

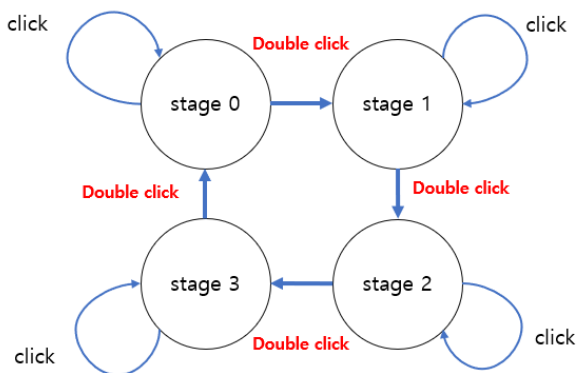
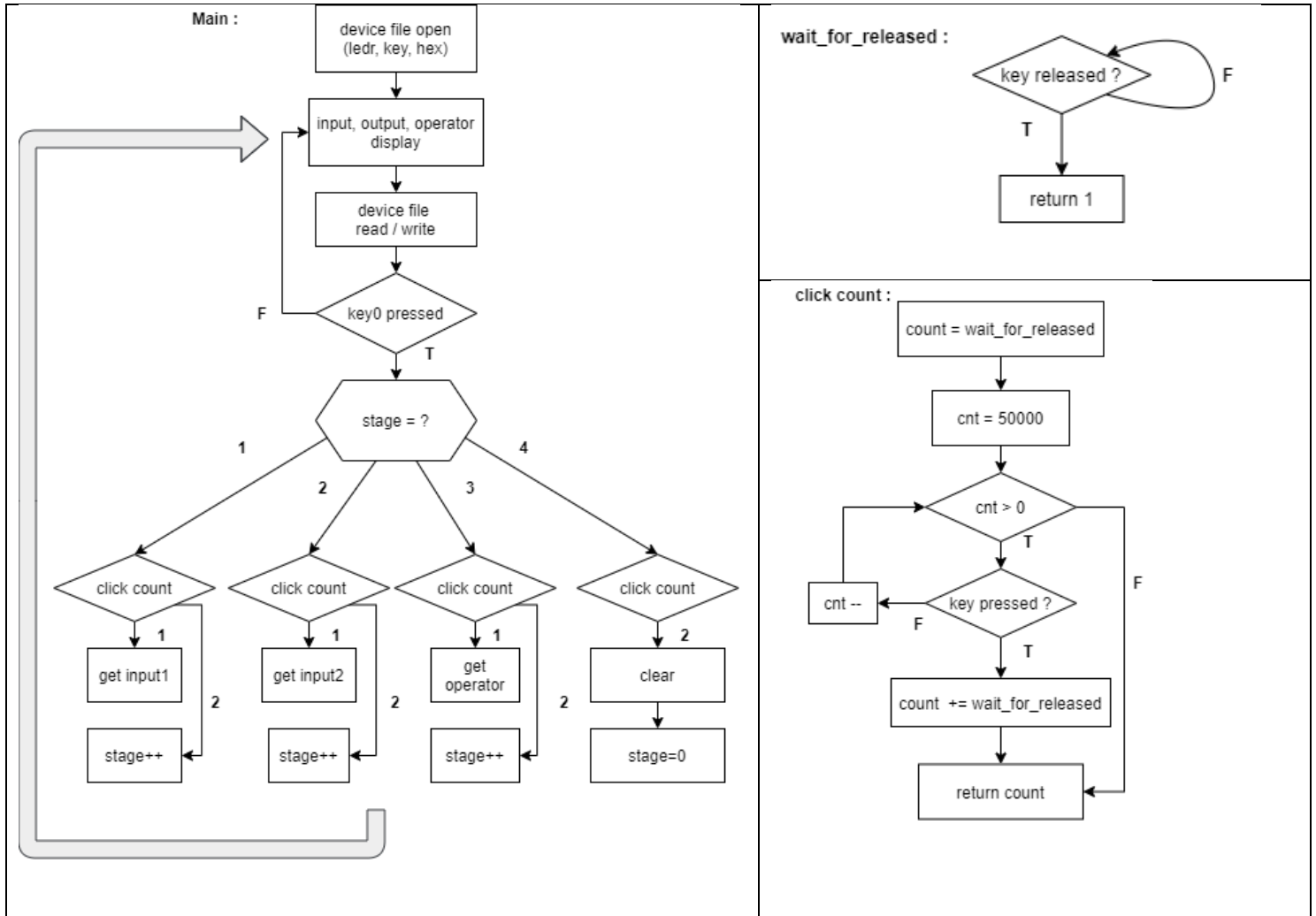
임베디드시스템 설계 및 실습 - 보고서

제출일: 2021-05-26; Lab 4

이름: 박관영(2016124099),김서영(2017106007),오한별(2018124120) ; 4조

✓ Assignment 1

개요



Stage 0 : get input1
 Stage 1 : get input2
 Stage 2 : get operation
 Stage 3 : clear

→ simple calculator의 state diagram

임베디드시스템 설계 및 실습 - 보고서

본 lab에서는 linux의 kernel에 대한 이해를 기반으로 Device driver를 만들고, Device driver를 사용하는 user program을 만드는 실험을 수행한다. Kernel이란 OS의 핵심적인 component로써 resource management를 담당하는 프로그램을 의미한다. 또한, Device driver는 kernel의 일부로써 device를 제어하는 프로그램을 의미한다. User program은 HW component에 직접 access 할 수 없기 때문에 system call을 통해 kernel이라는 service api를 사용하여 간접적으로 access하는 방식으로 수행이 된다.

본 lab에서는 parallel port에 연결된 IO device를 제어하는 user program을 사용하기 때문에 해당 IO device에 대한 device driver를 만들고, 그 device driver를 사용하는 user program을 만드는 방식으로 진행하였다. 이 때 Device driver는 kernel mode에서 동작하는 program이기 때문에 kernel에서 지원하는 device를 추가하기 위해서 Run time 중에 해당 module을 kernel에 삽입하는 LKM 방식으로 진행할 수 있었다. Kernel에 module을 적재한 후, device file을 생성하면 user program에서 해당 file을 open하는 것만으로 device에 access 할 수 있게 된다.

