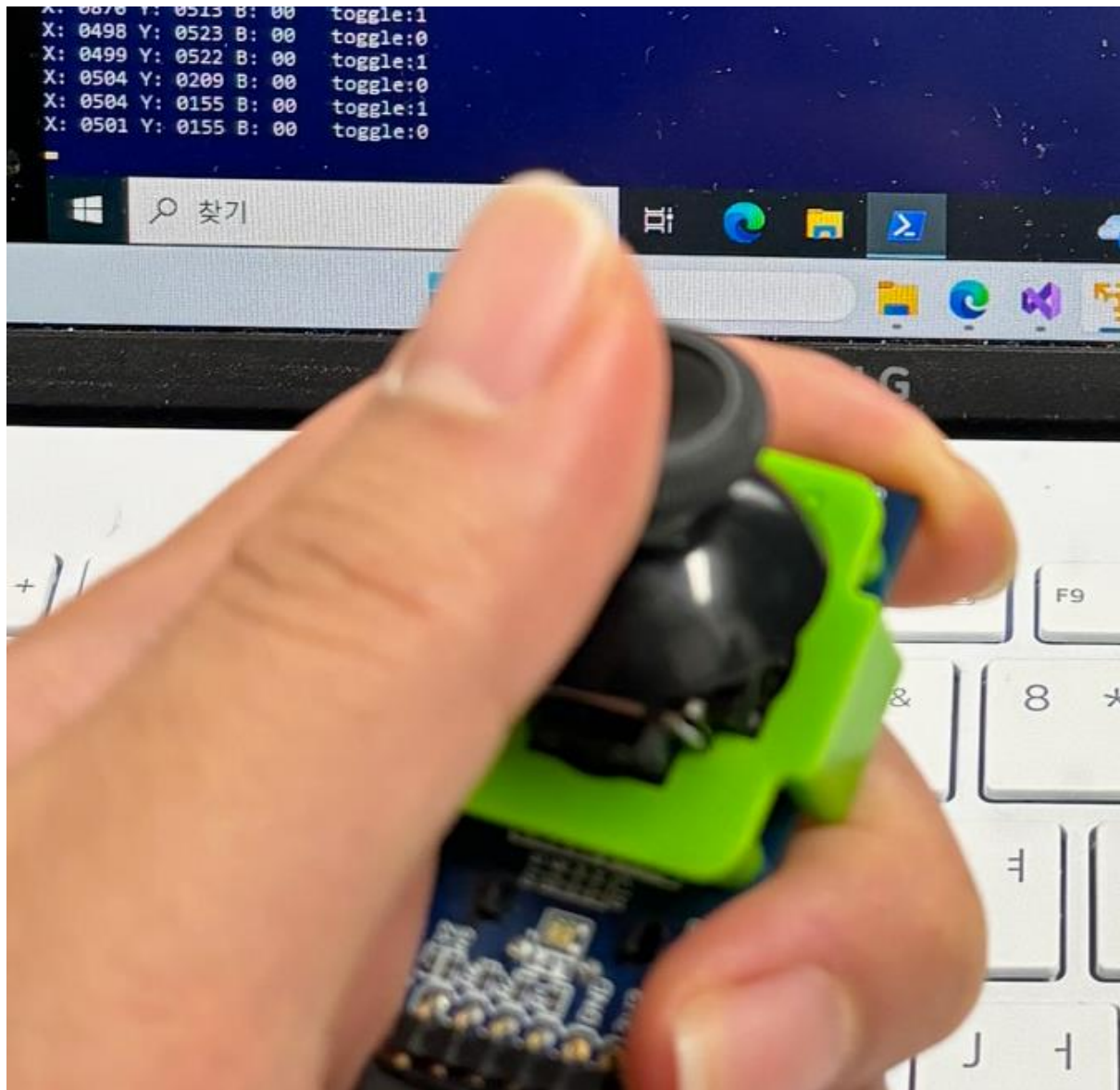
다음은 조이스틱과 nano board 간의 연결을 확인하는 모습이다.



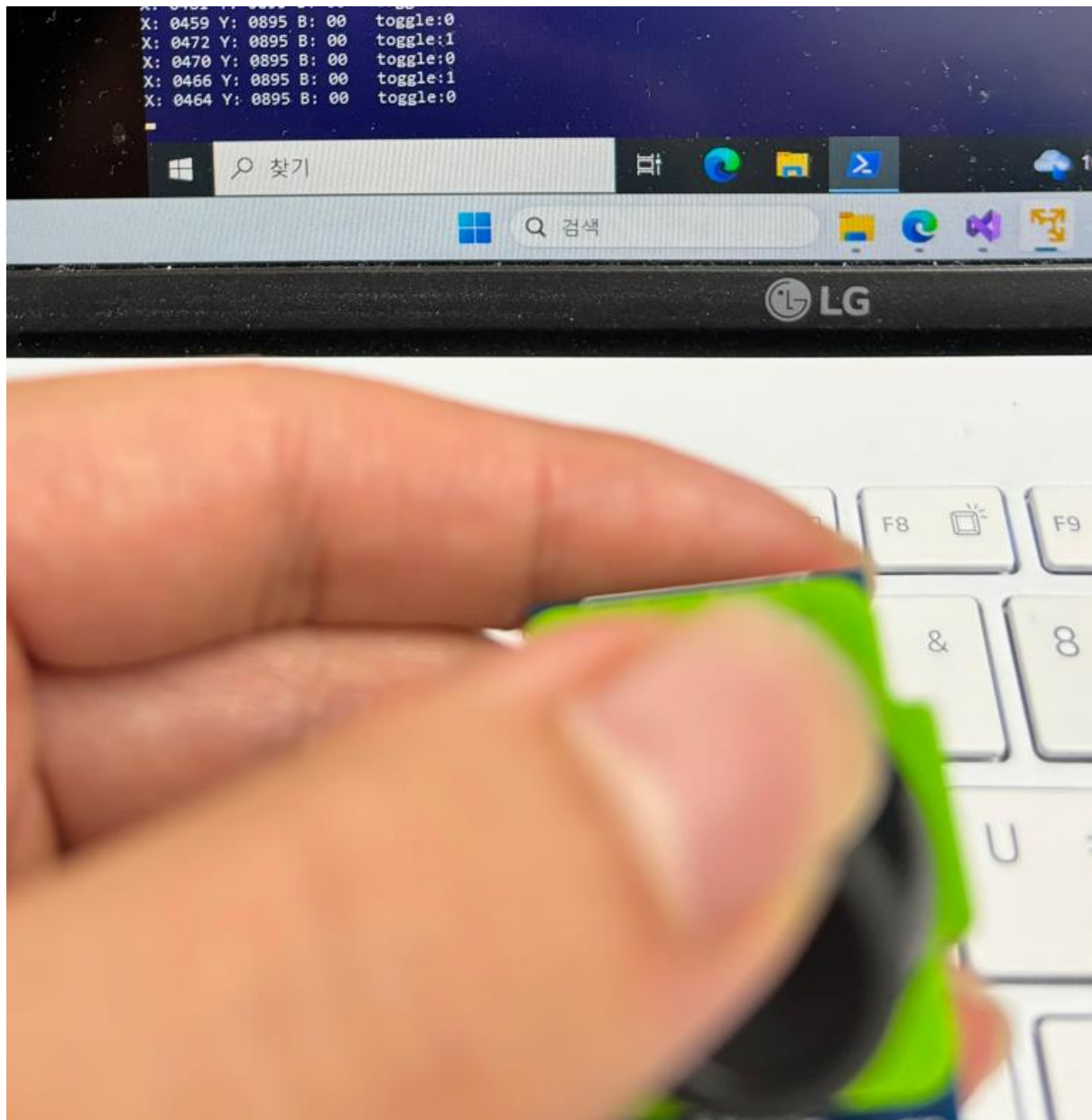
조이스틱을 왼쪽으로 뺏기니 X 값이 커져서 869 가 나왔다.



