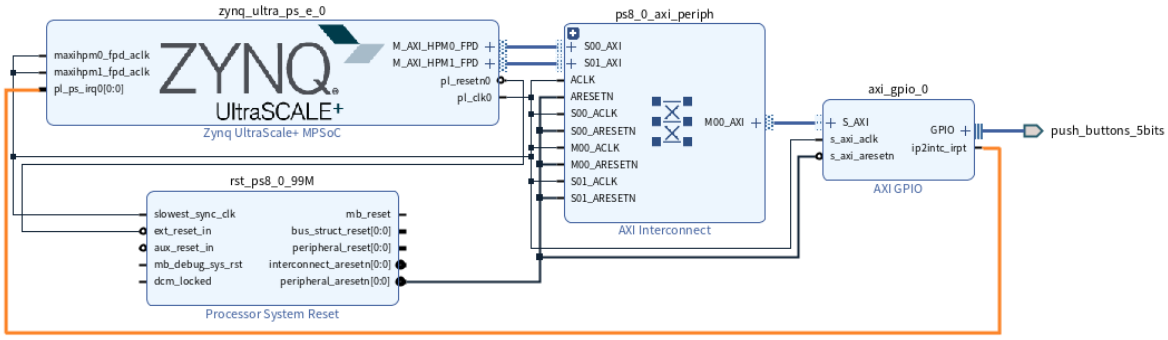
# 5주차 : Interrupt vs. Polling

## 1. AXI GPIO Interrupt

이전까지는 memory address를 mapping하여 device driver를 만드는 것을 완료하였고, 이제는 어떠한 두 객체사이의 통신을 통해 보드를 제어해보고자 한다. OS의 도움을 받아 인터럽트를 사용하면 목표를 달성 할 수 있다. 그러기 위해서는 Vivado에서 interrupt를 받기 위해 기존 IP에서 일련의 설정을 추가하여야 한다.




기존 AXI GPIO IP에서 상세 설정에서 Enable Interrupt를 체크하고, 새로 생긴 port를 zynq_ultra_ps_e_0의 pl_ps_irq에 연결해준다. 만약 PS IP에 Interrupt를 받는 port가 없다면 상세 설정에서 PS-PL configuration에서 interrupt를 받는 것으로 하고, 이때 받는 interrupt는 PL to PS가 되어야 한다. 위와 같은 과정을 거쳤다면 Vivado에서 이전에 하던 것과 동일하게 bitstream을 만들어 export해주고, Petalinux를 빌드해주면 된다.

Petalinux를 빌드할 때 추가사항으로, PL configuration 사항들이 device-tree에 들어가야 하는데, 안들어가는 경우가 발생한다. 이전에도 AXI GPIO가 FPGA에 포함되어 있을 때, Petalinux를 부팅하면 부팅 로그에 XGpio가 어떤 주소에 mapping되었는지에 대한 로그가 나와야 하는데, 이 로그가 나오질 않았다. 세부 사항은 후술하고, petalinux-config –get-hw-description 시점에 DTG(Device Tree Generating) 메뉴에서 Remove PL in Device-tree를 체크 해제 해주어야 부팅 로그에 원하는 로그가 나온다. 해당 로그는 아래 그림과 같다.

```
[    3.805267] of-fpga-region fpga-full: FPGA Region probed
[    3.811202] XGpio: gpio@a0000000: registered, base is 507
```

이렇게 나오면 Petalinux Device-tree에 올바르게 등록되어 있는 것을 확인 할 수 있다. 이제 간단한 모듈을 만들어서, kernel에서 Interrupt과정이 이루어지는지 확인해보도록 한다.