진행 내용

```
<<<user program>>>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <math.h>

volatile int segment(int);           // segment number display code
volatile int segment_op(char);       // segment operator display code
int wait_for_released(int, char*);   // polling key for released
int click_count(int, char*);         // click / double click detect

char operator;                       // display될 연산자
char operator_eq;                    // 등호

int main(int argc, char* argv[]){
    int hex3_hex0, hex5_hex4, key, ledr;           // device file descriptor

    int input1 = 0; int input2 = 0; int output = 0; // operand, output value

    int hex3_hex0_data = 0; int hex5_hex4_data = 0; // hex에 display될 값
    char key_data; char ledr_data;                 // key에서 읽을 값 / ledr에 display될 값

    int stage=0; int op_sel=0;                     // 현재 stage 값 / 수행할 연산을 결정하는 값

    int pressed_count;                             // key가 click된 횟수 (1 혹은 2)

    hex3_hex0 = open("/dev/hex3_hex0", O_WRONLY); // device file open
    hex5_hex4 = open("/dev/hex5_hex4", O_WRONLY);
    key = open("/dev/key", O_RDONLY);
    ledr = open("/dev/ledr", O_WRONLY);
    while(1){
```

임베디드시스템 설계 및 실습 - 보고서

```
hex3_hex0_data = segment(output) | segment_op(operator_eq)<<8 | ₩
    segment(input2)<<16 | segment_op(operator)<<24;    // set hex3-0 display value
hex5_hex4_data = segment(input1);                    // set hex5-4 display value
ledr_data = (int)pow(2, op_sel);                      // set ledr display value

write(hex3_hex0, &hex3_hex0_data, 4);                // hex3_hex0에 data write
write(hex5_hex4, &hex5_hex4_data, 4);                // hex5_hex4에 data write
read(key, &key_data, 1);                             // key에서 data read
write(ledr, &ledr_data, 1);                          // ledr에 data write

if ((key_data & 0x1)!=0){                            // key0 check

    if (stage==0){                                    // stage0 : get input1

        pressed_count = click_count(key, &key_data); // click / double click check
        if(pressed_count==1) input1++;               // get input1
        if(pressed_count==2) stage = (stage + 1) % 4; // Next stage

    }
    else if (stage==1){                                // stage1 : get input2

        pressed_count = click_count(key, &key_data); // click / double click check
        if(pressed_count==1) input2++;               // get input2
        if(pressed_count==2) stage = (stage + 1) % 4; // Next stage

    }
    else if (stage==2){                                // stage2 : get operator

        pressed_count = click_count(key, &key_data); // click / double click check
        if(pressed_count==1) op_sel = (op_sel + 1) % 4; // operator select
        if(pressed_count==2) {                        // selected operation 실행, operator display
            switch(op_sel){
                case 0: output = input1 + input2; operator = '+'; break;
                case 1: output = input1 - input2; operator = '-'; break;
                case 2: output = input1 * input2; operator = '*'; break;
                case 3: output = input1 / input2; operator = '/'; break;
            }
            operator_eq = '=';                        // 등호 display
            stage = (stage + 1) % 4;                  // Next stage
        }
    }
}

else if (stage==3){                                    // stage3 : clear

    pressed_count = click_count(key, &key_data);     // click / double click check
    if(pressed_count==2) {                            //clear
```

임베디드시스템 설계 및 실습 - 보고서

```
        input1 = 0; input2 = 0; output = 0;
        operator = 0; operator_eq = 0; op_sel=0;
        stage = (stage + 1) % 4;                                // Next stage
    }
}

}

return 0;
}

volatile int segment(int num){
    volatile int seg_value;
    seg_value = (num<=0) ? 0x3f :W
                (num==1) ? 0x06 :W
                (num==2) ? 0x5b :W
                (num==3) ? 0x4f :W
                (num==4) ? 0x66 :W
                (num==5) ? 0x6d :W
                (num==6) ? 0x7d :W
                (num==7) ? 0x07 :W
                (num==8) ? 0x7f : 0x6f;
    return seg_value;
}

volatile int segment_op(char op){
    volatile int seg_value;
    seg_value = (op=='+') ? 0b1110011 :W
                (op=='-') ? 0b1000000 :W
                (op=='*') ? 0b1110110 :W
                (op=='/') ? 0b1011110 :W
                (op=='=') ? 0b0001001 : 0x0;
    return seg_value;
}

int wait_for_released(int fd, char* key_data){

    int pressed_count=0;
    while(1){
        read(fd, key_data, 1);                                // polling key
        if(*key_data==0) {pressed_count=1; break;}           // if key released, pressed_count = 1 → break
    }
    return pressed_count;                                      // pressed_count return
}

int click_count(int fd, char* key_data){
```

임베디드시스템 설계 및 실습 - 보고서

```
int cnt=50000; // 실험적으로 결정된 값
int pressed_count=0;
pressed_count=wait_for_released(fd, key_data); // key pressed event 발생
while(cnt>0){ // 일정 시간 내에 key0 pressed event가 다시 발생하는 지 확인
    read(fd, key_data, 1); // polling key
    if(*key_data==1) { // key0 check
        pressed_count += wait_for_released(fd, key_data); // pressed_count++ → break
        break;
    }
    cnt--;
}
return pressed_count; // key pressed event가 발생한 횟수 return
}
```

<<<DEVICE DRIVER LKM (read / write)>>>

/* character device driver이다. 임의의 device에 대해서 작성하였으므로 device name은 dev로 통일함. */

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/ioport.h>
```

```
#include <asm/uaccess.h>
#include <asm/io.h>
```

```
#include "address_map_arm.h"
```

```
#define LICENSE "KAU_EMBEDDED"
#define AUTHOR "THKIM"
#define DESCRIPTION "KAU EMBEDDED SYSTEMS LAB ASSIGNMENT 1"
```

```
#define DEV_MAJOR          0
#define DEV_NAME           "DEV"
#define DEV_MODULE_VERSION "DEV v0.1"
```

```
static int DEV_major = 0;
static volatile int* DEV_ptr; // device address
```