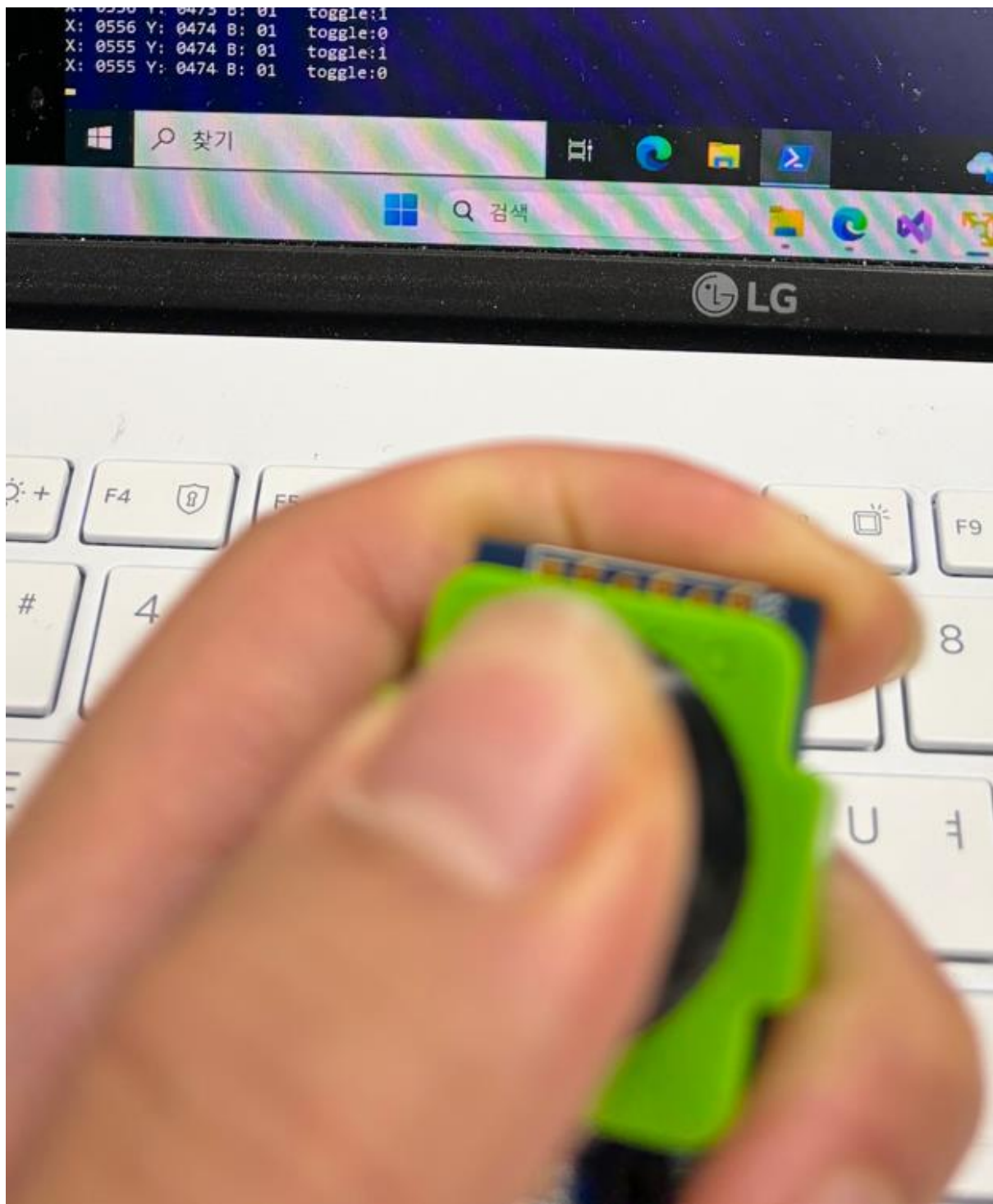
조이스틱을 오른쪽으로 밀면 X 값이 작아져서 153 이 나온다.



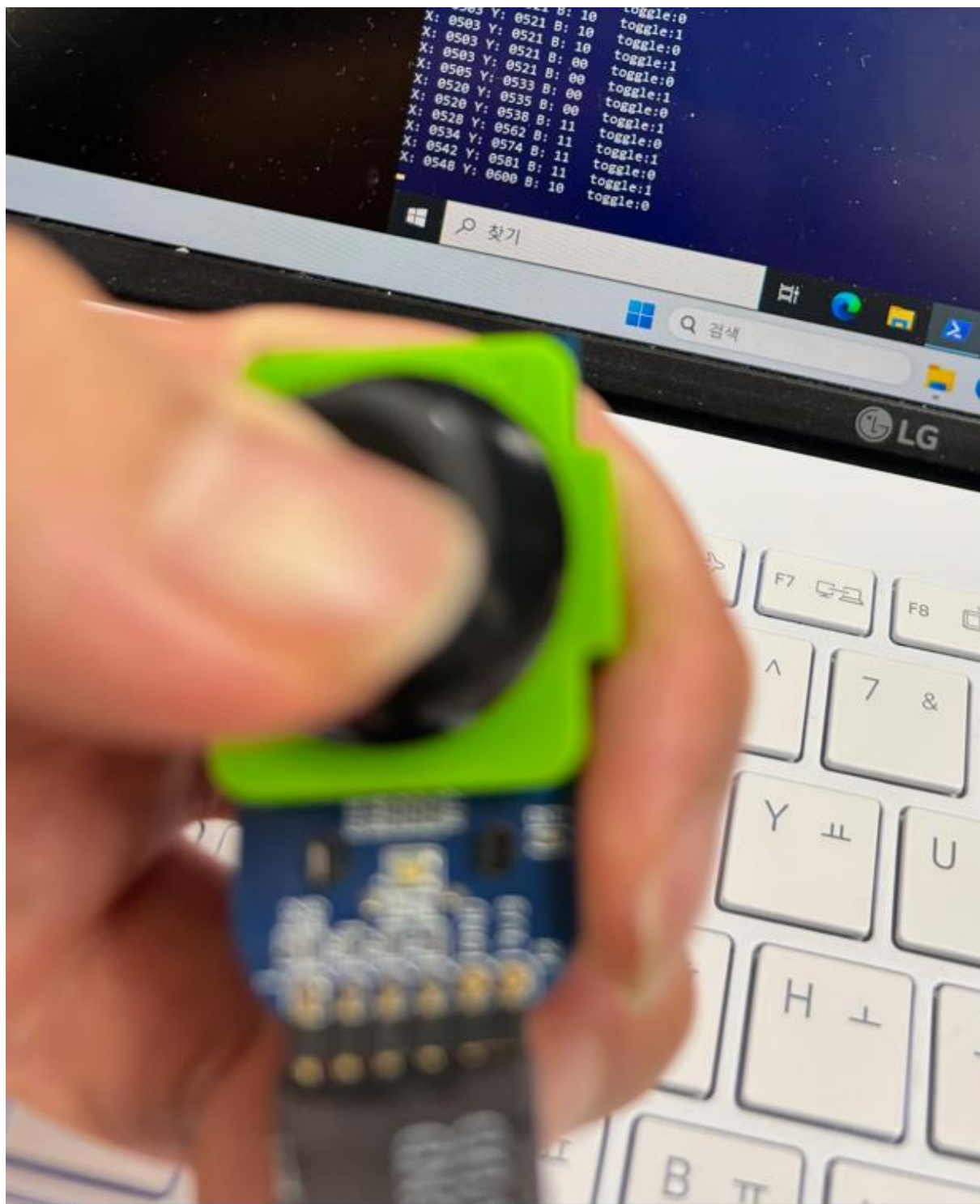
조이스틱을 위로 밀면 Y 값이 작아져서 155 가 나왔다.



