Embedded System Software 과제 3

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20211584 장준영

개발기간: 2024. 5. 23. -2024. 5. 30.

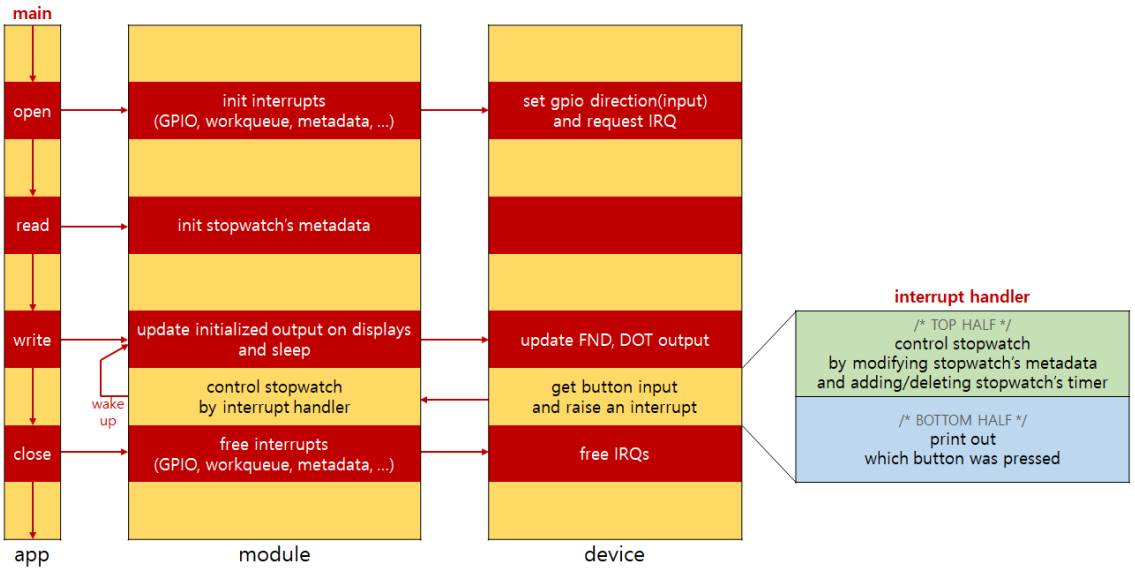
최 종 보 고 서

I. 개발 목표

이번 프로젝트에서는 FPGA 디바이스와 함께 운용되는 스탑워치 디바이스 드라이버를 구현하고, 해당 디바이스 드라이버를 테스트할 수 있는 유저 어플리케이션을 개발한다. 저번 프로젝트인 타이머 디바이스와 매우 유사하지만, 아주 중요한 차이점인 'GPIO를 통한 IRQ 핸들링'을 구현해야 한다. 기존에 구현해두었던 모듈 및 디바이스 코드에, 버튼 입력으로 인터럽트를 발생시키고 해당 인터럽트를 알맞게 처리하는 루틴을 추가해야 한다. 핵심적으로 버튼 GPIO-IRQ 매핑, 인터럽트 핸들링, top half/bottom half 등의 개념이 이용된다.

II. 개발 범위 및 내용

가. 개발 범위



이번 프로젝트의 구조도는 위와 같다. app에서 디바이스 파일을 open 하면 모듈에서 버

튼 인터럽트를 받기 위한 준비를 한다. 이 과정에서 GPIO 방향 설정, IRQ 요청, 인터럽트 핸들러를 위한 workqueue 할당, 변수 초기화 등이 일어난다. 이후 read 하면 스탑워치의 메타데이터(소요 시간, 플래그)를 초기화한다. write는 스탑워치를 실질적으로 실행하는 부분으로, 초기화된 메타데이터에 따라 FPGA 디스플레이에 출력하고 버튼 인터럽트를 대기한다. 여기서부턴 버튼 인터럽트에 따라 스탑워치를 사용할 수 있게 된다. write 연산은 스탑워치 종료 명령이 올 때까지 sleep 한다. 스탑워치가 종료되고 write 연산도 종료되면 마지막으로 close 하여 인터럽트를 위해 요청해둔 IRQ를 free 하고 workqueue를 없앤다. 인터럽트 핸들러는 이번 과제에서 중요하게 여기는 top half 루틴과 bottom half 루틴으로 나누어져 있다. Top half에는 실시간성이 중요하고 빠르게 끝낼 수 있는 연산이, bottom half에는 실시간성이 중요하지 않고 시간이 오래 걸리는 연산이 존재한다. Bottom half를 schedule 하기 위한 자료구조로 workqueue를 사용할 예정이다.

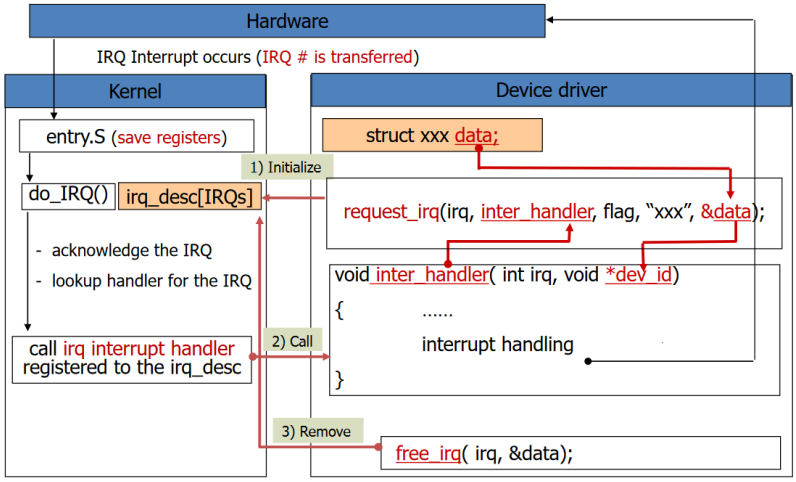
나. 개발 내용

[1] GPIO/IRQ

GPIO(General Purpose Input/Output)은 컴퓨터 하드웨어에서 단순한 디지털 신호로 입출력을 할 수 있는 인터페이스다. GPIO 방향을 입력 또는 출력으로 설정할 수 있고, 전압을 통해 high/low 상태를 전전할 수 있다. 리눅스 커널에서는 GPIO의 방향을 설정하고 해당 GPIO 포트를 IRQ 번호에 매핑할 수 있는 API를 제공한다.