```
int DEV_open(struct inode *minode, struct file *mfile){ // open function (device file open시 수행됨)
    DEV_ptr = ioremap_nocache (LW_BRIDGE_BASE, LW_BRIDGE_SPAN) + BASE_addr; // virtual address mapping
    printk(KERN_INFO "[DEV_open]\n"); //kernel log print
    return 0;
}
```

```
int DEV_release(struct inode *minode, struct file *mfile){ // release function (device file close시 수행됨)
```

임베디드시스템 설계 및 실습 – 보고서

```
printk(KERN_INFO "[DEV release]\n");           // kernel log print
return 0;
}

ssize_t DEV_read_byte(struct file *inode, char *gdata, size_t length, loff_t *off_what){ // read function (device file을 read 시 수행됨)

    unsigned char c;
    c = *DEV_ptr;           // virtual address mapping된 device에서 값을 read
    put_user(c, gdata);     // user space로 data passing (1byte)

    printk(KERN_INFO "[DEV_read_byte] %d\n", c);
    return length;
}

// write function (device file에 write 시 수행됨)
ssize_t DEV_write_byte(struct file *inode, const char *gdata, size_t length, loff_t *off_what){

    unsigned char c;
    get_user(c, gdata);     // user space로부터 data passed (1byte)
    *DEV_ptr = c;           // virtual address mapping된 device에 값을 write
    printk(KERN_INFO "[DEV_write_byte] %d\n", c);
    return length;
}

static struct file_operations DEV_fops = {       // file operation 구조체에 해당 device에서 사용할 함수들을 등록
    .owner      = THIS_MODULE,
    .read       = DEV_read_byte,
    .write      = DEV_write_byte,
    .open       = DEV_open,
    .release    = DEV_release,
};

int DEV_init(void){                             // call-back function (해당 module이 initialize될 때 call)
    int result = register_chrdev(DEV_MAJOR, DEV_NAME, &DEV_fops); // character device 등록
    if(result<0){
        printk(KERN_WARNING "Can't get any major\n");
        return result;
    }
    DEV_major = result;
    printk(KERN_INFO "[DEV_init] major number : %d\n", result);
    return 0;
}

void DEV_exit(void){                             // call-back function (해당 module이 exit될 때 call)
    printk(KERN_INFO "[DEV_exit]\n");
    unregister_chrdev(DEV_major, DEV_NAME);      // character device 해제
}
```

임베디드시스템 설계 및 실습 - 보고서

```
module_init(DEV_init);      // init call-back function 등록
module_exit(DEV_exit);     // exit call-back function 등록

MODULE_LICENSE(LICENSE);
MODULE_AUTHOR(AUTHOR);
MODULE_DESCRIPTION(DESCRIPTION); // module information
```

→ file operation structure

```
static struct file_operations
{
    struct module *owner;
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    .
    .
    .
};
```

File system을 통해 character device driver와 user program을 연결시켜주는 구조체이다. <linux/fs.h>에 정의되어 있고, 위와 같은 형태로 정의되어 있다. File operation structure에 정의되어 있는 field 중에서 해당 module이 사용할 field에 함수 포인터를 등록하는 형태로 사용하게 된다. User program에서 해당 device file을 open하여 특정 기능을 사용하게 되면, call-back function과 비슷하게 file operation structure에 등록된 함수가 call되는 형식으로 구현이 된다. 즉, file structure는 User program이 IO device를 file 형태로 access할 수 있도록 하는 user interface 역할을 한다.

→ Register_chrdev

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)
```

Device의 major number와 name, file operation structure를 받아 해당 device driver를 등록하고, major number를 반환하는 함수이다. 이 때, major의 값이 0이라면 major number는 자동으로 할당되고, 반환되는 major number는 0부터 255사이의 값이다.

→ ioremap_nocache

```
void * ioremap_nocache(unsigned long offset, unsigned long size);
```

Mmap과 비슷하게 physical address를 virtual address로 mapping하는 함수이다. Kernel도 user program과 마찬가지로 virtual address space에서 동작하기 때문에 특정 device에 access하기 위해서는 해당 device의 physical address를 virtual address로 mapping하는 과정이 필요하다. User space에서 사용하는 mmap 함수는 kernel에 mapping을 요청하는 system call이지만, device driver는 이미 kernel mode에서 동작하고 있기 때문에 mmap 함수와 구분되는 ioremap_nocache 함수를 사용한다. 함수명에 _nocache가 붙은 이유는 cache non-coherence problem을 해소하기 위함이다.

임베디드시스템 설계 및 실습 - 보고서

```
ssize_t HEX5_HEX4_write_byte(struct file *inode, const char *gdata, size_t length, loff_t *off_what){  
  
    unsigned char a, b;  
    get_user(a, gdata);  
    get_user(b, gdata+1);  
  
    *HEX5_HEX4_ptr = a | b<<8;  
    printk(KERN_INFO "[HEX5_HEX4_write_byte] %d\n", b);  
    return length;  
}
```

```
ssize_t HEX3_HEX0_write_byte(struct file *inode, const char *gdata, size_t length, loff_t *off_what){  
  
    unsigned char a, b, c, d;  
    get_user(a, gdata);  
    get_user(b, gdata+1);  
    get_user(c, gdata+2);  
    get_user(d, gdata+3);  
    *HEX3_HEX0_ptr = a | b<<8 | c<<16 | d<<24;  
    printk(KERN_INFO "[HEX3_HEX0_write_byte] %d\n", c);  
    return length;  
}
```

➔ 위의 두 함수는 hex device driver에서 사용되는 write 함수이다. Hex는 한 device가 8비트 이상의 영역을 차지하고 있기 때문에 다음과 같이 여러 번에 걸쳐서 data를 write하는 방식으로 함수를 구현하였다. Hex3_Hex0 device는 4byte의 값을 읽고, 각 byte의 data를 순서대로 display하는 방식으로 구현했고, Hex5_Hex4 device는 2byte의 값을 읽고 순서대로 display하는 방식으로 구현했다.