조이스틱을 아래로 땡기면 Y 값이 커져서 895 가 나왔다.

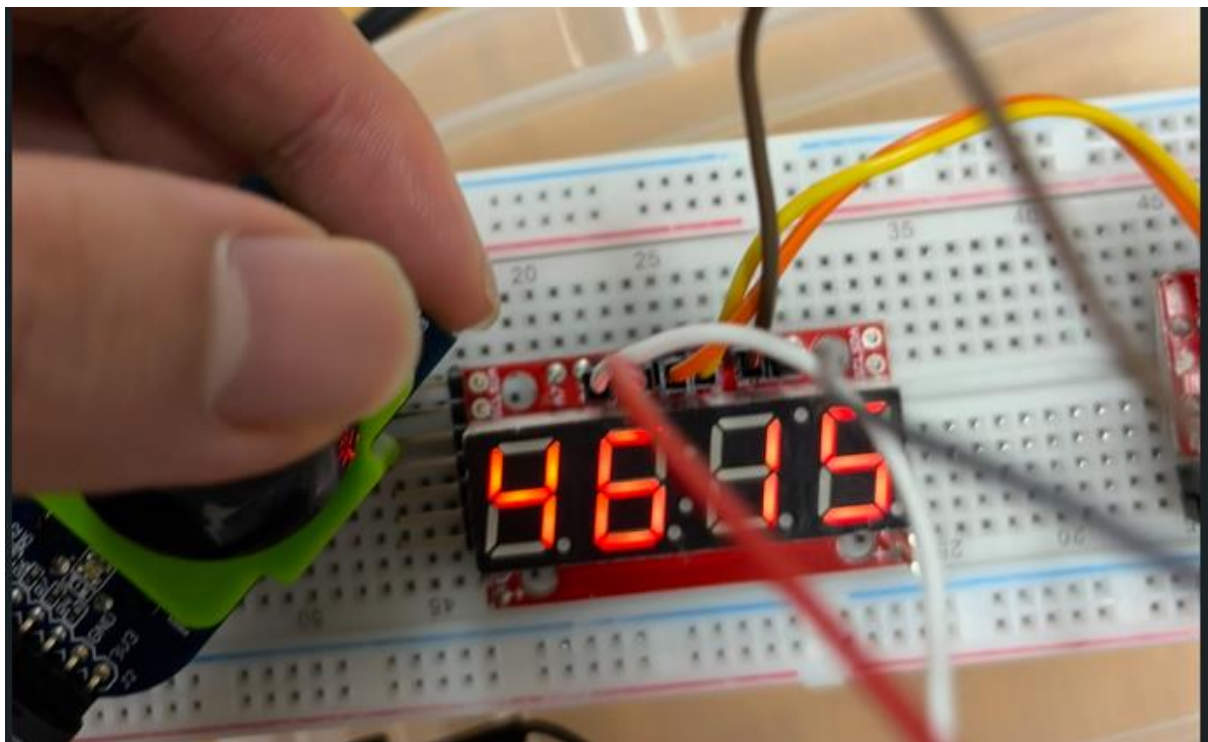


조이스틱 앞쪽 버튼을 누르니 B 값이 01로 바뀌었다.

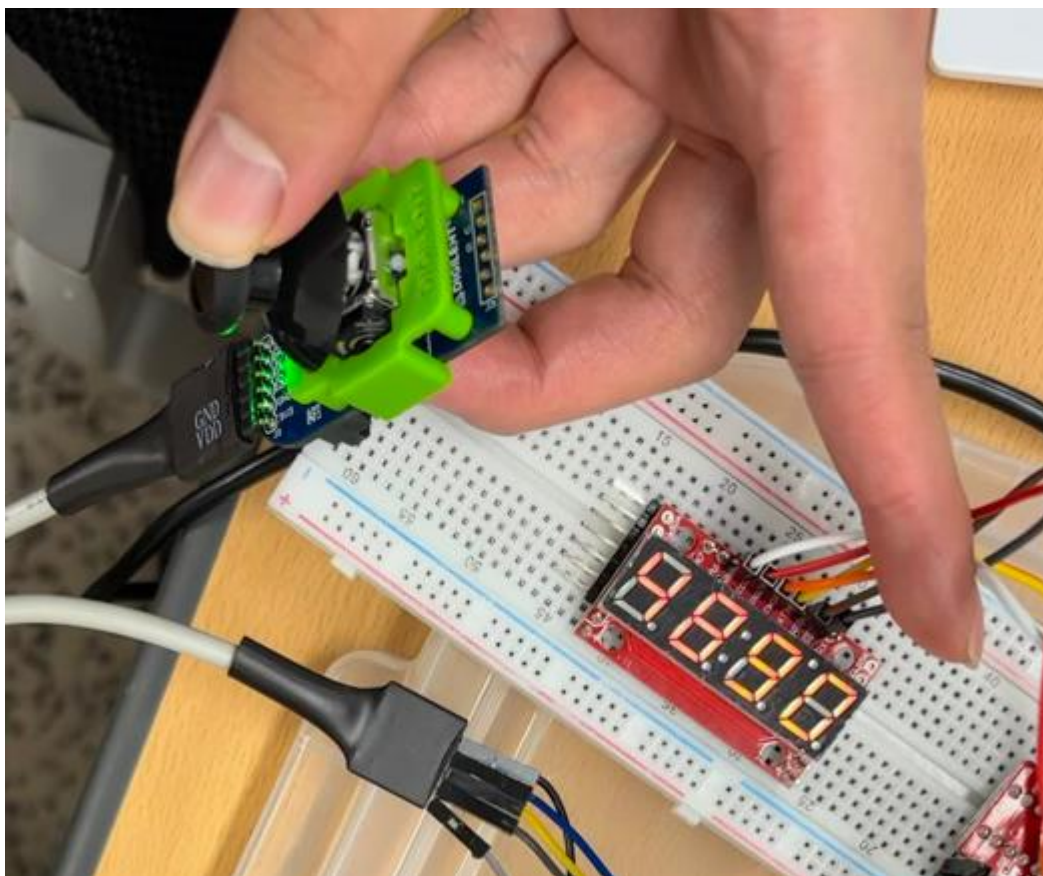


조이스틱 뒤쪽 버튼을 누르니 B 값이 10 이 나왔다. 그리고 그 윗줄 로그와 같이 양 버튼을 둘 다 누르면 11 도 뜬다.