IRQ(Interrupt Request)는 하드웨어 장치가 프로세서에게 특정 이벤트가 발생했음을 알리는 방법으로, 인터럽트를 발생시킨 장치에 각각 고유한 번호를 부여하여 프로세스가 어떤 장치에서 인터럽트가 발생했는지 알 수 있게 해준다. IRQ 번호 당 인터럽트 핸들러가 존재하기 때문에, 발생한 인터럽트를 구별하여 상황에 맞게 처리할 수 있다. 이번 프로젝트에서는 네 가지의 버튼 입력 GPIO에 매핑된 IRQ 번호에 맞게 인터럽트 핸들러를 작성하고, 알맞은 스탑워치 동작을 수행하도록 하여 하드웨어 입력을 처리할 것이다.

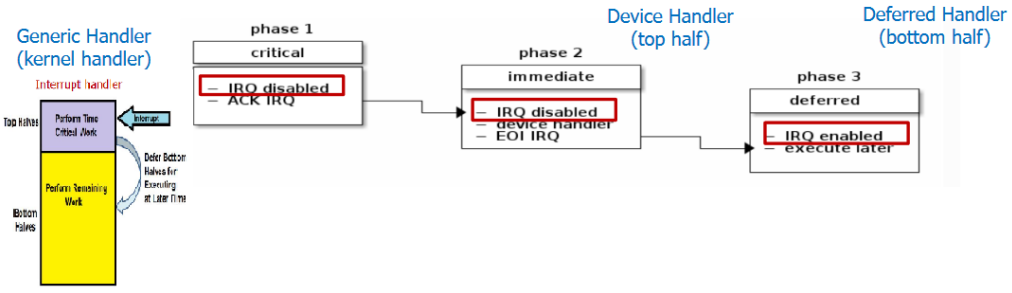
[2] 인터럽트 핸들링



ref: 강의 자료 Chapter 5 Interrupt Handling and Blocking I/O

강의 자료에 설명되어있는 인터럽트 핸들링의 과정이다. 가장 먼저, request_irq 명령을 통해 IRQ 번호에 인터럽트 핸들러, 발생 플래그, 장치 이름, 전송할 데이터(보통 같은 IRQ 번호를 가진 장치끼리의 minor ID로 사용) 등을 할당한다. 등록이 완료되면, 이후 인터럽트가 발생했을 때 irq_desc를 탐색하여 IRQ 번호에 맞는 핸들러를 호출하여 인터럽트를 핸들링한다. IRQ를 제거하고 싶을땐 free_irq 명령을 사용하여 테이블에서 제거한다. 이번 프로젝트에서도 gpio_to_irq로 얻은 IRQ 번호에 각각 알맞은 핸들러를 작성해 request_irq 명령으로 등록할 예정이다.

[3] Top half/Bottom half



ref: 강의 자료 Chapter 5 Interrupt Handling and Blocking I/O

인터럽트는 기본적으로 queuing 되지 않는다. 하나의 인터럽트 핸들러가 수행중일 땐 다른 인터럽트를 받을 수 없고, 이 경우 무시된 인터럽트는 아무런 동작도 하지 못한다는 의미이다. 따라서, 인터럽트 핸들러의 수행 시간은 반드시 짧아야 한다. 하지만 불가피하게 인

터럽트 핸들러 내에서 수행 시간이 긴 연산을 해야한다면, time critical한 작업은 핸들러 내에서 수행되도록 top half에 작성하고, 나머지 수행 시간이 길고 time critical하지 않은 작업은 인터럽트를 막지 않은 채 다른 flow에서 수행할 수 있도록 bottom half에 작성해준다. bottom half 작업을 schedule 하기 위한 자료 구조로는 softirq, tasklet, workqueue가 존재한다. Softirq는 커널 차원의 코드로 기존 인터럽트 핸들러 수준의 정교하고 예민한 시간적 이슈가 있을 때 사용하고, tasklet 역시 softirq의 일부분으로서 interrupt context에서 작동하기 때문에 이번 프로젝트에서는 workqueue를 사용하였다. 우선 workqueue가 구현이 가장 쉽고, 설계 상 나이브하게 예약해도 문제가 없는 작업만을 bottom half에서 수행할 예정이기 때문이다.

III. 추진 일정 및 개발 방법

가. 추진 일정

2024. 5. 23.	저번 프로젝트의 코드에서 app.c와 device.c를 수정하였다. app.c를 수정하면서 각각의 file operation을 어떻게 이용할지 계획하였다. device.c에서는 이번 프로젝트에서 사용하는 FND와 DOT matrix의 동작부를 수정하였는데, 스탑워치의 elapsed time만을 인자로 받고 내부에서 포매팅하도록 구현하였다.
2024. 5. 24. ~ 2024. 5. 25.	디바이스 드라이버 모듈의 코드를 module.c에 작성하였다. 기존의 타이머 디바이스 드라이버의 코드에서 달라진 점은, GPIO를 통해 하드웨어 인터럽트를 관리해야 한다는 것이다. GPIO/IRQ, 인터럽트 핸들링, workqueue 등을 유념하여 구현하였다.
2024. 5. 29. ~ 2024. 5. 30.	**V.** 에서 후술할 문제로 인해 스탑워치의 동작 방식을 모두 바꾸었다. 실제 동작 중에는 문제가 전혀 없었지만, 주석을 추가하기 위해 코드를 리뷰하는 중에 잠재적인 동기화 문제의 가능성을 찾아 이를 개선하였다. 이후 완성된 코드를 컴파일하여 최종적으로 테스트하고 주석을 추가하였다.