## 2. AXI GPIO Interrupt – Module code

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");
static unsigned int gpio_button = 507;
//static unsigned int gpio_led = 499;
static unsigned int irqNum;
static bool ledValue = 0;
static irq_handler_t irq_handler(unsigned int irq, void *dev_id, struct pt_regs
*regs){
    printk(KERN_ALERT "Button pressed\n");
    return (irq_handler_t) IRQ_HANDLED;
}
static int button_init(void){
    int result;
    gpio_request(gpio_button,"sysfs");
    gpio_direction_input(gpio_button);
    gpio_export(gpio_button,false);
    printk(KERN_INFO"GPIO_TEST : The button state is currently
%d\n",gpio_get_value(gpio_button));
    irqNum = gpio_to_irq(gpio_button);
    printk(KERN_INFO"GPIO_TEST : The button is mapped to IRQ %d\n",irqNum);
    result =
request_irq(irqNum,(irq_handler_t)irq_handler,IRQF_TRIGGER_RISING,"switch",NULL
);
    if(result < 0){
        printk(KERN_INFO"GPIO_TEST : Failed to request IRQ %d\n", result);
        return result;
    }
    return 0;
}
static void button_exit(void){
    free_irq(irqNum,NULL);
    gpio_unexport(gpio_button);
    gpio_free(gpio_button);
}
module_init(button_init);
module_exit(button_exit);
```

위 코드는 보드의 push button을 누르면 Interrupt가 발생되는 것을 받아 handler로 넘기는 과정을 등록한 코드이다. 먼저 버튼이 몇번 gpio인지 알아내야 한다. 알아내는 방법은 위의 부팅 로그를 보면 base가 몇번인지 알 수 있다. 다음은 document를 보면 알 수 있지만, 이번에는 devmem을 통해 알아냈다. Devmem 0xA0000000 w를 치면 현재 저 주소에 있는 값을 알 수 있는데, 버튼을 누르고 있으면 값이 들어와 있을테니 이러한 방식으로 알아냈다. 그 결과, 가운데 버튼이 첫번째 비트, 좌 버튼이 두번째, 하 버튼이 세번째, 우 버튼이 네번째, 상 버튼이 다섯번째 비트에 해당되었다. 따라서, 첫번째 비트인 가운데 버튼이 507이므로 좌 버튼은 508번일 것이다. 이번에는 한개의 버튼만 필요하므로 507만

코드에서 사용하였다. 또한, gpio API 사용이 필요해졌다. Interrupt를 사용하기 전에는 주소만 가지고 있으면 됐고, 이 주소는 Vivado에서 확인이 가능하였기 때문에 크게 gpio API를 사용하지 않아도 되었다. 이번에는 interrupt를 interrupt table에 등록하는데 필요한 interrupt number가 필요하였고, 이를 임의로 지정할 수 없었기 때문에 미리 내장되어 있는 번호를 가져오는 것이 중요했다. 실제로, Vivado에서 vitis로 넘어가면, xparameters.h에 interrupt number를 121이라고 주지만, 이 번호로 등록을 하려하면 interrupt가 서로 matching되지 않아 핸들러가 등록이 되지 않았다.

```
gpio_direction_input(gpio_button);
```

이 함수는 gpio 의 direction 을 지정해준다. Gpio_direction_input 이므로 button 은 모두 input 으로 설정된다.

```
gpio_export(gpio_button,false);
```

이 함수는 /sys/class/gpio에서 export 파일 역할을 하는 듯 하다. 해당 디렉토리에서 echo <#gpiochip number> > export를 입력하면 gpio#에 해당하는 interface를 조작할 수 있다. 그 과정을 해주는 함수이다. 두번째 인자가 false인 것은 true라면 export하는 과정에서 direction이 바뀔 수 있다고 한다.

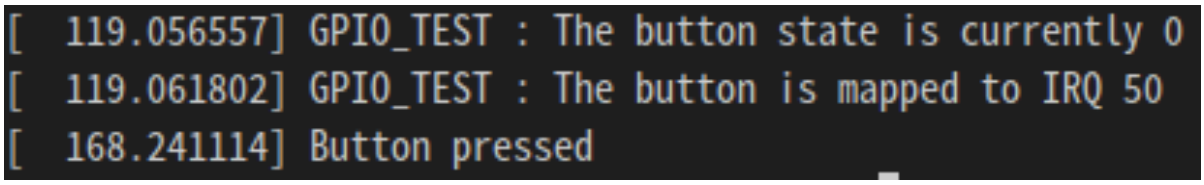
```
irqNum = gpio_to_irq(gpio_button);
```

이 함수는 request_irq의 첫번째 인자에 해당하는 인터럽트 주소를 가져 올 수 있는 함수이다. Gpio_button은 button에 해당하는 gpio number이고 이 parameter를 넣으면 해당하는 인터럽트 주소를 얻어올 수 있다. 이를 request_irq에 넣어 Interrupt를 Kernel에 등록시키는 것이다.