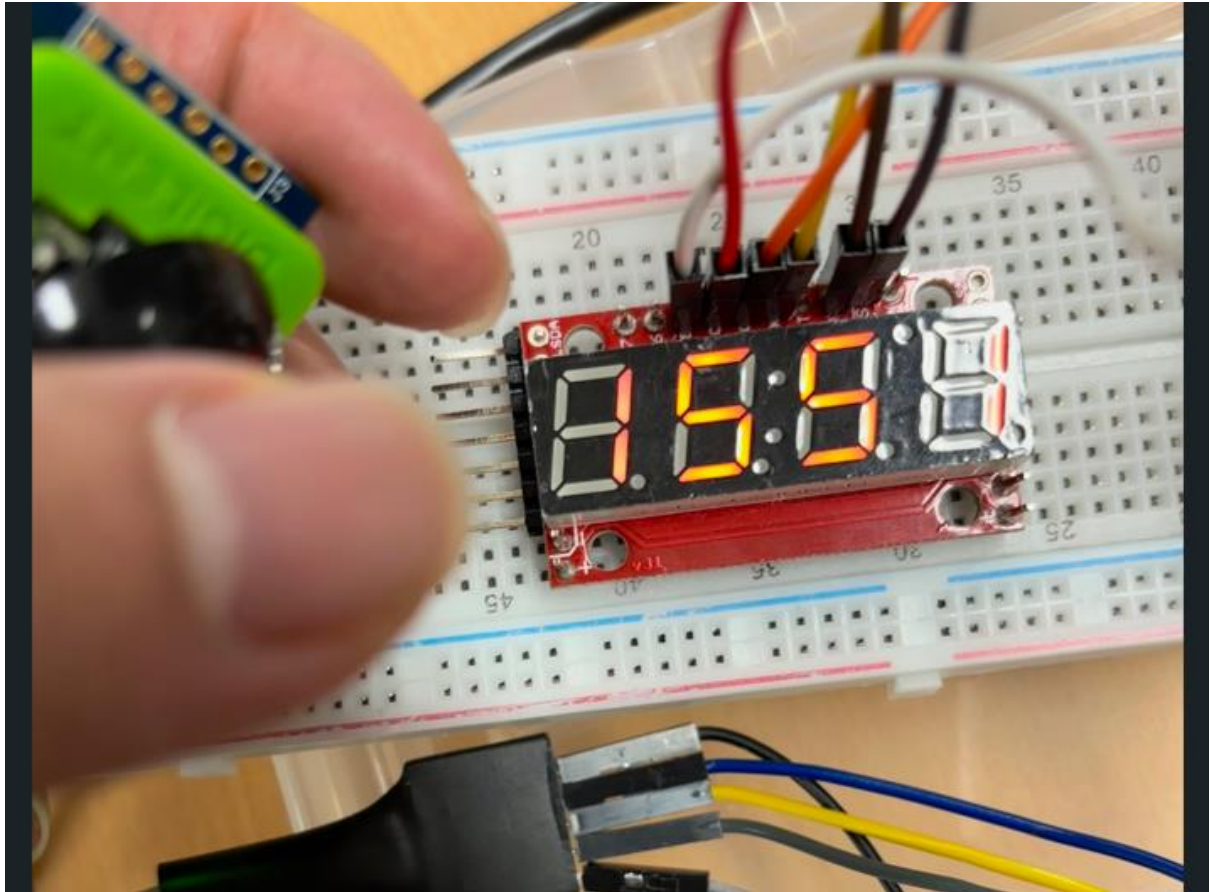
다음으로는 SPI 연결된 7-segment 에 값이 잘 나오는지 확인한다. 7-segment 에는 X 좌표 2 자리가 왼쪽에, Y 좌표 2 자리가 오른쪽에 나와서 총 4 자리 10 진수로 값이 표현된다.



조이스틱을 위로 올렸더니 4615 로 x 값 약 460, y 값 약 150 으로 값이 나옴을 확인할 수 있다.



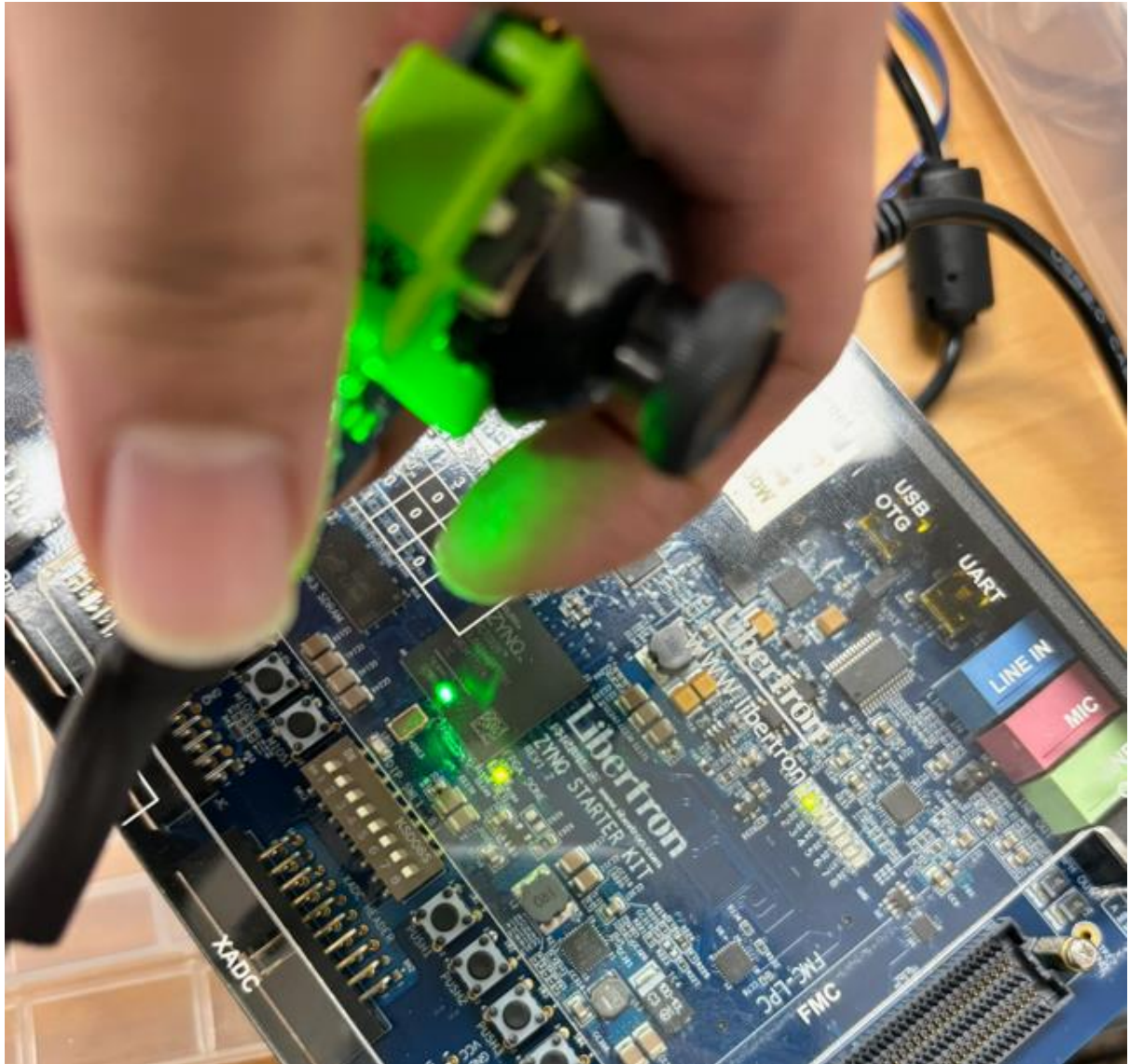
조이스틱을 아래로 내리면 4890 이 떈서 x 값 약 480, y 값 약 900 이 뜸을 알 수 있다.