나. 개발 방법

[1] app

```c
int main()
{
    /* Open the stopwatch device file. */
    int stopwatch_fd = open(DEV_FILE_LOC, O_RDWR);
    if (stopwatch_fd == -1)
    {
        printf("ERROR(app.c) : stopwatch device file open failed\n");
        return -1;
    }
    /* Initialize the stopwatch. */
    read(stopwatch_fd, NULL, 0);
    /* Start the stopwatch. */
    write(stopwatch_fd, NULL, 0);
    /* Close the stopwatch device file. */
    close(stopwatch_fd);
    return 0;
}
```

응용 프로그램의 코드는 정말 간단하다. 앞서 설계한 대로, 네 개의 file operation이 각각 배정된 역할을 수행하기 위해 호출되고 있다. open으로 디바이스 파일을 열어 필요한 초기화를 하고, read로 스탑워치의 메타데이터를 초기화하고, write로 스탑워치를 실질적으로 시작하고, close로 메모리 회수 등의 마무리 작업을 수행한다. 모듈 코드를 확인하면 알 수 있지만, 스탑워치가 종료될 때까지 write 연산에서 응용 프로그램이 sleep 하고 있다.

[2] device

(※ 과제 2에서 주로 다루었던 디바이스 주소 매핑 및 입출력은 이번 과제의 문맥과 다르다고 생각하여 자세히 설명하지 않고, 지난 과제와 코드 면에서 달라진 부분만 설명하였습니다.)

```c
void fnd_write(const int elapsed)
{
    int min, sec;
    if (elapsed >= TIME_LIMIT)
    {
        min = 0;
        sec = 0;
    }
    else
    {
        min = elapsed / 600;
        sec = (elapsed / 10) % 60;
    }
    /* Concatenate min and sec to unsigned short value. */
    unsigned short int _s_value = ((min / 10) << 12) + ((min % 10) << 8) +
                                  ((sec / 10) << 4) + (sec % 10);
    /* Write value to FND. */
    outw(_s_value, (unsigned int)dev_addr[FND]);
}
```

이번 프로젝트에서는 FND와 DOT matrix만 사용한다. 또, 출력되는 형태가 스탑워치의 소

요 시간(elapsed)과만 관련이 있기 때문에, 편의성을 위해 elapsed만을 인자로 받아 내부에서 값을 변환하였다. 먼저 MMSS 형태로 네 자리 숫자를 출력해야 하는 FND에서는 100ms를 주기로 증가하는 elapsed에서 min과 sec를 각각 계산하여 출력한다. 만약 타이머가 종료되었거나 한계 시간을 초과한 경우 0000을 출력한다.

```
void dot_write(const int elapsed)
{
    int i;
    int hms;
    if (elapsed ⩾ TIME_LIMIT)
        hms = DOT_BLANK;
    else
        hms = elapsed % 10;
    /* Write pattern to DOT matrix. */
    for (i = 0; i < 10; i++)
        outw(dot_number[hms][i] & 0x7F, (unsigned int)dev_addr[DOT] + i * 2);
}
```

100ms 단위의 값을 출력해야 하는 DOT matrix에서는 동일한 과정으로 elapsed에서 단위 시간을 추출한다. 만약 타이머가 종료되었거나 한계 시간을 초과한 경우 비어있는 값을 출력하는 인덱스를 지정한다.

[3] module

```
/* TIMER DEVICE DEFINITIONS */
#define MAJOR_NUM 242
#define DEV_NAME "stopwatch"
#define DEV_FILE_LOC "/dev/stopwatch"

/* TIMER STRUCTURE */
typedef struct _STOPWATCH
{
    struct timer_list timer;
    int elapsed;
    int reset, paused;
} STOPWATCH;

/* STOPWATCH TIME LIMIT */
#define TIME_LIMIT (100 * 60 * 10)
```

가장 먼저 헤더 파일에 필요한 정의를 한다. 메이저 넘버, 디바이스 이름, 디바이스 파일 위치, 타이머와 메타데이터 필드를 포함하고 있는 스탑워치 구조체, 스탑워치의 한계 시간 등이 정의된다. 스탑워치 구조체에서 elapsed는 타이머가 진행 중이라면 100ms에 1씩 증가하고, reset과 paused는 각각 초기화, 일시 정지 플래그이다.

```c
/* STOPWATCH */
static STOPWATCH stopwatch;
/* timer to check if the stop button is pressed for 3 seconds */
static struct timer_list stop_timer;
/* 'already_open' flag for I/O(open) blocking */
enum
{
    NOT_USED = 0,
    EXCLUSIVE_OPEN = 1
};
static atomic_t already_open = ATOMIC_INIT(NOT_USED);
/* mutex semaphore to prevent user process from exiting before the stopwatch ends */
struct semaphore STOPWATCH_QUIT;
```

module.c의 소스코드에 선언된 전역 변수이다. 코드 전체에서 관리할 스탑워치인 stopwatch, vol- 버튼이 3초 이상 눌렸는지 확인하기 위한 stop_timer, 중복 open을 막기 위한 already_open 어토믹 변수, 스탑워치가 종료될 때까지 write 연산이 끝나지 않게 하기 위한 STOPWATCH_QUIT semaphore이다.

```c
/* button informations for gpio */
static unsigned int btn_gpio[BTN_NUM] = {
    IMX_GPIO_NR(1, 11),
    IMX_GPIO_NR(1, 12),
    IMX_GPIO_NR(2, 15),
    IMX_GPIO_NR(5, 14),
};
static irqreturn_t (*btn_handler[BTN_NUM])(int, void *) = {
    btn_home_handler,
    btn_back_handler,
    btn_vol_up_handler,
    btn_vol_down_handler,
};
static const int btn_flag[BTN_NUM] = {
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
};
static const char *btn_name[BTN_NUM] = {
    "BTN#HOME",
    "BTN#BACK",
    "BTN#VOL+",
    "BTN#VOL-",
};
/* work queue for bottom half */
static struct workqueue_struct *wqueue = NULL;
typedef struct _BTN_WORK
{
    struct work_struct work;
    int type;
} BTN_WORK;
```