```
request_irq(irqNum,(irq_handler_t)irq_handler,IRQF_TRIGGER_RISING,"switch",NULL
);
```

이 함수는 interrupt를 Kernel에 등록하는 함수이다. 첫번째 인자는 interrupt 주소이고, 두번째 인자는 해당 interrupt를 받으면 어떠한 handler로 통제할지를 알려주는 것이다. IRQF_TRIGGER_RISING은 해당 신호가 Posedge일 때 인터럽트가 발생하는 것이다. 따라서, 버튼이 눌릴 때 인터럽트는 발생한다. 다시 나올 때는 발생하지 않는다. “switch”는 /proc/interrupts에 등록되는 이름으로 쓰인다.

이 module을 Insmod로 등록하고, 버튼을 누르면 다음과 같은 kernel message와 몇 번의 interrupt가 발생했는지를 /proc/interrupts를 통해 알 수 있다.



```
           CPU0       CPU1       CPU2       CPU3
  3:       4572       5452       4221       4023     GICv2  30 Level     arch_timer
  6:          0          0          0          0     GICv2  67 Level     zynqmp_ipi
 12:          0          0          0          0     GICv2 155 Level     axi-pmon, axi-pmon
 13:          0          0          0          0     GICv2 156 Level     zynqmp-dma
 14:          0          0          0          0     GICv2 157 Level     zynqmp-dma
 15:          0          0          0          0     GICv2 158 Level     zynqmp-dma
 16:          0          0          0          0     GICv2 159 Level     zynqmp-dma
 17:          0          0          0          0     GICv2 160 Level     zynqmp-dma
 18:          0          0          0          0     GICv2 161 Level     zynqmp-dma
 19:          0          0          0          0     GICv2 162 Level     zynqmp-dma
 20:          0          0          0          0     GICv2 163 Level     zynqmp-dma
 21:          0          0          0          0     GICv2 164 Level     Mali_GP_MMU, Mali_GP, Mali_PP0_MMU, Mali_PP0, Mali_PP1_MMU, Mali_PP1
 22:          0          0          0          0     GICv2 109 Level     zynqmp-dma
 23:          0          0          0          0     GICv2 110 Level     zynqmp-dma
 24:          0          0          0          0     GICv2 111 Level     zynqmp-dma
 25:          0          0          0          0     GICv2 112 Level     zynqmp-dma
 26:          0          0          0          0     GICv2 113 Level     zynqmp-dma
 27:          0          0          0          0     GICv2 114 Level     zynqmp-dma
 28:          0          0          0          0     GICv2 115 Level     zynqmp-dma
 29:          0          0          0          0     GICv2 116 Level     zynqmp-dma
 31:       1367          0          0          0     GICv2  95 Level     eth0, eth0
 33:        121          0          0          0     GICv2  49 Level     cdns-i2c
 34:        156          0          0          0     GICv2  50 Level     cdns-i2c
 35:          0          0          0          0     GICv2  42 Level     ff960000.memory-controller
 36:          0          0          0          0     GICv2  57 Level     axi-pmon, axi-pmon
 37:         43          0          0          0     GICv2  47 Level     ff0f0000.spi
 38:          0          0          0          0     GICv2  58 Level     ffa60000.rtc
 39:          0          0          0          0     GICv2  59 Level     ffa60000.rtc
 40:          0          0          0          0     GICv2 165 Level     ahci-ceva[fd0c0000.ahci]
 41:       5507          0          0          0     GICv2  81 Level     mmc0
 42:         73          0          0          0     GICv2  53 Level     xuartps
 45:          0          0          0          0     GICv2  84 Edge      ff150000.watchdog
 46:          0          0          0          0     GICv2  88 Level     ams-irq
 47:       6690          0          0          0     GICv2 154 Level     fd4c0000.dma
 48:     494816          0          0          0     GICv2 151 Level     fd4a0000.zynqmp-display
 55:          0          0          0          0     GICv2  97 Level     xhci-hcd:usb1
IPI0:      2882       4149       4997       6976       Rescheduling interrupts
IPI1:       250        357        821        515       Function call interrupts
IPI2:         0          0          0          0       CPU stop interrupts
IPI3:         0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:         0          0          0          0       Timer broadcast interrupts
IPI5:         0          0          0          0       IRQ work interrupts
IPI6:         0          0          0          0       CPU wake-up interrupts
Err:          0
```

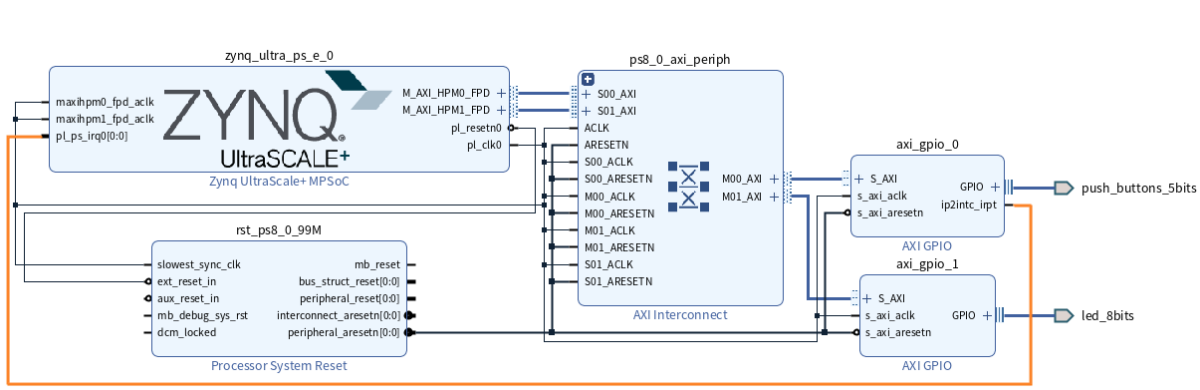
<GPIO에 대한 interrupt가 없는 상황 (Before)>