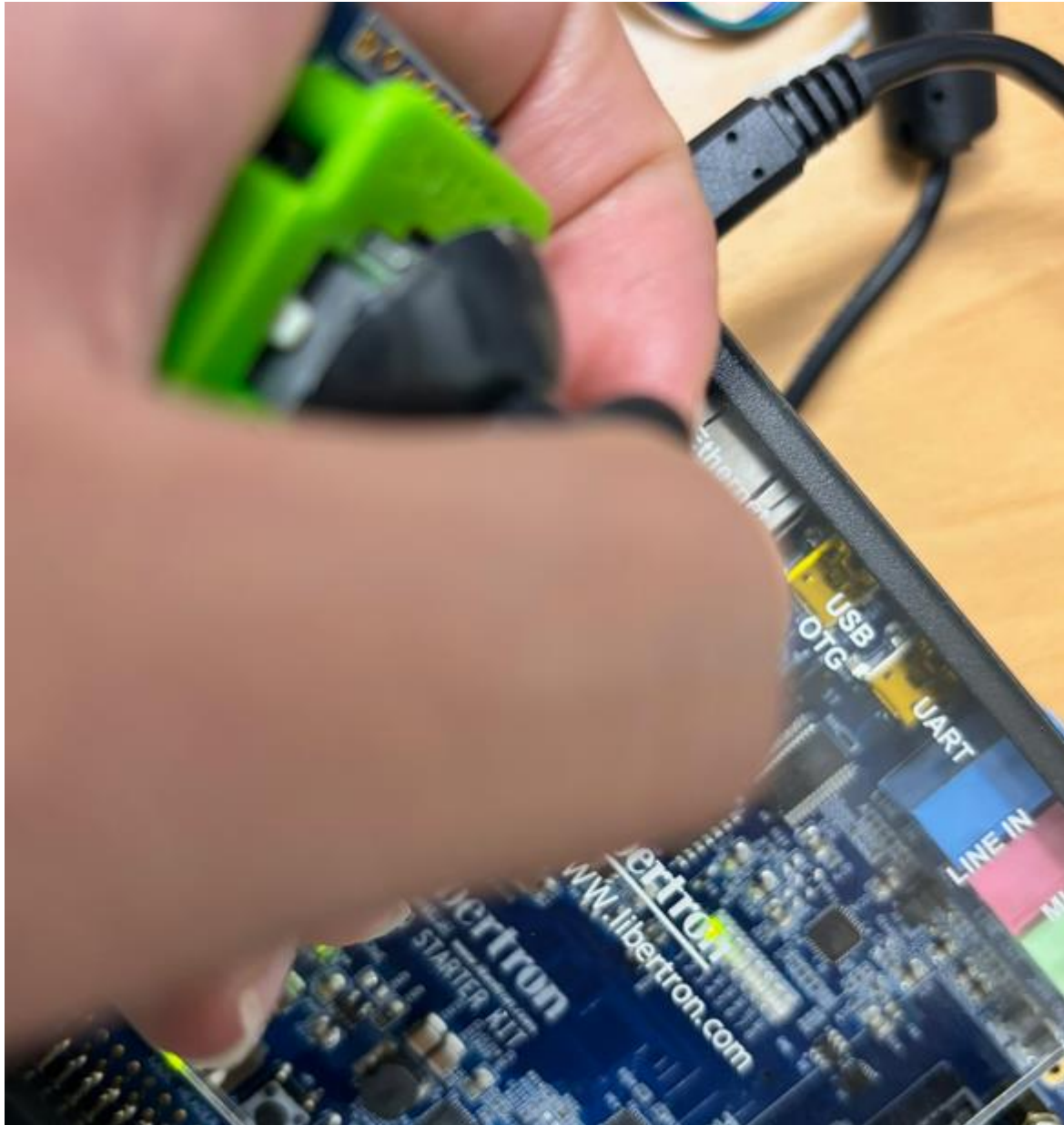
조이스틱을 오른쪽으로 밀면 1551, x 값 약 150, Y 값 약 510 이 뜸을 알 수 있다.

조이스틱을 왼쪽으로 미는 사진을 찍지 않았음을 보고서를 작성하면서 깨달았다 아마 9051 과 같은 값이 뜰 것이다.

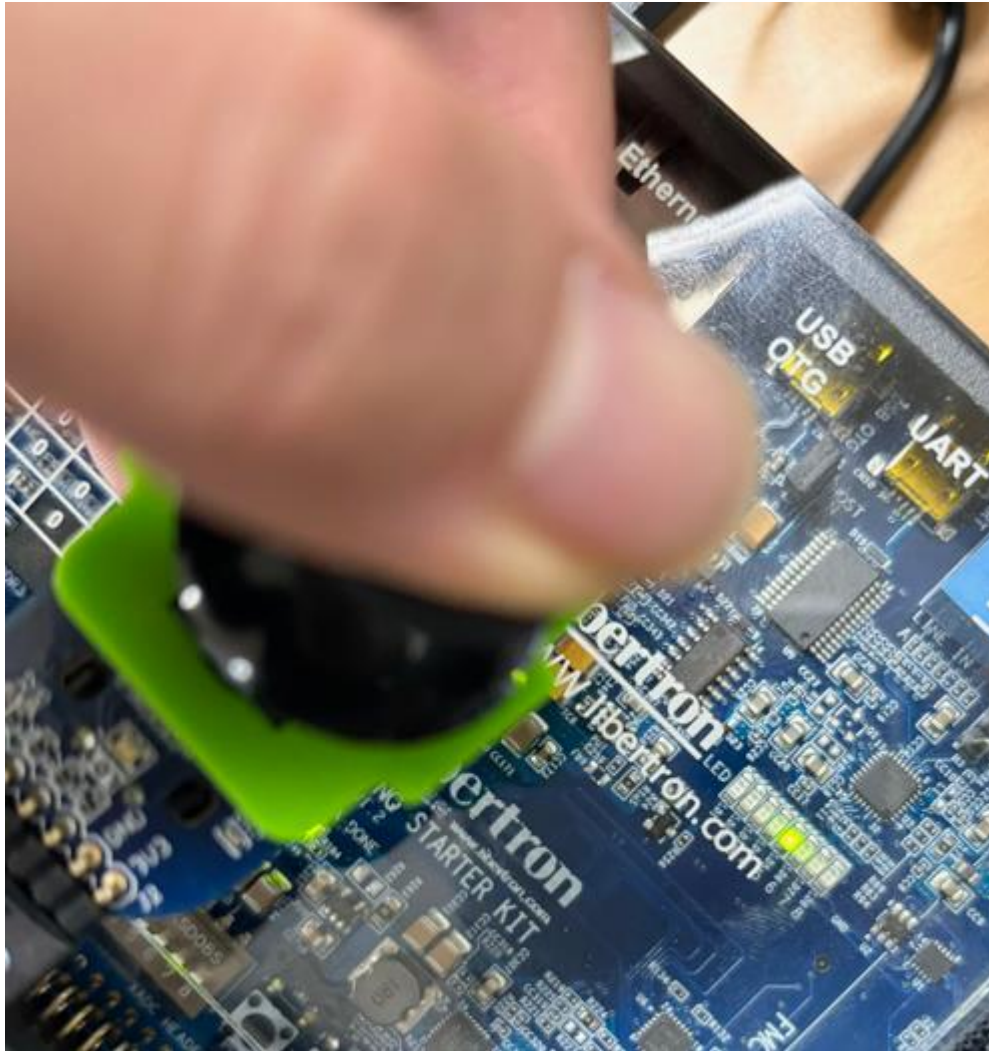
다음으로는 조이스틱과 ZYNQ 보드상의 LED 가 뜨는지를 확인한다.