진행 결과

1. Makefile (build the kernel module)

```
obj-m += key_dd.o  
obj-m += led_r_dd.o  
obj-m += hex3_hex0_dd.o  
obj-m += hex5_hex4_dd.o  
  
all:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  
clean:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
root@delsoclinux:~/LAB4/assignment1# make  
make -C /lib/modules/3.18.0/build M=/home/root/LAB4/assignment1 modules  
make[1]: Entering directory `/usr/src/3.18.0'  
CC [M] /home/root/LAB4/assignment1/key_dd.o  
CC [M] /home/root/LAB4/assignment1/led_r_dd.o  
CC [M] /home/root/LAB4/assignment1/hex3_hex0_dd.o  
CC [M] /home/root/LAB4/assignment1/hex5_hex4_dd.o  
Building modules, stage 2.  
MODPOST 4 modules  
CC /home/root/LAB4/assignment1/hex3_hex0_dd.mod.o  
LD [M] /home/root/LAB4/assignment1/hex3_hex0_dd.ko  
CC /home/root/LAB4/assignment1/hex5_hex4_dd.mod.o  
LD [M] /home/root/LAB4/assignment1/hex5_hex4_dd.ko  
CC /home/root/LAB4/assignment1/key_dd.mod.o  
LD [M] /home/root/LAB4/assignment1/key_dd.ko  
CC /home/root/LAB4/assignment1/led_r_dd.mod.o  
LD [M] /home/root/LAB4/assignment1/led_r_dd.ko  
make[1]: Leaving directory `/usr/src/3.18.0'
```

➔ 'make' build tool을 위한 build script이다. C source code를 compile하는 것과 마찬가지로 gcc에 여러가지 옵션을 통해 device driver를 compile 할 수 있지만, 보다 간편하게 compile하기 위해서 다음과 같이 Makefile을 사용하여 compile한다. 이후 'make' 명령만으로 모든 device driver의 source file을 compile하고 'make clean' 명령만으로 모든 compile된 결과물들을 삭제할 수 있다. Device driver를 compile한 결과물 중 .ko (kernel object) file을 kernel에 적재함으로써 kernel의 기능을 확장할 수 있다.

임베디드시스템 설계 및 실습 - 보고서

2. Insert the module

```
root@delsoclinux:~/LAB4/assignment1# insmod key_dd.ko
root@delsoclinux:~/LAB4/assignment1# insmod hex3_hex0_dd.ko
root@delsoclinux:~/LAB4/assignment1# insmod hex5_hex4_dd.ko
root@delsoclinux:~/LAB4/assignment1# insmod ledr_dd.ko

root@delsoclinux:~/LAB4/assignment1# dmesg | tail -5
[ 135.166576] Disabling lock debugging due to kernel taint
[ 135.166992] [KEY_init] major number : 248
[ 139.762997] [HEX3_HEX0_init] major number : 247
[ 146.572923] [HEX5_HEX4_init] major number : 246
[ 152.192961] [LEDR_init] major number : 245
```

→ 'insmod' 명령을 통해 kernel object file을 kernel에 install 할 수 있다. 'dmesg' 명령을 통해 kernel log를 확인할 수 있고, module이 정상적으로 install 되어서 device driver에서 등록한 call-back function이 수행되어 kernel log에 print 됨을 확인할 수 있다. 또한, Device driver의 major number를 자동할당 하도록 설정하였기 때문에 kernel에 install한 순서대로 할당이 된 것을 확인할 수 있다.

3. make device file

```
root@delsoclinux:~/LAB4/assignment1# mknod /dev/key u 248 0
root@delsoclinux:~/LAB4/assignment1# mknod /dev/hex3_hex0 u 247 0
root@delsoclinux:~/LAB4/assignment1# mknod /dev/hex5_hex4 u 246 0
root@delsoclinux:~/LAB4/assignment1# mknod /dev/ledr u 245 0
```

→ 'mknod' 명령을 사용하여 user space에서 access 할 수 있는 device file을 생성할 수 있다. 이 때 file type / major number / minor number를 통해서 device file의 type과 major-minor number를 특정할 수 있다. 본 lab에서는 byte 단위의 data passing만 이루어지므로 character type의 device file을 생성하게 된다. 또한 device driver를 공유하는 device는 없으므로 minor number는 구분할 필요가 없지만, device file을 통해 open 하고자 하는 device driver를 특정하기 위해서 device file의 major number를 open 하고자 하는 device driver의 major number로 설정한다.

4. user program compile / execute

```
root@delsoclinux:~/LAB4/assignment1# gcc -c main.c -o main.o -marm
root@delsoclinux:~/LAB4/assignment1# gcc main.o -lm -o main
root@delsoclinux:~/LAB4/assignment1# ./main
```

→ gcc를 통해 user program을 compile & link 한 뒤 실행한다. 이 때 user program의 pow 함수를 사용하는 과정에서 라이브러리 사용을 명시하기 위해 -lm option을 통해 link하는 것이 필요했다.

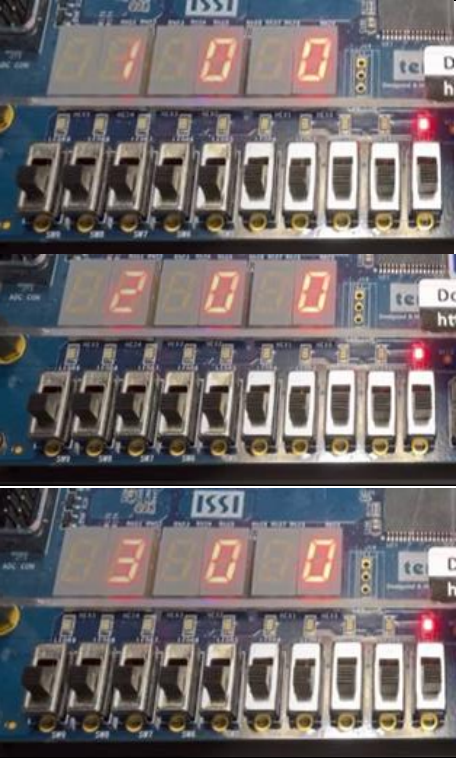
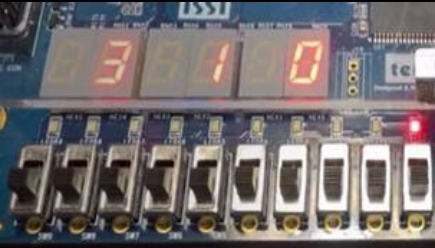
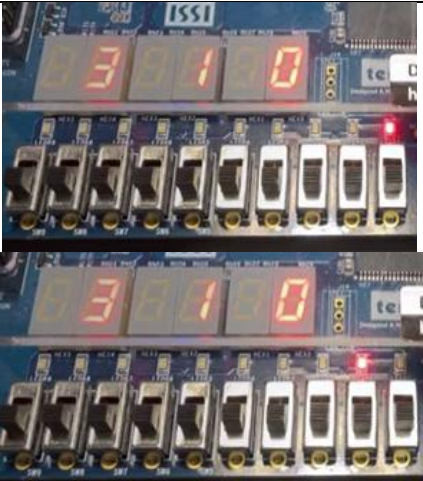