```
 47:      23830          0          0          0     GICv2 154 Level     fd4c0000.dma
 48:    2208610          0          0          0     GICv2 151 Level     fd4a0000.zynqmp-display
 50:          1          0          0          0     xgpio   0 Edge      button_isr
 55:          0          0          0          0     GICv2  97 Level     xhci-hcd:usb1
```

<버튼을 한번 눌렀을 때 들어온 /proc/interrupts>

# 3. Interrupt Time

Interrupt와 Polling은 학부 수업 시간에도 주로 비교되는 대상들이다. 보통 수업에서는 Interrupt는 Context-switch의 비용이 비싸 자주 일어나지 않는 event에 적합한 처리 방식이고, polling은 기본적으로 busy-wait으로 일정 주기로 값을 감시하고 있으면서 값이 조건에 맞는 값이 들어온다면 필요한 일을 처리하는 것이기 때문에 자주 일어나는 event에 적합한 처리 방법으로 설명되어지고 있다. 두 방식을 언제 어떻게 쓰느냐는 명확하게 주어지지는 않고, 주어진 상황에 optimal한 방법을 취하는 방식으로 알려져 있다. 이번 파트에서는 interrupt를 활용하여 LED의 전원을 조작하는 것과, polling 방식으로 계속 값을 read하여 LED의 전원을 조작하는 것을 통해 대략적인 시간을 비교해보고자 한다.
먼저 필요한 FPGA는 다음과 같다.




이 회로는 위의 회로에서 AXI GPIO를 LED용으로 하나 더 추가한 것과 같다. Interrupt와 Polling을 비교하기 위해 하나는 interrupt를 enable하여 PS와 연결시켜주었고, 한 회로는 interrupt를 꺼두었다. 두 FPGA에 대응되는 Petalinux를 각각 build하였으며, 하나의 SD카드를 사용하였다.

A. Interrupt

Interrupt는 kernel 단계에서 등록하고 LED의 전원을 조작하는 것은 interrupt handler에서 처리하였다. 아래는 해당 코드이다.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");
static unsigned int gpio_button = 507;
static unsigned int gpio_led = 499;
static unsigned int irqNum;
static bool ledValue = 0;
static irq_handler_t irq_handler(unsigned int irq, void *dev_id, struct pt_regs
*regs){
    printk(KERN_ALERT "Button pressed\n");
    ledValue = !ledValue;
    gpio_set_value(gpio_led,ledValue);
    return (irq_handler_t) IRQ_HANDLED;
```

```
}
static int button_init(void){
    int result;
    gpio_request(gpio_led,"sysfs");
    gpio_direction_output(gpio_led,ledValue);
    gpio_export(gpio_led,false);
    gpio_request(gpio_button,"sysfs");
    gpio_direction_input(gpio_button);
    gpio_export(gpio_button,false);
    printk(KERN_INFO"GPIO_TEST : The button state is currently
%d\n",gpio_get_value(gpio_button));
    irqNum = gpio_to_irq(gpio_button);
    printk(KERN_INFO"GPIO_TEST : The button is mapped to IRQ %d\n",irqNum);
    result =
request_irq(irqNum,(irq_handler_t)irq_handler,IRQF_TRIGGER_RISING,"switch",NULL
);
    if(result < 0){
        printk(KERN_INFO"GPIO_TEST : Failed to request IRQ %d\n", result);
        return result;
    }
    return 0;
}
static void button_exit(void){
    free_irq(irqNum,NULL);
    gpio_unexport(gpio_button);
    gpio_free(gpio_button);
    gpio_unexport(gpio_led);
    gpio_free(gpio_led);
}
module_init(button_init);
module_exit(button_exit);
```