버튼 인터럽트를 등록 및 관리하는데 필요한 전역 변수이다. 각 버튼의 GPIO 번호(버튼의 포트 및 인덱스로 계산된 값. 실습 시간에 제공된 코드를 참고.)를 나타내는 btn_gpio, 각 버튼이 발생시키는 인터럽트의 핸들러 주소를 나타내는 btn_handler, 인터럽트가 발생하는 시점을 나타내는 btn_flag, 각 버튼의 이름을 나타내는 btn_name이 있다. 이 네 가지 배열은

request_irq 명령에 사용될 예정이다. 여기서 vol- 버튼만 rising과 falling 시에 모두 인터럽트를 발생시키는데, 이는 vol- 버튼이 눌린 시간을 계산해야 되기 때문이다. 이는 이후 핸들러에 대한 설명에 더 자세히 나와 있다. 그 외에도 bottom half 루틴을 위한 workqueue인 wqueue, workqueue에 전송할 BTN_WORK 타입이 존재한다. BTN_WORK 구조체의 type 필드는 버튼의 종류를 나타낸다.

```c
/* HEADER 1: replaced fop functions for device file */
static int stopwatch_open(struct inode *, struct file *);
static int stopwatch_close(struct inode *, struct file *);
static int stopwatch_read(struct file *, char __user *, size_t, loff_t *);
static int stopwatch_write(struct file *, const char __user *, size_t, loff_t *);
static struct file_operations stopwatch_fops = {
    .owner = THIS_MODULE,
    .open = stopwatch_open,
    .release = stopwatch_close,
    .read = stopwatch_read,
    .write = stopwatch_write,
};

/* HEADER 2: interface/utility functions for managing stopwatch */
static void stopwatch_display();
static void stopwatch_add();
static void stopwatch_handler(unsigned long);

/* HEADER 3: interrupt control functions */
static void intr_init();
static void intr_free();
static void stop_timer_handler(unsigned long);
static void stop_timer_add();
static irqreturn_t btn_home_handler(int, void *);
static irqreturn_t btn_back_handler(int, void *);
static irqreturn_t btn_vol_up_handler(int, void *);
static irqreturn_t btn_vol_down_handler(int, void *);
static void wq_handler(struct work_struct *);
```

module.c에는 세 종류의 함수가 존재한다. (1)file operation을 대체하는 함수, (2)스탑워치를 관리하기 위한 함수, (3)인터럽트 제어를 위한 함수 순서로 설명하겠다.

```c
/* stopwatch_open - Replace 'open' fop on device file. */
static int stopwatch_open(struct inode *inode, struct file *file)
{
    /* Compare already_open flag with NOT_USED and set it to EXCLUSIVE_OPEN if equal;
     * Otherwise, meaning that file is already opened, return error(-EBUSY).
     */
    if (atomic_cmpxchg(&already_open, NOT_USED, EXCLUSIVE_OPEN))
        return -EBUSY;
    /* Increase module usage. */
    try_module_get(THIS_MODULE);
    /* Initialize inturrupts and workqueue. */
    intr_init();
    return 0;
}

/* stopwatch_close - Replace 'close' fop on device file. */
static int stopwatch_close(struct inode *inode, struct file *file)
{
    /* Set already_open flag to NOT_USED */
    atomic_set(&already_open, NOT_USED);
    /* Decrease module usage. */
    module_put(THIS_MODULE);
    /* Free IRQs and destroy workqueue. */
    intr_free();
    return 0;
}
```

먼저 open과 close를 대체하는 stopwatch_open과 stopwatch_close이다. 우선 stopwatch_open에선, 중복 open을 방지하기 위해 already_open 변수를 확인해 이미 열렸다면 에러를 리턴하고, 그렇지 않으면 EXCLUSIVE_OPEN으로 변경한다. 이후 모듈의 usage를 1 증가시키고, 인터럽트를 관리하기 위한 준비를 한다. stopwatch_close에선 정반대의 동작을 한다. already_open을 NOT_USED로 변경하고, 모듈 usage를 1 감소시키고, 인터럽트 관리를 중지한다.

```c
/* stopwatch_read - Replace 'read' fop on device file. */
static int stopwatch_read(struct file *file, char __user *buffer, size_t length, loff_t *offset)
{
    /* Initialize stopwatch fields. */
    stopwatch.elapsed = 0;
    stopwatch.reset = 1;
    stopwatch.paused = 0;
    return 1;
}

/* stopwatch_write - Replace 'write' fop on device file. */
static int stopwatch_write(struct file *file, const char __user *buffer, size_t length, loff_t *offset)
{
    del_timer_sync(&(stopwatch.timer));
    /* Update the displays. */
    stopwatch_display();
    down_interruptible(&STOPWATCH_QUIT); /* blocked until the stopwatch ends */
    return 1;
}
```

read와 write를 대체하는 stopwatch_read와 stopwatch_write이다. stopwatch_read에선 스탑워치의 메타데이터 필드를 초기화한다. 스탑워치가 막 실행되었을 땐, home 버튼을 통해 시작되어야 하므로 elapsed는 0, reset은 1, paused는 0이 된다. stopwatch_write에선 초기화한 메타데이터 값을 바탕으로 FPGA 디스플레이를 업데이트하고, semaphore down을 통해 sleep된다.

```c
/* stopwatch_display - Display stopwatch's elapsed time data on FPGA devices. */
static void stopwatch_display()
{
    fnd_write(stopwatch.elapsed);
    dot_write(stopwatch.elapsed);
}

/* stopwatch_add - Set stopwatch's timer fields and register timer. */
static void stopwatch_add()
{
    stopwatch.timer.expires = get_jiffies_64() + (HZ / 10) /* 0.1 (sec) */;
    stopwatch.timer.data = (unsigned long)&stopwatch;
    stopwatch.timer.function = stopwatch_handler;
    add_timer(&(stopwatch.timer));
}
```

다음으로 스탑워치를 관리하기 위한 함수들이다. stopwatch_display에선 현재 elapsed를 바탕으로 디스플레이(FND, DOT)를 업데이트한다. stopwatch_add에선 스탑워치 타이머의 기