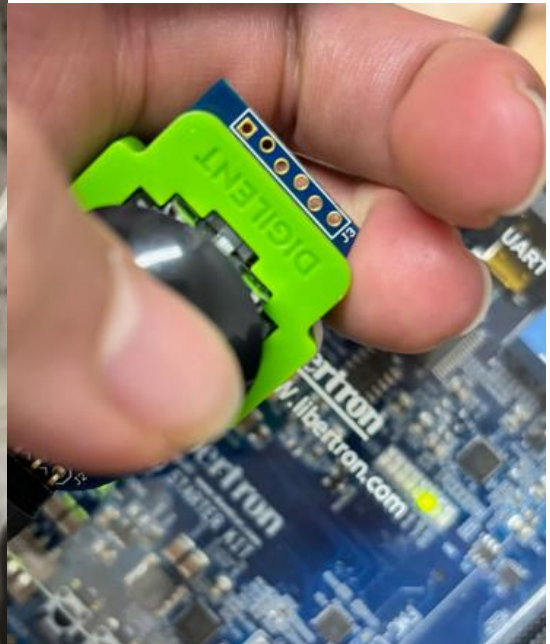
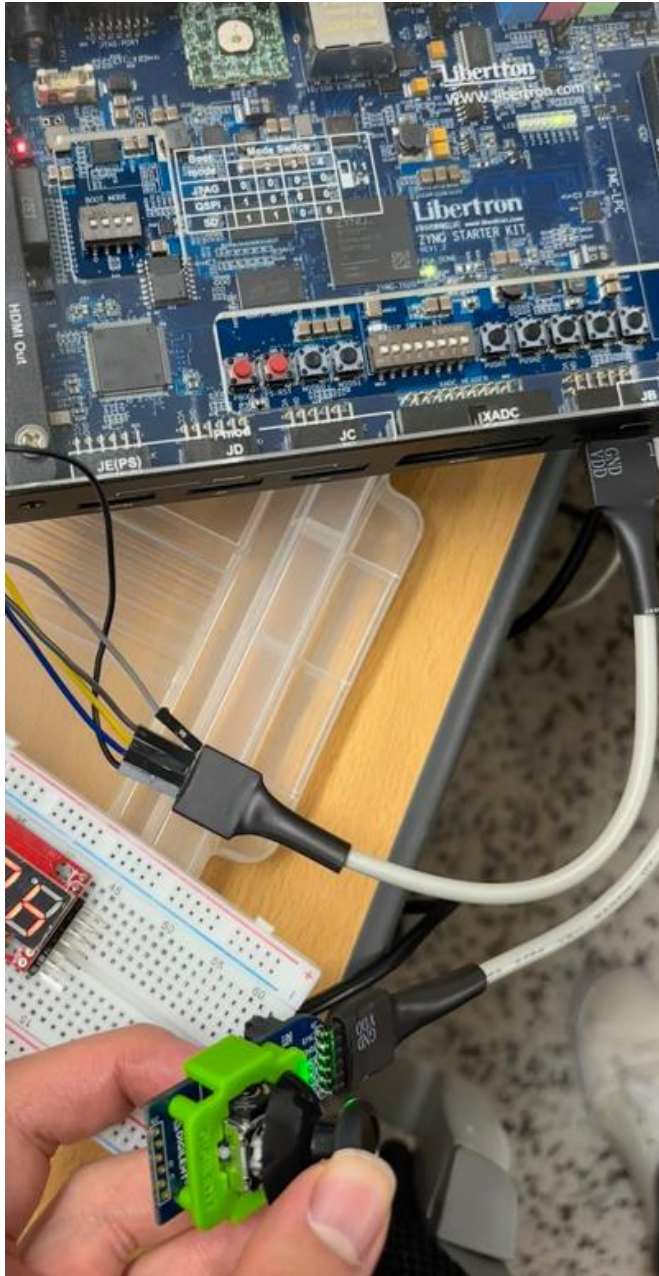
후면 버튼을 눌러서 2 번 LED 를 점등할 수 있다.



전면 버튼을 눌러서 1 번 LED 를 점등시킬 수 있다.



조이스틱을 위로 밀면 5 번 LED 가 점등한다.



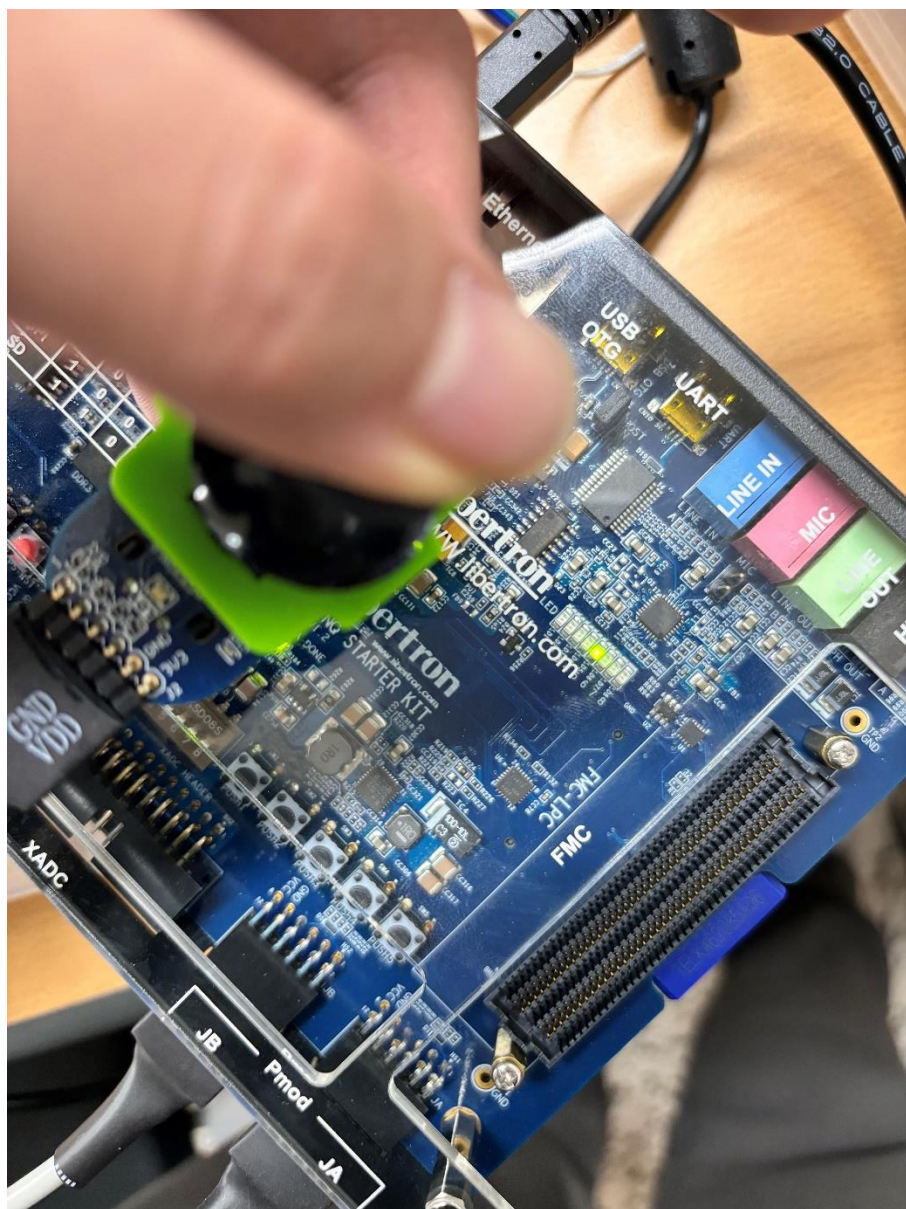
화질이 조금 좋지 못하지만 조이스틱을 아래로 내리면 6 번 LED 를 점등할 수 있다.