Interrupt 처리 방식과 동일하게 gpio API를 사용하여 LED를 활성화시키고, irq_handler에서 현재의 LED값을 반전시키게 구현하였다. 즉, 버튼을 한번 누를 때마다 LED 값이 꺼지거나 켜지거나로 작동한다.

Interrupt 처리 시간 계산 식은 다음과 같이 정의하였다.

$$T_{interrupt} = T_{CT(user \rightarrow kernel)} + T_{Handler} + T_{CT(kernel \rightarrow user)}$$

```
# tracer: function_graph
# button is isq_handler
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0) # 2721.692 us |  button [inter2]();
 0) # 2695.069 us |  button [inter2]();
 0) # 2695.209 us |  button [inter2]();
 0) # 2694.730 us |  button [inter2]();
 0) # 2700.830 us |  button [inter2]();
 0) # 2694.679 us |  button [inter2]();
 0) # 2700.150 us |  button [inter2]();
 0) # 2697.579 us |  button [inter2]();
 0) # 2700.010 us |  button [inter2]();
 0) # 2694.520 us |  button [inter2]();
```

$$T_{Handler-avg} = 2699.4468us$$

```
|      /* sched_switch: prev_comm=kpktgend_3 prev_pid=311 prev_prio=120
prev_state=S ==> next_comm=swapper/3 next_pid=0 next_prio=120 */
 3)               |      fpsimd_thread_switch() {
 3)   0.870 us    |        __get_cpu_fpsimd_context();
 3)   0.840 us    |        fpsimd_save();
 3)   0.840 us    |        __put_cpu_fpsimd_context();
 3)   6.150 us    |      }
 3)   0.861 us    |      hw_breakpoint_thread_switch();
 3)   0.830 us    |      uao_thread_switch();
 ------------------------------------------
 3)  kpktgen-311   =>    <idle>-0
 ------------------------------------------

 3)   1.450 us    |      finish_task_switch();

 ------------------1st
```

$$T_{Context-Switch-avg} = 12.0601us$$

$$\boldsymbol{T_{interrupt} = 2 * (12.0601) + 2699.4468 = 2723.567\ us}$$

측정하는 과정 중 어려움이 있어 가정을 몇가지 세우게 되었다.

1) ftrace 의 sched_switch 과정을 context-switch 로 가정한다.

2) User -> Kernel , Kernel -> User 의 Context-switch 를 동일하다 가정한다.