5. Demo



HEX5 : input1 / HEX3 : input2 / HEX0 : output ; LEDR : 0 ON - plus / 1 ON - minus / 2 ON - multiply / 3 ON - divide

다음과 같이 초기 상태가 display된다. 처음에는 모든 값이 0으로 초기화 되어있으므로 모든 자리가 0으로 display된다.

임베디드시스템 설계 및 실습 - 보고서

	click	Double click
Stage0	<div></div> <div>Stage1에서 key0를 click 함으로써 input1을 입력 받고, 다음과 같이 display되는 것을 볼 수 있다.</div>	Stage 천이
Stage1	<div></div> <div>Stage2에서 key0를 click 함으로써 input2를 입력 받고, 다음과 같이 display되는 것을 볼 수 있다.</div>	Stage 천이
Stage2	<div></div> <div>Stage3에서 key0를 click함으로써 ledr이 ON되는 자리가 달라지게 되고 이를 통해 어떤 연산이 수행될 지를 알 수 있다.</div>	<div></div> <div>Stage3에서 key0를 double click 함으로써 선택된 연산 결과와 operator들이 display되는 것을 확인할 수 있다. + stage 천이</div>

임베디드시스템 설계 및 실습 - 보고서

Staage3	No operation	 <p>Stage4에서 key0를 double click 함으로써 초기상태로 돌아가는 것을 확인할 수 있다. + stage 천이</p>
---------	--------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------

→ Demo video :

<https://www.youtube.com/watch?v=-X-nFEaNtLY>

결과 분석 및 팀원 간 토의 사항

ARM architecture 기반의 DE1-SoC HPS platform에서 Device driver를 통해 file 형태로 device에 access하는 user program과 device driver를 구현해 보았다. 지금까지의 lab은 user program 상에서의 동작으로 어떠한 프로그램을 구현했다면, 이번 lab은 linux OS 기반 system에서 핵심적인 요소인 kernel에 어떠한 module을 적재하여 특정 기능을 확장하는 과정을 통해 user program에서 HW component에 간접적으로 access할 수 있도록 하는 실험을 진행하였다.

전반적인 program flow는 이전 lab과 비슷하게 흘러갔지만, MMIO를 통해 program을 구현하였을 때는 system call을 통해 access하고자 하는 device의 physical address를 system call 형태의 함수(mmap)를 사용하여 virtual address로 mapping 한 뒤 device에 access 했다. 반면 device driver를 사용하여 device를 access 한다면, 단순히 file open 형태로 device에 access할 수 있게 된다. 또한, 최소한의 key를 사용하기 위해 key0를 누르는 시간을 측정하여 stage를 천이하는 방식을 사용했으나 좀 더 간단한 click / double click 방식으로 바꾸어 프로그램의 user interface를 편리하게 할 수 있었다.

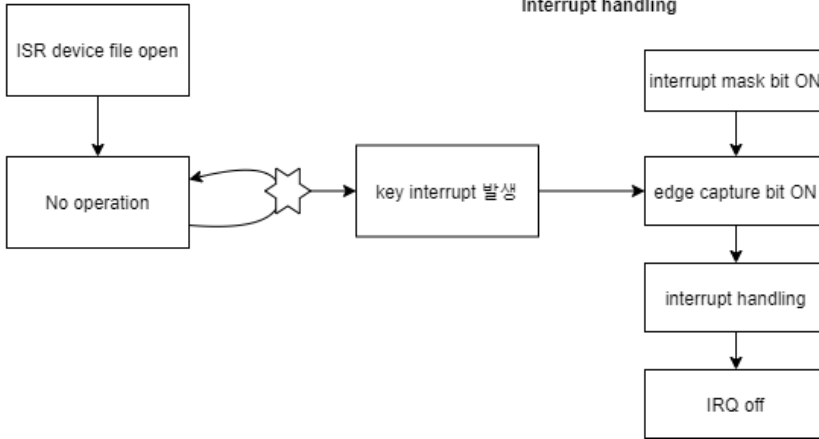
Bare-metal system이 아니라면, user program의 단순화를 위해 device driver를 통해 device control을 하는 것이 좀 더 바람직한 것으로 보인다.

임베디드시스템 설계 및 실습 - 보고서

✓ Assignment 2

개요

user space



본 lab에서는 user program이 interrupt I/O를 사용할 수 있도록 하는 예시 프로그램을 작성한다. 이를 위해 Interrupt 를 handling하는 모듈을 LKM 방식으로 만든 후 user 프로그램이 해당 file을 읽음으로써 interaction이 가능하도록 한다. Interrupt 방식은 다음과 같이 동작한다. 먼저, routine을 실행하고 있는 도중에 입출력 장치에서 필요할 때 마다 즉각적으로 CPU에게 interrupt 신호를 전송한다. 해당 interrupt가 요청한 작업을 실행하기 위해 하고 있는 동작을 멈추고 ISR (Interrupt Service Routine)으로 이동한다. ISR이 완료되면 CPU는 수행을 멈춘 곳으로 되돌아가 중단된 작업을 계속 진행한다. Interrupt control은 kernel mode에서 진행하므로 interrupt I/O를 받아서 user mode에서 특정 call back function을 사용하므로 Busy wait I/O보다 더 복잡한 형태의 coding이 필요하다.