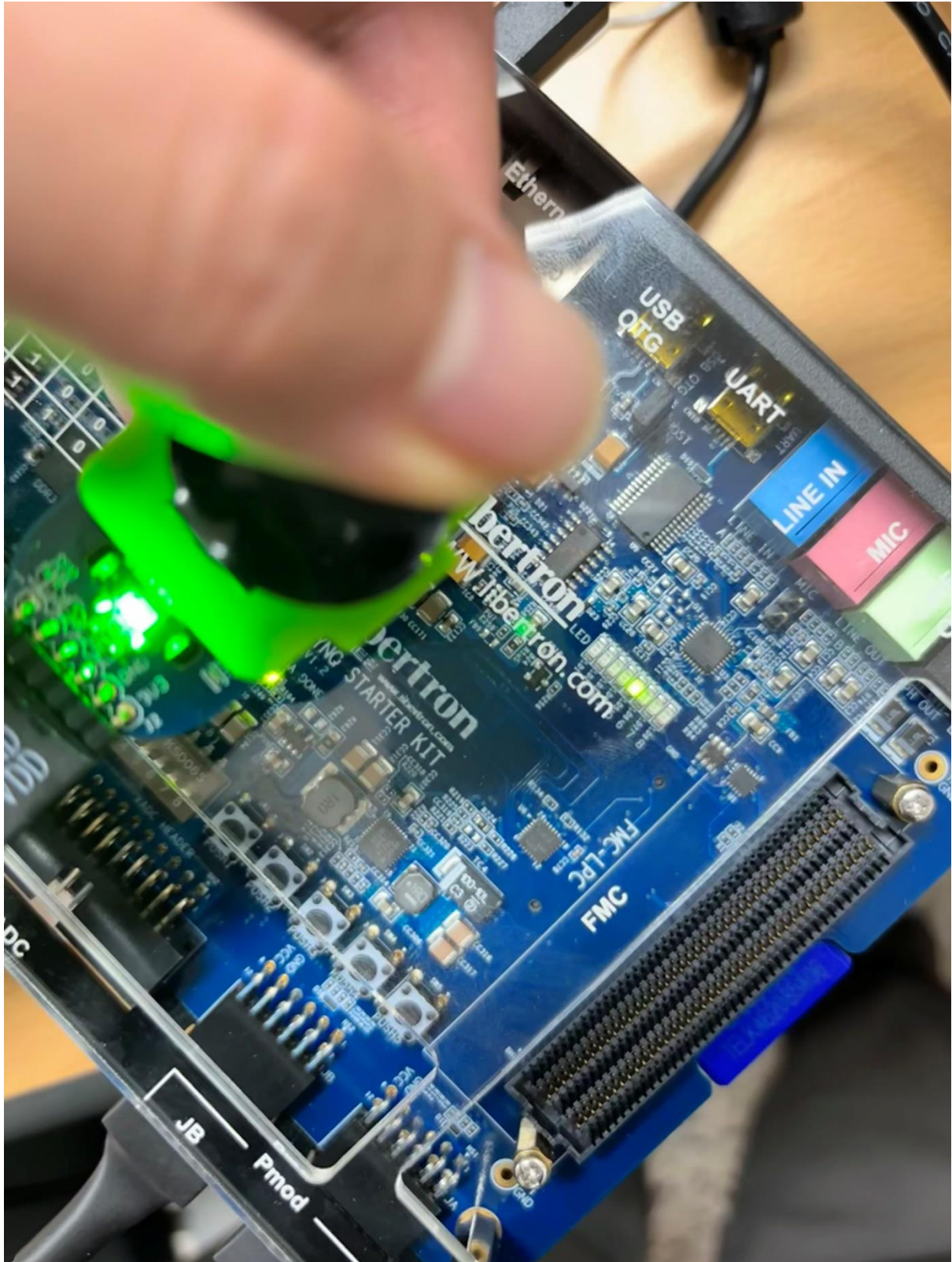


조이스틱을 우측으로 밀어서 7 번 LED 를 점등할 수 있다.

공교롭게도 또 좌측으로 미는 사진을 찍지 못했다. 좌측으로 밀면 8 번 LED 를 점등할 수 있다.

마지막으로 조이스틱 LED 점멸 여부를 확인한다.





같은 위치에서 LED 가 1 초 주기로 깜빡거린다.

이번 과제를 하면서 기본적으로 주어진 문제사항은 SPI 모듈 설계에 실수가 있어서 비트가 반전되어서 data 가 들어온다는 점이다. 이를 해결하기 위해 커널 SPI 드라이버 코드를 수정하였다.

커널 모드에서 유저 모드로 데이터를 전송해주는 함수인 `copy_to_user` 함수를 찾고 다음 캡처의 사진이 spi 버스로 읽은 데이터를 유저 모드로 보내주는 함수인 것 같아 수정하였다.

```

/* Read-only message with current device setup */
static ssize_t
spidev_read(struct file* filp, char __user* buf, size_t count, loff_t* f_pos)
{
    struct spidev_data* spidev;
    ssize_t status = 0;

    /* chipselect only toggles at start or end of operation */
    if (count > bufsiz)
        return -EMSGSIZE;

    spidev = filp->private_data;

    mutex_lock(&spidev->buf_lock);
    status = spidev_sync_read(spidev, count);
    if (status > 0) {
        unsigned long missing;
        int i;

        for (i = 0; i < status; i++)
            spidev->rx_buffer[i] = ~(spidev->rx_buffer[i]);

        missing = copy_to_user(buf, spidev->rx_buffer, status);
        if (missing == status)
            status = -EFAULT;
        else
            status = status - missing;
    }
    mutex_unlock(&spidev->buf_lock);

    return status;
}

```

선택된 부분이 추가로 넣어준 코드이다. Rx_buffer 배열을 status 개의 원소를 참조해서 비트 반전시켰다. 그리고 이 수정된 코드가 들어있는 모듈을 다시 컴파일해서 운영체제 모듈로 넣어주면 비트 반전을 드라이버 레벨에서 바로잡을 수 있다.

다른 문제는 분명 SPI 버스로 7-segment 에 data 를 쓰는데, ZYNQ 보드도 자꾸 이 정보를 읽어와서 쓰는 문제이다. NANO 보드에서 나가는 CE 핀을 ZYNQ 보드를 고르는 핀과 7-segment 를 고르는 핀을 각각 빼 가면서 테스트해 보았는데, 7-segment 는 CE 가 없으면 당연히 데이터를 통신하지 않아서 표시되는 값을 갱신하지 않는다. 즉, CE 에 영향을 받는다. 그리고 ZYNQ 도 CE 를 빼면 마찬가지로 데이터를 통신하지 않아서 NANO terminal 상에서 데이터를 읽지 않고 있음을 확인할 수 있고, 조이스틱으로부터 데이터를 읽어오지 않기 때문에 쓰지도 않는다. 물론 모든 값이 0,0,0,0 혹은 255,255,255,255 로 들어오므로 ZYNQ 보드 LED 를 제어하는 신호와 조이스틱 LED 를 제어하는 신호는 분명 나갈 텐데 이 역시도 통신이 되지 않기 때문에 LED 가 점등되지 않는다. 결론은 둘 다 연결했을 때, ZYNQ 보드가 7-segment 에 보내는 신호도 CE 와 상관 없이 자기가 받는다고 생각하고

읽어오는 것으로 추측할 수 있다. 특이한 점은, 7-segment 는 ZYNQ 보드로 가는 데이터를 참조하지 않는 것처럼 보이는데, ZYNQ 보드의 LED 를 보면 원래 제어해야 하는 불빛이 아주 약하게 잠깐 들어오고 7-segment 의 3 번째로 와야 하는 값(Y 좌표의 10 의 자릿수 값)의 bit 가 들어와 있는 것을 확인할 수가 있다. 또한 조이스틱의 LED 도 조이스틱을 움직여야만 깜빡이는 걸로 ZYNQ 보드가 CE 와 관련 없이 데이터를 자기 자신으로 주는 데이터로 인식하는 것으로 추측할수 있었다.