그 결과 Interrupt 처리 과정 시간은 2723.567 us 라는 결과가 나오게 되었다.

Polling 또한 진행해보고자 하였다. 결과는 내지 않았고 진행 과정 중 후술 할 판단에 의해 구현만 하고 경과 시간 결과를 내진 않았다. 다음은 Polling 관련 User app 코드와 device driver 코드이다.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/unistd.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <linux/delay.h>
#include <linux/io.h>
#include <linux/device.h>
#define DEVICENAME "poll"
#define MINOR_BASE 0
MODULE_LICENSE("GPL");
static dev_t my_dev; // major # && minor #
static struct class *my_class;
static struct cdev my_cdev;
static unsigned int gpio_button = 507;
static unsigned int gpio_led = 499;
static unsigned int irqNum;
static bool ledValue = 0;
static bool buttonstatus;
/*static irq_handler_t button(unsigned int irq, void *dev_id, struct pt_regs
*regs){
```

```
    printk(KERN_ALERT "Button pressed\n");
    ledValue = !ledValue;
    gpio_set_value(gpio_led,ledValue);
    return (irq_handler_t) IRQ_HANDLED;
}*/
static int button_open(struct inode *inode, struct file *file){
    printk(KERN_INFO "Driver opened\n");
    return 0;
}
static int button_release(struct inode *inode, struct file *file){
    printk(KERN_INFO "Driver closed\n");
    return 0;
}
static ssize_t button_read(struct file* file, char* buf, size_t count, loff_t*
offset){
    buttonstatus = gpio_get_value(gpio_button);
    copy_to_user(buf,&buttonstatus,sizeof(buttonstatus));
    return 0;
}
static ssize_t button_write(struct file* file, const char* buf, size_t count,
loff_t* offset){
    ledValue = !ledValue;
    gpio_set_value(gpio_led,ledValue);
    return 0;
}
static struct file_operations led_driver_fops = {
    .owner = THIS_MODULE,
    .open = button_open,
    .write = button_write,
    .read = button_read,
    .release = button_release,
};
static int button_init(void){
    int result;
    alloc_chrdev_region(&my_dev,MINOR_BASE,1,DEVICENAME);
    cdev_init(&my_cdev,&led_driver_fops);
    cdev_add(&my_cdev,my_dev,1);
    my_class = class_create(THIS_MODULE,DEVICENAME); // /sys/class
    device_create(my_class,NULL,my_dev,NULL,DEVICENAME); // dev/NAME
    gpio_request(gpio_led,"sysfs");
    gpio_direction_output(gpio_led,ledValue);
    gpio_export(gpio_led,false);
    gpio_request(gpio_button,"sysfs");
    gpio_direction_input(gpio_button);
    gpio_export(gpio_button,false);
    printk(KERN_INFO"GPIO_TEST : The button state is currently
%d\n",gpio_get_value(gpio_button));
    //irqNum = gpio_to_irq(gpio_button);
    //printk(KERN_INFO"GPIO_TEST : The button is mapped to IRQ %d\n",irqNum);
    //result =
request_irq(irqNum,(irq_handler_t)button,IRQF_TRIGGER_RISING,"switch",NULL);
   /* if(result < 0){
        printk(KERN_INFO"GPIO_TEST : Failed to request IRQ %d\n", result);
        return result;
    }*/
    return 0;
}
```

```
static void button_exit(void){
    //free_irq(irqNum,NULL);
    gpio_unexport(gpio_button);
    gpio_free(gpio_button);
    gpio_unexport(gpio_led);
    gpio_free(gpio_led);
    device_destroy(my_class,my_dev);
    class_destroy(my_class);
    unregister_chrdev_region(my_dev,1);
}
module_init(button_init);
module_exit(button_exit);
```

---

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(){
    int fd;
    int value;
    bool input = true;
    fd = open("/dev/poll", O_RDWR);
    if(fd<0){
        printf("Unable to open file\n");
        printf("%d",errno);
        return 0;
    }
    while(1)
    {
        read(fd,&value,sizeof(int));
        if(value == 1)
        {
            write(fd,&input,sizeof(int));
            input = !input;
        }
        sleep(2);
    }
    close(fd);
    return 0;
}
```

Polling Device driver 는 기존의 interrupt code 에서 interrupt 관련 코드를 제거하고, user 와 통신하기 위한 read, write, open, release 함수를 정의하여 주었다. User application 은 간단하게 무한 루프를 돌며 2 초 주기로 read 를 하여 button 이 눌렸는지 확인하고 눌렸다면 write system call 을 보내 LED 값을 반전시킨다.