본값을 설정하고 타이머 리스트에 타이머를 추가한다.

```c
/* stopwatch_handler - Timer handler called whenever the stopwatch's timer expires
 *                     (timer.expires == get_jiffies_64()).
 */
static void stopwatch_handler(unsigned long timeout)
{
    stopwatch.elapsed++;
    /* End the stopwatch when the time limit is reached. */
    if (stopwatch.elapsed >= TIME_LIMIT)
    {
        stopwatch.elapsed = TIME_LIMIT;
        stopwatch_display();
        up(&STOPWATCH_QUIT);
        return;
    }
    /* Update the displays and add timer. */
    stopwatch_display();
    stopwatch_add();
}
```

스탑워치 타이머가 만료될 때마다 호출되는 stopwatch_handler이다. 우선 elapsed를 1 증가시키고, 만약 elapsed가 한계 시간을 넘어섰다면 스탑워치를 종료한다. 스탑워치가 종료되면서 디스플레이는 초기화되고 semaphore up을 통해 write 연산도 종료된다. 한계 시간을 넘어서기 전인 경우 바뀐 elapsed를 바탕으로 디스플레이를 업데이트하고 타이머를 다시 추가한다.

```c
static unsigned int intr_cnt = 0; /* count how many times interrupt was called */
static int vol_down_pressed = 0;  /* check if the vol down button is pressed or released */

/* intr_init - Initialize interrupts, workqueue, and timer. */
static void intr_init()
{
    int i;
    for (i = 0; i < BTN_NUM; i++)
    {
        /* Set GPIO direction to input. */
        gpio_direction_input(btn_gpio[i]);
        /* Request IRQ for each button. */
        request_irq(gpio_to_irq(btn_gpio[i]), btn_handler[i], btn_flag[i], btn_name[i], 0);
    }
    /* Create workqueue. */
    wqueue = create_workqueue("STOPWATCH_WQ");
    /* Initialize stop_timer. */
    init_timer(&stop_timer);
    /* Initialize variables. */
    intr_cnt = 0;
    vol_down_pressed = 0;
}
```

마지막으로 인터럽트를 제어하는 함수들이다. 가장 먼저, 인터럽트가 몇 번째 불렸는지 세는 intr_cnt와 vol- 버튼이 눌려있는지 눌려있지 않은지 저장하는 vol_down_pressed 플래그가 있다. intr_init 함수에선 각 버튼에 대해 GPIO 방향을 입력으로 설정하고, IRQ 번호를 요

청한다. 또, workqueue를 생성하고 stop_timer를 초기화한다. 관리하는 메타데이터도 0으로 초기화해준다.

```c
/* intr_free - Free interrupts, workqueue, and timer. */
static void intr_free()
{
    int i;
    for (i = 0; i < BTN_NUM; i++)
        /* Free IRQ for each button. */
        free_irq(gpio_to_irq(btn_gpio[i]), NULL);
    /* Flush and destroy workqueue. */
    flush_workqueue(wqueue);
    destroy_workqueue(wqueue);
    /* Delete stop_timer. */
    del_timer_sync(&stop_timer);
    /* Initialize variables. */
    intr_cnt = 0;
    vol_down_pressed = 0;
}
```

반대로 intr_free에선 버튼에 할당되었던 IRQ를 free 하고, workqueue를 없애고, stop_timer를 삭제한다.

```c
/* stop_timer_add - Add stop_timer with a 3-second expiration. */
static void stop_timer_add()
{
    stop_timer.expires = get_jiffies_64() + (3 * HZ); /* 3 (sec) */
    stop_timer.data = NULL;
    stop_timer.function = stop_timer_handler;
    /* Add the timer. */
    add_timer(&stop_timer);
}

/* stop_timer_handler - Timer handler for stop_timer. */
static void stop_timer_handler(unsigned long timeout)
{
    /* Delete stop_timer. */
    del_timer_sync(&(stopwatch.timer));
    /* Set stopwatch's elapsed time to limit to notice the end of the program
     * and update the displays.
     */
    stopwatch.elapsed = TIME_LIMIT;
    stopwatch_display();
    /* Do semaphore-up-operation to wake up the process. */
    up(&STOPWATCH_QUIT);
}
```

stop_timer_add는 stop_timer가 3초 뒤에 stop_timer_handler를 호출하고 종료될 수 있도록 타이머를 추가한다. stop_timer_handler에선 앞서 stopwatch_handler에서도 본 스탑워치 종료 루틴을 수행한다.

```c
/* btn_home_handler - Interrupt handler for the home button. */
static irqreturn_t btn_home_handler(int irq, void *data)
{
    /* TOP HALF - Handle real-time critical behaviors. */
    if (!stopwatch.paused && stopwatch.reset)
    {
        /* Start or restart the stopwatch from reset state. */
        stopwatch_add();
        /* Set the reset flag to false. */
        stopwatch.reset = 0;
    }
    /* BOTTOM HALF - Use workqueue to handle operations
     * that take too long to be processed in the interrupt handler
     * and for which is not very real-time critical.
     */
    /* Allocate memory for bottom half work. */
    BTN_WORK *home_work = (BTN_WORK *)kmalloc(sizeof(BTN_WORK), GFP_KERNEL);
    if (home_work)
    {
        /* Initialize work. */
        INIT_WORK((struct work_struct *)home_work, wq_handler);
        /* Set the button type. */
        home_work->type = BTN_HOME;
        /* Queue work to workqueue. */
        queue_work(wqueue, (struct work_struct *)home_work);
    }
    return IRQ_HANDLED;
    /* All other interrupt handlers follow the same process. */
}
```