이러한 interrupt I/O의 특성을 바탕으로 key가 push 될 때 마다 interrupt가 발생하고 값이 하나씩 증가하며 이를 LED로 표시하는 프로그램을 작성한다. 프로그램의 flow는 왼쪽과 같다.

이를 구현하기 위해 총 두 개의 c파일이 필요하다. 먼저 interrupt ISR을 kernel에 추가하여 interrupt가 발생할 때 supervisor mode에서 실행할 수 있도록 하는 key_isr.c 소스코드를 작성하고, 다음으로 user mode에서 동작하는 application을 작성하여 key_isr file에 access하여 interrupt I/O를 사용할 수 있도록 한다.

진행 내용

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <linux/fs.h>
#include <linux/ioport.h>
#include "address_map_arm.h"
#include "interrupt_ID.h"
```

임베디드시스템 설계 및 실습 - 보고서

```
#define LICENSE "KAU_EMBEDDED"
#define AUTHOR "THKIM"
#define DESCRIPTION "KAU EMBEDDED SYSTEMS LAB EXAMPLE"

#define ISR_MAJOR          0
#define DEV_NAME           "ISR"

void* LW_virtual;
volatile int* LEDR_ptr;
volatile int* KEY_ptr;
static int ISR_major = 0;

irq_handler_t irq_handler(int irq, void* dev_id, struct pt_regs *regs)
{
    *LEDR_ptr = *LEDR_ptr + 1;
    *(KEY_ptr + 3) = 0xf;    // clear edgecapture register

    return (irq_handler_t) IRQ_HANDLED;
}

static int ISR_open(struct inode *minode, struct file *mfile){

    int value;

    //Map the physical addr of H2F Bridge to a virtual address
    LW_virtual = ioremap_nocache (LW_BRIDGE_BASE, LW_BRIDGE_SPAN);

    LEDR_ptr = LW_virtual + LEDR_BASE;
    *LEDR_ptr = 0x0;

    KEY_ptr = LW_virtual + KEY_BASE;

    //Clear the PIO edgecapture register (Clear any pending interrupt)
    *(KEY_ptr + 3) = 0x1;
    //Enable IRQ generation for the 4 buttons
    *(KEY_ptr + 2) = 0x1;

    //Register the interrupt handler
    value = request_irq(KEYS_IRQ, (irq_handler_t) irq_handler, IRQF_SHARED, "pushbutton_irq_handler", (void*) (irq_handler));

    return value;
}

static int ISR_release(struct inode *minode, struct file *mfile){
```

임베디드시스템 설계 및 실습 - 보고서

```
*LEDR_ptr = 0;
free_irq(KEYS_IRQ, (void*) irq_handler);
printk(KERN_INFO "[KEY release]\n");
}

//Call-back functions for the file IOs.
static struct file_operations ISR_fops = {
    .owner          = THIS_MODULE,
    .open           = ISR_open,
    .release        = ISR_release,
};

static int __init initialize_pushbutton_handler(void)
{
    int result = register_chrdev(ISR_MAJOR, DEV_NAME, &ISR_fops);
    if(result<0){
        printk(KERN_WARNING "Can't get any major\n");
        return result;
    }
    ISR_major = result;
    printk(KERN_INFO "[ISR_init] major number : %d\n", result);

    return 0;
}

static void __exit cleanup_pushbutton_handler(void)
{
    printk(KERN_INFO "[ISR_exit]\n");
    unregister_chrdev(ISR_major, DEV_NAME);
}

module_init(initialize_pushbutton_handler);
module_exit(cleanup_pushbutton_handler);

MODULE_LICENSE(LICENSE);
MODULE_AUTHOR(AUTHOR);
MODULE_DESCRIPTION(DESCRIPTION);

//user application
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

임베디드시스템 설계 및 실습 - 보고서

```
int main()
{
    FILE* fd = fopen("/dev/key_isr", "r"); //읽기 형식으로 파일 열음

    while(1);
}
```

각 함수 별 진행하는 내용은 다음과 같다.

```
static struct file_operations ISR_fops = {
    .owner          = THIS_MODULE,
    .open           = ISR_open,
    .release        = ISR_release,
};
```

→ call-back function을 정의해 둔 구조체로 user mode에서 모듈이 정의된 파일을 open하면 ISR_open 함수가 실행되고, 파일을 close하면 ISR_release 함수가 실행된다. ISR_open함수는 KEY interrupt 가 발생할 때 마다 LEDR값을 하나씩 증가시키는 Interrupt request를 return한다. File operation 구조체를 통해 기존 program과 다르게 user space에서 device file 형태로 access할 수 있도록 user interface를 형성한다. ISR_open 함수와 interrupt request를 수행하는 irq_handler함수는 다음과 같다.

```
static int ISR_open(struct inode *minode, struct file *mfile){
    int value;

    LW_virtual = ioremap_nocache (LW_BRIDGE_BASE, LW_BRIDGE_SPAN);

    LEDR_ptr = LW_virtual + LEDR_BASE;
    *LEDR_ptr = 0x0;

    KEY_ptr = LW_virtual + KEY_BASE;

    *(KEY_ptr + 3) = 0x1;
    *(KEY_ptr + 2) = 0x1;

    value = request_irq(KEYS_IRQ, (irq_handler_t) irq_handler, IRQF_SHARED, "pushbutton_irq_handler", (void*) (irq_handler));

    return value;
}
```

```
irq_handler_t irq_handler(int irq, void* dev_id, struct pt_regs *regs)
{
    *LEDR_ptr = *LEDR_ptr + 1;
    *(KEY_ptr + 3) = 0xf;

    return (irq_handler_t) IRQ_HANDLED;
}
```

→ ISR_open) HPS에서 Light Weight bridge를 사용하고 있으므로 KEY와 LEDR의 pointer의 offset을 설정한다. KEY interrupt를 처리하기 위해 KEY의 interrupt mask register를 IRQ enable로 설정하고 edge capture register를 초기화한다. 그 후 KEY가 push 되었는지 여부에 따라 interrupt를 발생시켜 해당 request를 처리하는 interrupt handler를 등록한다.