추가로, Kernel level 에서 Context-switch 와 같은 시간을 측정하는 방법은 없을까?

Polling 시간을 내지 않은 판단은 아래와 같다.

➔ Interrupt 와 비교하였을 때, Polling 이 갖는 단점은 busy-waiting 이 존재한다는 것이다. 어떤 event 가 발생하였다 했을 때, 일정 주기로 Check 를 실시하기 때문에 즉각성 측면에서 보았을 때 최악의 경우는 check 가 끝난 바로 직후 Event 가 발생하여 다음 주기를 기다려야 할 때이다. 또한, 현재의 Polling 구현 방식은 반드시 context-switch 가 필요하다. Read 를 device driver 에서 하기 때문에 context-switch 가 반드시 필요하므로 어떠한 경우에도 interrupt 보다 나은 결과를 도출하지 못한다.

위와 같은 구현 방식을 택했던 이유는, interrupt 와 polling 을 비교할 때 Context-switch 가 주요한 비교대상이라 생각하였기 때문이다. Interrupt_handler 안에서 처리하는 프로세스와 polling 으로 처리하는 프로세스는 같은 것이므로 이 프로세스 전후의 시간 차가 비교대상이 될 것이다. 하지만, 현재 보고자 하는 것은 polling 이 user 에서 kernel 쪽을 계속 관찰하고 있기 때문에 Interrupt 가 더 나은 결과를 만들 것이다. 만약, polling 을 kernel 내부에서 구현한다면 의미 있는 비교가 될 것이다. 하지만, kernel 내부에서 일종의 무한 루프를 돌리는 방법이 생각나지 않아 비교하지 못하였다.

# 4. Discussion

A. Network on Board

이번 주에 랜선을 추가로 얻어 보드에 연결하고, SSH 로 내부망을 통해 보드와 데스크탑 간의 원격 접속으로 코딩을 실시하였다. 확실히 SD 카드를 옮기고 할 때보다 시간 효율성이 많이 높아졌다. 무엇보다, 마우스&키보드 어댑터를 옮기지 않아도 되어 정말 좋았다...

B. PL configuration on device-tree

이전에도 계속해서 겪었던 문제사항이었다. 웹사이트 tutorial 들을 보면 booting log 에 XGpio@0xa000000 과 같이 vivado 에서 설정한 사항들이 등록되어지는데, 그것이 나오지 않아 의문사항이 있었다. 이전까지는 절대 주소를 밟아 통제가 가능했기 때문에 그다지 문제사항이 되지 않았지만, 이번에는 Interrupt 번호를 받는 과정에서 미리 등록되어 있는 인터럽트 번호를 알지 못해 이를 kernel 에 handler 를 등록하지 못하는 상황이 벌어졌다. Gpio API 를 사용하기 위해서는 Gpiochip number 가 필요하였는데, vivado 에서 설정된 사항들이 OS 쪽으로 반영되지 않아 이것이 불가능 하였다. 웹사이트를 찾아본 결과, Device-tree 에 등록된 사항들이 boot log 로 나오는 것을 확인하였고, 내 Device-tree 를 확인해본 결과 기본 device 들만 있고 vivado 에서 설정한 device 들은 존재하지 않았다. Kernel 을 setting 할 때, DTG setiing 이 있었고, 여기서 Remove PL on device tree 를 해제 해주어야 반영이

정상적으로 이루어진다. 이것이 이루어지면, /sys/class/gpio 에 PL 사항으로 등록된 번호의 gpiochip 이 생기고, echo number > export 를 하면 해당 번호의 gpio 를 직접 파일로 통제할 수 있는 디렉토리가 생성된다.