home 버튼에서 발생하는 인터럽트를 처리하는 btn_home_handler이다. home 버튼은 구동된 맨 처음 시점이나 vol+ 버튼으로 초기화 된 시점의 스탑워치를 시작하는 역할이므로, paused 상태가 아니고 reset 상태인 경우에만 스탑워치의 타이머를 추가하고 reset 상태를 해제한다. 이후 bottom half 처리를 위해 BTN_WORK 공간을 마련하고, work 초기화 및 타입 설정 이후 workqueue에 추가한다. 이 bottom half 과정은 이후 설명할 모든 인터럽트 핸들러에 대해 공통이다.

```c
/* btn_back_handler - Interrupt handler for the back button. */
static irqreturn_t btn_back_handler(int irq, void *data)
{
    /* TOP HALF */
    if (!stopwatch.reset)
    {
        if (stopwatch.paused)
            /* Add the timer for progress. */
            stopwatch_add();
        else
            /* Delete the timer for pausing. */
            del_timer_sync(&(stopwatch.timer));
        /* Toggle the paused/progress state. */
        stopwatch.paused = 1 - stopwatch.paused;
    }
    /* BOTTOM HALF */
    BTN_WORK *back_work = (BTN_WORK *)kmalloc(sizeof(BTN_WORK), GFP_KERNEL);
    if (back_work)
    {
        INIT_WORK((struct work_struct *)back_work, wq_handler);
        back_work->type = BTN_BACK;
        queue_work(wqueue, (struct work_struct *)back_work);
    }
    return IRQ_HANDLED;
}
```

back 버튼에서 발생하는 인터럽트를 처리하는 btn_back_handler이다. back 버튼은 paused/progress 상태를 토글링해야 하므로, reset 상태가 아닌 경우에만 동작한다. 만약 현재 paused 상태라면 다시 재개하기 위해 스탑워치 타이머를 추가하고, 그렇지 않은 경우 일시 정지를 위해 스탑워치 타이머를 제거한다. 이후 paused 플래그를 재설정한다.

```c
/* btn_vol_up_handler - Interrupt handler for the volume up button. */
static irqreturn_t btn_vol_up_handler(int irq, void *data)
{
    /* TOP HALF */
    /* Delete the timer for reset. */
    del_timer_sync(&(stopwatch.timer));
    /* Initialize stopwatch fields. */
    stopwatch.elapsed = 0;
    stopwatch.reset = 1;
    stopwatch.paused = 0;
    /* Update the displays. */
    stopwatch_display();
    /* BOTTOM HALF */
    BTN_WORK *vol_up_work = (BTN_WORK *)kmalloc(sizeof(BTN_WORK), GFP_KERNEL);
    if (vol_up_work)
    {
        INIT_WORK((struct work_struct *)vol_up_work, wq_handler);
        vol_up_work->type = BTN_VOL_UP;
        queue_work(wqueue, (struct work_struct *)vol_up_work);
    }
    return IRQ_HANDLED;
}
```

vol+ 버튼에서 발생하는 인터럽트를 처리하는 btn_vol_up_handler이다. reset을 위해 스탑워치 타이머를 제거하고, 메타데이터 역시 초기값으로 설정한 후 디스플레이를 업데이트한다.

```c
/* btn_vol_down_handler - Interrupt handler for the volume down button. */
static irqreturn_t btn_vol_down_handler(int irq, void *data)
{
    /* TOP HALF */
    /* Toggle the pressed/released state. */
    vol_down_pressed = 1 - vol_down_pressed;
    if (vol_down_pressed)
        /* Add stop_timer to time how long the vol down button is pressed. */
        stop_timer_add();
    else
        /* Delete the stop_timer because the button was released before 3 seconds. */
        del_timer_sync(&stop_timer);
    /* BOTTOM HALF */
    BTN_WORK *vol_down_work = (BTN_WORK *)kmalloc(sizeof(BTN_WORK), GFP_KERNEL);
    if (vol_down_work)
    {
        INIT_WORK((struct work_struct *)vol_down_work, wq_handler);
        vol_down_work->type = BTN_VOL_DOWN + vol_down_pressed;
        queue_work(wqueue, (struct work_struct *)vol_down_work);
    }
    return IRQ_HANDLED;
}
```

vol- 버튼에서 발생하는 인터럽트를 처리하는 btn_vol_down_handler이다. 우선 vol- 버튼의 인터럽트는 누를 때와 뗄 때 모두 발생하기 때문에, 신호가 발생할 때마다 vol_down_pressed를 제어해주어야 한다. 만약 현재 버튼이 눌린 상태라면, 3초를 세는

stop_timer를 추가한다. 이 타이머가 추가되고 3초가 지날 때까지 유지된다면 타이머 핸들러에서 스탑워치를 종료할 것이다. 만약 버튼이 떼진 상태라면, stop_timer를 제거한다. 이런 방식으로 버튼이 눌린 시간을 측정할 수 있다.

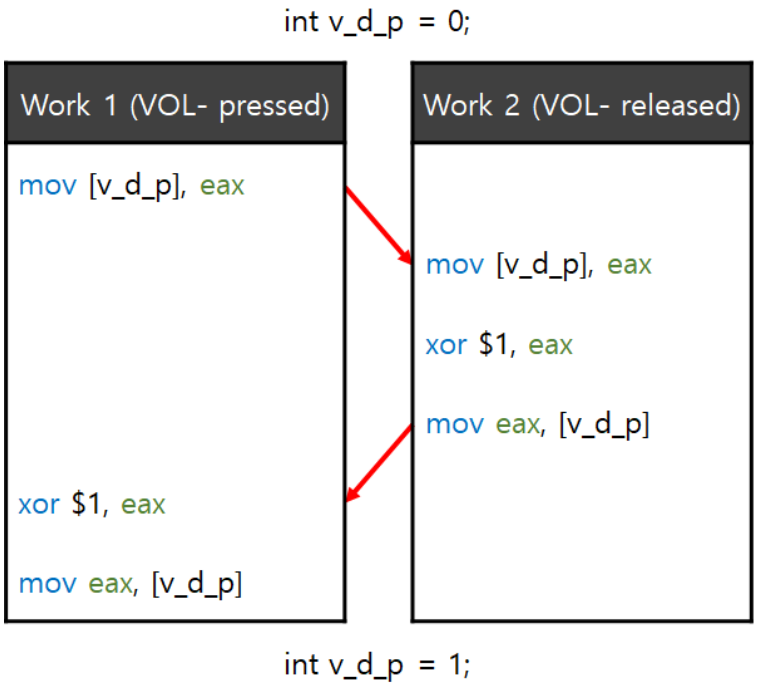
```c
/* wq_handler - Handle the work in the workqueue. */
static void wq_handler(struct work_struct *work)
{
    /* Print interrupt count. */
    printk(KERN_INFO "[INTR %u] ", ++intr_cnt);
    /* Print which button was pressed. */
    switch (((BTN_WORK *)work)->type)
    {
    case BTN_HOME:
        printk(KERN_INFO "HOME pressed\n");
        break;
    case BTN_BACK:
        printk(KERN_INFO "BACK pressed\n");
        break;
    case BTN_VOL_UP:
        printk(KERN_INFO "VOL+ pressed\n");
        break;
    case BTN_VOL_DOWN + 0:
        printk(KERN_INFO "VOL- released\n");
        break;
    case BTN_VOL_DOWN + 1:
        printk(KERN_INFO "VOL- pressed\n");
        break;
    default:
        break;
    }
    /* Free allocated memory for work. */
    kfree((BTN_WORK *)work);
}
```