원인은 넷 중 하나로 생각할 수 있다. 나노 보드 유저 모드에 있는 파이썬 코드가 이상하거나, 나노 보드 SPI 커널 코드가 이상하거나, 보드에 올라간 C 코드가 이상하거나, FPGA 를 굽는 모듈이 이상하거나. 우선 코드에는 이상이 없다. 오히려 spidev 모듈이 하이 레벨로 구현해 줘서 함수만 잘 쓰면 버스 연결 쪽으로는 고려할 것이 없다. 또한 SPI 버스로 ZYNQ 로 데이터를 보내는 동안 중간에 또 7-segment 로 데이터를 보내서 간섭이 생기나 싶어 ZYNQ 로 데이터를 쓰고 0.25 초 기다리고 7-segment 로 데이터를 쓰고 0.25 초 기다리고를 반복시켜도 작동은 똑같았다. 결론적으로 SPI 관련 명령어는 이상이 없다고 볼 수 있다. ZYNQ 보드에서 작동하는 C 코드는 ZYNQ 보드의 작동이 워낙 단순하므로 실질적으로 무한반복문 안에 들어가는 C 언어 문장은 5 줄이다. 5 줄을 천천히 확인해보도 이상은 없다. 운영체제 모듈도 건드린 것이라고는 copy_to_user 를 수행하기 전 유저 메모리로 복사해 줄 커널 변수 값을 비트 반전한 것밖에 없다. 그리고 수정한 커널이 이상이었다면 원본 커널 모듈로는 정상이어야 하는데 원본 커널 모듈을 설치하고 파이썬 코드를 작동해도 똑같은 이상이 있었다. 결론적으로 가장 의심이 되는 부분은 HDL 로 작성된 SPI SLAVE 모듈이다.

이 문제를 해결하기 위해 쓰지 않는 ZYNQ 보드로 가는 변수의 [31:16]bit 값을 0 이 아니라 FFFF 로 보내는 것이었다. 이렇게 해서 ZYNQ 보드 C 코드 상에서 FFFF 로 시작하지 않으면 해당 값을 무시하게 설계를 하였다. 결국 CE 와 관계없이 데이터를 받아오는 ZYNQ 보드 내 모듈을 해결하기 위해 software 로 해결한 셈이다. 참고로 7-segment 로 보내는 값은 각 byte 별로 0~10 사이이기 때문에 한 바이트에 FF 가 절대 올 수가 없다. 따라서 FFFF 는 오로지 ZYNQ 보드가 본인으로 보내는 정보를 인식하게 하기 위한 더미데이터이다.

실습을 학교에서 끝나치고 집에 오면서 다르게 해결하는 방법도 떠올렸다. 7-segment 는 자기에게 보내는 데이터를 정확하게 아니까 time.sleep 을 하기 전에 먼저 7-segment 에 표시 데이터를 보내고 ZYNQ 보드 제어 데이터를 다음에 보내면 위 문단과 같은 작업 없이도 정상 작동할 것 같다고 추측한다. 하지만 이미 모든 장비를 반납하고 보고서를 작성하고 있기에 그럴 것 같다고 추측만 해 본다.

2. 분석 및 토의

이번 과제에서는 조이스틱 신호를 받아서 ZYNQ 보드를 통해 Nano 보드로 넘겨서 계산을 완료 한 후 다시 ZYNQ 보드를 통해 조이스틱과 ZYNQ 보드 내부에서 LED 를 제어하고 Nano 보드에서 직접 SPI 통신으로 7-segment 에 데이터를 보내는 실습을 하였다. 이 과정에서 조이스틱 데이터가 비트

반전되어서 나오는 문제, ZYNQ 보드에 올라간 SPI 모듈이 CE 신호와 관련이 없이 모든 데이터를 받아들이는 문제가 있었다. 비트 반전 문제는 커널 레벨에서 디바이스 드라이버를 수정해서 해결하였고, ZYNQ 보드 내 모듈 오작동 문제는 ZYNQ 보드 행동 변경 및 ZYNQ 보드에 보내는 데이터 형식을 변경하여 해결하였다.

이번주 과제에서는 3 가지의 질문이 있다. 하드웨어에 버그가 났을 경우 어떻게 고칠 수 있는가에 대한 질문이다.

우선 하드웨어에 버그가 있음을 확인했을 때, 어플리케이션 코드를 변경하는 경우는 어떤 경우인가에 대한 질문에 답변은 “급할 때”이다. 혹은 하드웨어 버그가 작동에는 이상이 없고 출력신호에만 이상이 있을 때 가볍게 유저 어플리케이션 레벨에서 소프트웨어 코드 수정을 통해 해결할 수 있을 것이다. 이 방법은 해당 디바이스를 작동시키는 전용 프로그램이 따로 있을 경우에 쓰기 유용할 것이다. 하지만 디바이스를 사용하기 위해 따로 프로그램을 까는 것을 선호하지 않는 유저에게는 해당 하드웨어 자체를 쓰기 힘든 경우도 생길 수 있을 것이다.

다음으로는 커널 코드를 수정하는 경우이다. 위와 같이 해당 디바이스를 사용하기 위해 따로 프로그램을 깔지 않고도 디바이스를 정상 작동시키기 위해 커널 드라이버를 수정하는 방법도 있다. 이 방법을 사용하면 해당 디바이스를 다양한 응용프로그램이 다른 정상적인 디바이스를 다루듯이 편하게 작동시킬 수 있을 것이다. 문제는 커널을 수정해야 하기 때문에 사용자가 드라이버를 깔아야 한다는 점이다. 그래도 교수님 설명에 따르면 마이크로 소프트에 일단 드라이버를 보내면 윈도우에서 필요한 드라이버는 알아서 찾아서 깔아 준다고 해서 유저 입장에서는 따로 응용프로그램을 까는 것보다 훨씬 편하게 디바이스를 사용할 수 있을 것이다.

마지막으로는 펌웨어를 수정하는 경우이다. 이 경우는 디바이스 내부 프로세서에서 자체적으로 하드웨어 버그를 잡아주거나 하드웨어 버그가 너무 심해서 특정 기능 자체를 정지시키는 방법으로도 해결할 수 있을 것이다. 다만 특정 기능을 정지시킬 정도면, 그 제품을 생산한 회사의 손해가 막심할 것이다. 대신에 유저 입장에서는 일단 드라이버나 응용프로그램이 따로 필요하지 않음으로 가장 자연스럽게 디바이스를 사용할 수 있어서 좋은 점이 있을 것으로 생각한다.

감사합니다!

3. References

이성원/마이크로아키텍처 강의자료/광운대학교/2023