임베디드시스템 설계 및 실습 - 보고서

```
static int ISR_release(struct inode *minode, struct file *mfile){
    *LEDR_ptr = 0;
    free_irq(KEYS_IRQ, (void*) irq_handler);
    printk(KERN_INFO "[KEY release]\n");
}
```

→ ISR_release) LEDR을 초기화하고 free_irq를 통해 interrupt service 함수를 해제하며 printk로 kernel log를 남겨 KEY에 의한 interrupt가 제거되었음을 표시한다.

```
static int __init initialize_pushbutton_handler(void)
{
    int result = register_chrdev(ISR_MAJOR, DEV_NAME, &ISR_fops);
    if(result<0){
        printk(KERN_WARNING "Can't get any major\n");
        return result;
    }
    ISR_major = result;
    printk(KERN_INFO "[ISR_init] major number : %d\n", result);

    return 0;
}

static void __exit cleanup_pushbutton_handler(void)
{
    printk(KERN_INFO "[ISR_exit]\n");
    unregister_chrdev(ISR_major, DEV_NAME);
}

module_init(initialize_pushbutton_handler);
module_exit(cleanup_pushbutton_handler);
```

→ 모듈을 생성 및 추가하는 부분이며 모듈이 생성 및 삭제될 때 Kernel log를 남기도록 하였다. 또한 interrupt handling device driver를 character device와 같은 형태로 등록하여 file 형태로 user space에서 interaction이 가능하도록 하였다.

진행 결과

1. Makefile (build the kernel module)

```
GNU nano 2.2.6      File: Makefile
obj-m +=key_isr.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
root@delsoclinux:~/lab4# make
make -C /lib/modules/3.18.0/build M=/home/root/lab4 modules
make[1]: Entering directory `/usr/src/3.18.0'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory `/usr/src/3.18.0'
```


임베디드시스템 설계 및 실습 - 보고서

```
root@delsoclinux:~/lab4# ls
Makefile      assignment2_main.c  key_isr.ko      key_isr.o
Module.symvers  interrupt_ID.h      key_isr.mod.c   modules.order
address_map_arm.h  key_isr.c          key_isr.mod.o
```

→ assignment1에서와 마찬가지로 Makefile이라는 build script를 통해 C 소스파일을 간편하게 compile한다. make 명령어로 지정된 source file을 컴파일 하면 다음과 같은 파일들이 생성됨을 확인할 수 있다.

Module.symvers	심볼 테이블의 이름으로 심볼테이블은 컴파일러 또는 interpreter에서 사용되는 데이터 구조이다. 심볼 테이블은 모듈을 컴파일하면 해당 디렉토리에 생성되며 함수의 위치 정보를 가지고 있다.
Key_isr.ko	Ko는 kernel object로 kernel에 load하여 kernel에서 수행하는 기능을 확장한다.
Key_isr.mod.c	Key_isr 모듈과 관련된 자료구조와 선언이 정의되어 있는 파일이다.
Key_isr.mod.o	Key_isr.c 파일의 object 파일로 다른 .o 파일을 linking 하여 .ko 파일을 작성한다.
Modules.order	Makefile에 기술된 module들의 순서가 저장된다.

2. Insert the module

```
root@delsoclinux:~/lab4# insmod key_isr.ko
root@delsoclinux:~/lab4# dmesg | tail -5
[ 6409.763137] [ISR_init] major number : 248
```

→ 명령어 insmod를 통해 .ko (kernel object) 파일을 kernel에 install 한다. Kernel log를 출력하는 dmesg 명령어로 확인해보면 다음과 같이 major number가 설정되었음을 확인할 수 있다.

3. make device file

```
root@delsoclinux:~/lab4# mknod /dev/key_isr c 248 0
```

```
root@delsoclinux:~/lab4# cat /proc/devices
Character devices:
1 mem
2 pty
3 tty
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
89 i2c
90 mtd
128 ptm
136 pts
153 spi
180 usb
189 usb device
248 ISR
249 ozwpan
250 bsg
251 ptp
252 pps
253 rtc
254 fpga
```

```
#define DEV_NAME "ISR"
```

→ mknod 명령어를 사용하여 디바이스 파일을 생성한다. assignment1과 마찬가지로 file type / major number / minor number

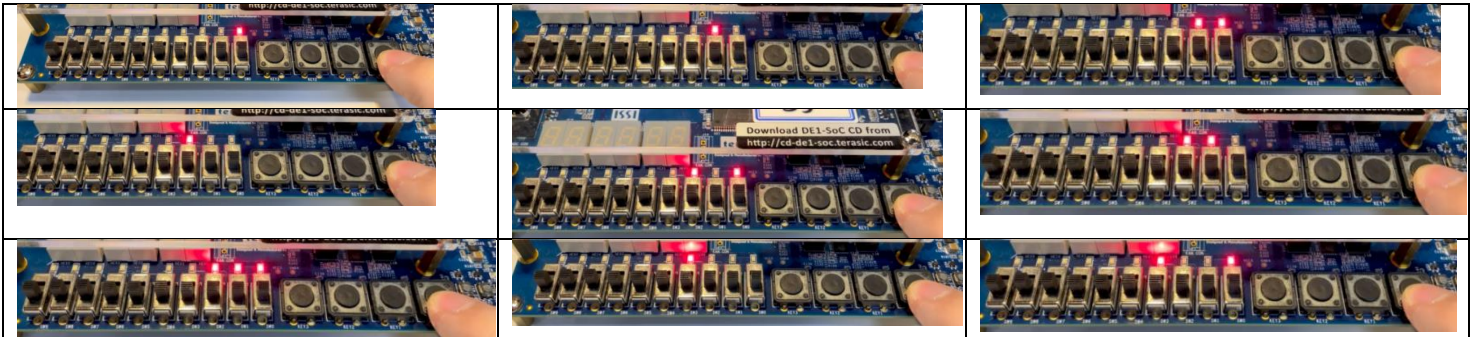
임베디드시스템 설계 및 실습 - 보고서

형식으로 파일을 생성한다. 리눅스 시스템에서 디바이스 파일은 "~/dev" 디렉토리에 있으며 kernel은 해당 디렉토리에 있는 파일을 하드웨어라고 간주한다. 따라서 User mode에서 동작하는 application은 "~/dev" 경로로 파일을 읽음으로써 하드웨어를 컨트롤 할 수 있다. 만약 디바이스 파일을 삭제하고 싶다면 "rm -rf /dev/[파일이름]" 명령어를 통해 삭제할 수 있다. 또한 cat /proc/devices 명령어를 통해 지정된 device name으로 device에 추가되어 있음을 확인할 수 있다.