Workqueue의 work를 처리하는 wq_handler이다. 어떤 과정을 bottom half에서 수행할까 생각하다가, 교수님께서 수업 시간에 메시지 로깅 같은 작업은 소요 시간이 길어 인터럽트 핸들러 내에서는 하면 안된다고 말씀하신 것이 떠올랐다. 따라서, 디버깅 과정에 필요한 디버그 메시지를 띄우는 것을 bottom half에서 수행하기로 설계하였다. 먼저 몇 번째 인터럽트인지 출력하고, 이후 어떤 버튼이 눌리거나 떼어진 상태인지 출력한다. 이후, workqueue에 추가하기 위해 할당했던 BTN_WORK의 메모리를 free 한다.

네 개의 인터럽트 핸들러에서 top half/bottom half를 공통적인 기준으로 분할하였다. 스탑워치의 필드 값을 불러오고 변형하여 저장하는 스탑워치의 실질적인 동작 부분이 top half, 디버깅을 위해 필요한 정보를 로깅하는 부분이 bottom half이다. 이 기준으로 핸들러를 분할한 이유는 다음과 같다.

[1] top half

스탑워치의 실질적인 동작을 맡는 부분이기 때문에 버튼이 눌리자 마자 수행되어야 하고, 굉장히 짧은 시간 내에 마무리지을 수 있기 때문에 인터럽트 핸들러 내에서 처리해도 문제가 없다. 또, 가장 주요한 문제인 동기화 문제를 차단하는 역할을 한다.



Workqueue의 work는 process context에서 수행된다. 즉, 그 수행 순서를 알 수 없고 동작 중에도 context switching이 일어날 수 있다. 위 상황은 vol- 버튼을 한 번 눌렀다 떼는 동작을 하여 vol_down_pressed(v_d_p) 값이 0->1->0으로 변화하길 기대하는 상황이다. 만약 work 1에서 eax 레지스터에 v_d_p 값을 load 하자마자 work 2로 context switch 되고, work 2에선 값 반전을 마치고 다시 v_d_p에 store 한다면, work 1에선 반전된 값을 확인할 수 없어 두 work 모두 v_d_p가 1일 때의 과정을 수행하게 된다. 물론 어셈블리 레벨의 instruction을 단순화하여 표시하였기 때문에 컴파일러에 따라 위 경우와 완전히 동일한 연산을 하지 않을지도 모르지만, 공통적으로 사용하는 변수의 일관성을 유지하기 위해 이러한 과정은 서로 배타적이어야 한다.

[2] bottom half

인터럽트 핸들러는 가능한 한 빠르게 실행되어야 하는데, printk와 같은 로깅 함수는 상대적으로 많은 시간을 소모한다. 따라서 이 과정에서 무시되는 인터럽트가 발생할 수 있다. 또, printk 함수는 reentrant 하지 않다. 즉, 한 번에 하나의 컨텍스트에서만 printk를 호출할 수 있다. 이로 인해 다른 루틴에서 printk가 동작하고 있을 때, 인터럽트 핸들러가 호출되어 printk를 또 호출한다면 문제가 발생할 수 있다.

```c
/* stopwatch_init - insmod */
static int __init stopwatch_init()
{
    /* Register device driver module. */
    int res = register_chrdev(MAJOR_NUM, DEV_NAME, &stopwatch_fops);
    if (res < 0)
    {
        printk("ERROR(timer.c) : register_chrdev failed\n");
        return -1;
    }
    /* Initialize stopwatch_END semaphore to 0. */
    sema_init(&STOPWATCH_QUIT, 0);
    /* Initialize timer. */
    init_timer(&(stopwatch.timer));
    /* Map all FPGA devies. */
    map_device();

    return 0;
}

/* stopwatch_exit - rmmod */
static void __exit stopwatch_exit()
{
    /* Unmap all FPGA devies. */
    unmap_device();
    /* Remove timer. */
    del_timer_sync(&(stopwatch.timer));
    /* Unregister device driver module. */
    unregister_chrdev(MAJOR_NUM, DEV_NAME);
}

module_init(stopwatch_init);
module_exit(stopwatch_exit);

/* Module license and author */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Junyeong JANG");
```

마지막으로 module_init, module_exit을 통해 모듈이 추가될 때와 제거될 때의 동작을 지정하고, 라이선스와 저자를 쓰면 모듈 코드가 완성된다. stopwatch_init과 stopwatch_exit에서는 모듈 전반에 필요한 스탑워치 타이머, semaphore, 디바이스 매핑/언매핑 등을 관리한다.

IV. 연구 결과

해당 프로젝트가 요구하는 사항을 모두 만족하도록 구현하였다.