4. user program compile / execute

```
root@delsoclinux:~/lab4# gcc -c assignment2_main.c -o assignment2_main.o -marm
root@delsoclinux:~/lab4# gcc assignment2_main.o -lm -o assign2
root@delsoclinux:~/lab4# ./assign2
```

→ gcc 컴파일러를 사용하여 user program을 compile & link 한 뒤 실행한다. User program이 진행될 때는 interrupt I/O에 access 할 수 있다.



실행 결과는 상단과 같으며 KEY0을 push할 때 마다 값이 하나씩 증가하며 LEDR에 표시됨을 확인할 수 있다.

```
root@delsoclinux:~/lab4# dmesg | tail -10
[ 10.249435] init: ttyl main process (1469) killed by TERM signal
[ 373.851014] key_isr: module license 'KAU_EMBEDDED' taints kernel.
[ 373.851030] Disabling lock debugging due to kernel taint
[ 373.851470] [ISR_init] major number : 248
[ 752.116668] [ISR_exit]
[ 757.833019] [ISR_init] major number : 248
[ 1079.044498] [KEY release]
[ 1991.115204] [KEY release]
[ 2059.348725] [KEY release]
[ 2157.024542] [KEY release]
```

→ user program을 종료시킨 뒤 kernel log를 확인해보면 다음과 같이 [KEY_release]가 되어 interrupt I/O 에 access할 수 없다.

결과 분석 및 팀원 간 토의 사항

User mode와 Kernel mode의 차이점에 대해 학습할 수 있었고, HW 제어 방식 중 하나인 interrupt I/O방식을 LKM으로 구현해 봄으로써 리눅스 시스템의 kernel에 대한 이해도를 높일 수 있었다. 특히 Makefile을 적절히 기술하여 여러 모듈을 동시에 효과적으로 compile할 수 있다는 것을 알게 되었고, 이를 기반으로 다양한 기능의 module을 만들어 kernel 상에서 처리할 수 있도록 하는 방법을 학습하였다.

본 설계에서는 Key interrupt 만을 사용하였지만, 더 많은 I/O device를 사용한다면 busy wait I/O 방식으로 구현할 시 한계가 발생할 것으로 보이며 CPU의 낭비를 방지하기 위해 Interrupt I/O 방식으로 I/O device를 제어하는 것이 효율적일 것으로 생각된다. 이러한 interrupt 방법을 통한 I/O 제어를 위해 irq handler를 등록 및 제거하는 함수인 request_irq와 free_irq의 사용법을 익혔으

임베디드시스템 설계 및 실습 - 보고서

며, 다음과 같은 함수의 원형을 바탕으로 lab을 진행할 수 있었다.

1) request_irq

```
#include <linux/interrupt.h>

int request_irq(unsigned int irq, irqreturn_t(*handler)(int, void*, struct pt_regs*),
               unsigned long flags, const char* device, void* dev_id);
```

Irq	Interrupt service 함수를 등록하고자 하는 interrupt 번호
Handler	Interrupt가 발생하였을 때 처리하는 Interrupt service 함수의 포인터
Flags	Interrupt의 옵션을 지정
Device	Interrupt 소유자에 대한 문자열
Dev_id	Interrupt 공유시에 interrupt를 구분하는 ID로 사용되거나 interrupt handler에 전달될 data 주소 지정에 사용된다.

Interrupt가 정상적으로 등록되면 0을, 비정상적으로 등록되면 음수 값을 반환한다. Assignment2에서 작성한 코드는 다음과 같다.

```
value = request_irq(KEYS_IRQ, (irq_handler_t) irq_handler, IRQF_SHARED, "pushbutton_irq_handler", (void*) (irq_handler));
```

본 lab에서는 flag를 IRQF_SHARED로 지정하여 Interrupt 번호가 여러 interrupt handler에 의해 공유 가능하도록 하였다.

2) free_irq

```
#include <linux/interrupt.h>

void free_irq(unsigned int irq, void* dev_id);
```

Irq	제거하고자 하는 interrupt service 함수의 interrupt 번호
Dev_id	Request_irq에서 사용한 dev_id와 동일한 값을 지정해야 isr이 제거된다.

작성한 코드는 다음과 같으며 request_irq에서의 dev_id와 동일한 값을 지정함을 확인할 수 있다.

```
free_irq(KEYS_IRQ, (void*) irq_handler);
```

✓ 조원 별 기여 사항 및 느낀점

이름	기여도 (0 - 100%)	기여 사항	느낀점
박관영	33.3%	- lab1 코드 작성 및 보고서 작성	OS 기반의 system에서 제공되는 kernel에 대해 학습할 수 있었고, 그 중 일부인 device driver를 작성해 봄으로써 loadable kernel module에 대한 이해도를 높일 수 있었다. 또한 device driver를 통해 user program에서 간단하게 hw component에 접근할 수 있는 것을 확인할 수 있었다.
오한별	33.3%	- 발표자료 작성 및 발표	Kernel level에서 HW를 제어하기 위한 device driver를 작성하는 방법과, device를 file로 취급하며 user program에서 device를 접근하는 방법에 대해 알아볼 수 있었다. 또한 driver 내부에 Interrupt handler를 정의하여 polled I/O 방식 외에 어떻게 device를 제어할 수 있는지 학습할 수 있었다.
김서영	33.3%	- lab2 코드 및 보고서 작성	↳ User mode와 Kernel mode의 차이점을 바탕으

임베디드시스템 설계 및 실습 - 보고서

			로 interrupt I/O방식을 LKM으로 구현해 봄으로써 리눅스 시스템의 kernel에 대하여 학습할 수 있었다. Device driver와 Interrupt 방식으로 HW를 제어하는 방법을 강구할 수 있었고, application에서 kernel과 어떻게 interaction 할 수 있는지 확인할 수 있었다.
--	--	--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------