FPGA device, interrupt, stopwatch를 제어하는 하나의 모듈을 구현하였다.
interrupt handler를 top half와 bottom half로 나누어 구현하였다.
보고서에 top half와 bottom half로 구분한 명확한 기준을 제시하였다.
소수점 첫 번째 자리까지의 시간이 유지된다.
VOL- 버튼을 3초 이상 누른 채 유지하여 스탑워치가 종료될 때까지 유저 어플리케이션이 종료되지 않는다.
프로그램 종료 시 출력 디바이스를 초기화한다.
그 외 사소한 요구 사항을 만족하도록 구현하였다.

V. 기타

Github에 커밋한 프로젝트 레포지토리의 버전 1을 확인하면, 문제 가능성이 있는 스탑워치 동작의 구현을 확인할 수 있다.

```
+ /* TIMER STRUCTURE */
+ typedef struct _STOPWATCH
+ {
+     struct timer_list timer;
+     int elapsed;
+     int paused, reset;
+     int stopped;
+ } STOPWATCH;
+
+ /* STOPWATCH */
+ static STOPWATCH stopwatch;
+
+ #define TIME_LIMIT (100 * 60 * 10)
+ #define STOP_NOT_PRESSED (TIME_LIMIT - 31)
```

먼저 초기에는 STOPWATCH 구조체에 paused, reset, stopped를 관리하고 타이머 핸들러에서 이를 각각 체크해 스탑워치가 어떤 상태에 있는지를 확인하고자 했다. back 버튼이 눌려 정지된 상태인지 저장하는 paused, vol+ 버튼이 눌려 초기화 되었거나 막 시작하였는지 저장하는 reset, vol- 버튼이 처음 눌렸을 때의 elapsed를 저장하는 stopped 필드를 통해 추가

적인 타이머 선언 없이 모든 동작을 구현할 수 있을 거라고 생각했다.

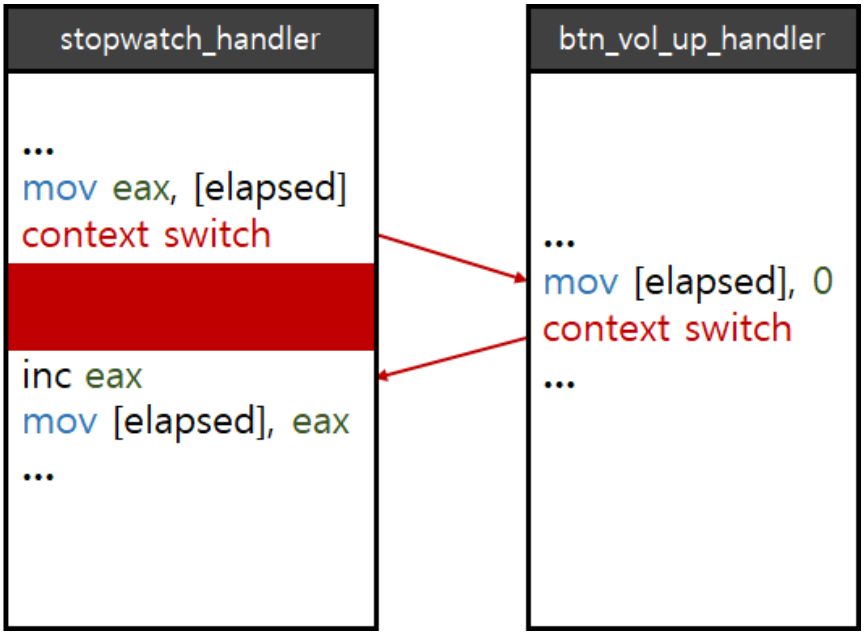
```
+ static void stopwatch_handler(unsigned long timeout)
+ {
+     if (!stopwatch.paused && !stopwatch.reset)
+         stopwatch.elapsed++;
+
+     if (stopwatch.elapsed - stopwatch.stopped >= 30)
+     {
+         stopwatch.elapsed = TIME_LIMIT;
+         stopwatch_display();
+         up(&STOPWATCH_QUIT);
+         return;
+     }
+
+     stopwatch_display();
+     stopwatch_add();
+ }
```

스탑워치의 타이머 핸들러에서는 각각의 메타데이터 필드에 대한 처리를 한다. 멈춰있어야 하는 상황이라면 elapsed를 증가시키지 않고, stopped가 처음 눌린지 3초가 지났다면 중지한다.

```
+ irqreturn_t btn_vol_up_handler(int irq, void *data)
+ {
+     /* TOP HALF */
+     stopwatch.reset = 1;
+     stopwatch.paused = 0;
+     stopwatch.elapsed = 0;
+     /* BOTTOM HALF */
+     BTN_WORK *vol_up_work = (BTN_WORK *)kmalloc(sizeof(BTN_WORK), GFP_KERNEL);
+     if (vol_up_work)
+     {
+         INIT_WORK((struct work_struct *)vol_up_work, wq_handler);
+         vol_up_work->type = BTN_VOL_UP;
+         queue_work(wqueue, (struct work_struct *)vol_up_work);
+     }
+ }
```

인터럽트 핸들러의 일례로, vol+ 버튼이 눌렸을 때 각각의 필드를 초기값으로 설정한다. 이렇게 구현하면 이론적으로 동작의 문제는 없으나, 여러 context flow를 가지고 실행되는 커

널 프로그램의 특성상 동기화 문제가 발생한다.



코드를 읽으며 예상한 문제 상황 중 한 가지로, stopwatch_handler가 호출되어 elapsed가 1 증가하는 동안 인터럽트가 발생하여 btn_vol_up_handler가 호출되는 상황이다. 인터럽트 핸들러에서는 스탑워치를 초기화하기 위해 elapsed를 0으로 설정하지만, eax 레지스터에는 reset이 불리기 전의 elapsed가 저장되어 있어 이 값을 1 증가시키고 다시 elapsed에 쓴다. 결국 reset이 제대로 동작하지 않고, 스탑워치는 정지해있지만 디스플레이에 표시되는 elapsed는 0이 아닌 경우가 발생할 수 있다. 실제 동작 중에 문제가 발생한 적은 없기 때문에 이 추측이 정확한 지는 모르겠지만, 만에 하나 있을 지도 모르는 문제 상황을 없애기 위해 개선하였다.