Embedded System Software 과제 3

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20171664, 이상윤

개발기간: 2024. 05. 23. -2024. 05. 28.

최 종 보 고 서

I. 개발 목표

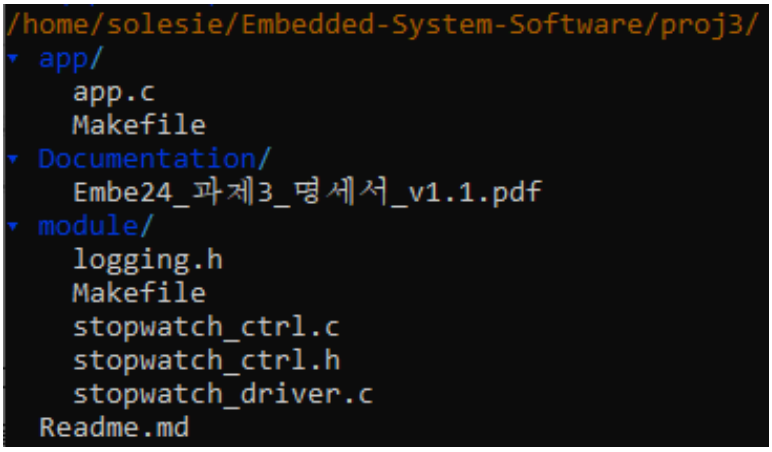
모듈 프로그래밍, 디바이스 드라이버 구현, 인터럽트(Top Half, Bottom Half) 등을 사용하여 간단한 스톱워치 디바이스 드라이버를 구현하고, 그것을 테스트하는 유저 응용 프로그램을 구현한다.

II. 개발 범위 및 내용

가. 개발 범위

1. dot, fnd 장치 출력을 구현한다.
2. Top Half는 프로그램 실행과정 중 대부분 캐쉬에 존재할 stopwatch 관련 변수를 수정하고, interrupt가 의도치 않게 작동하는 부분만 담당하도록 한다.
3. Bottom Half는 dot, fnd 장치 출력, 타이머 설정(세팅, 삭제, 등록), 대기중인 커널 프로세스 깨우기 와 같이 다소 무거운 작업을 담당하도록 한다.
4. Top Half와 Bottom Half가 상호작용함에 있어 명세서에 어긋나지 않도록 한다(race condition 등).
5. Module interface에 맞게 stopwatch device driver을 구현한다.
6. 해당 stopwatch device driver을 사용하는 유저 응용 프로그램을 구현한다.

나. 개발 내용



<그림1: 프로젝트 개발 디렉토리 구조>

1. module/stopwatch_driver.c

   insmod, rmmod 될때의 동작을 정의하고, open, close, read, ioctl과 같은 file operations들을 정의한다. ioctl command가 정의되지 않거나 device file을 생성하는 과정에 오류가 발생하는 등 직접적으로 정의되지 않은 모듈 컨트롤 상황에서 예외 컨트롤을 담당하도록 한다.

2. module/stopwatch_ctrl.c

   1) 명세서 분석

      해당 파일은 stopwatch logic의 핵심적인 부분을 담당하는 소스코드이다. 위 소스코드에서 구현할 개발 내용을 정리하기에 앞서, 명세서를 우선 분석한다.

      fnd는 종료/초기상태에서 0000을 유지해야한다. stopwatch가 1초마다 진행됨에 따라 앞 두자리는 분(최대 59분), 뒤 두자리는 초(최대 59초)를 표현한다. dot은 종료상태에 off, 초기상태에서 0을 유지해야한다. stopwatch가 0.1초마다 진행됨에 따라 0~9를 표현한다.

      home 버튼은 스톱워치를 시작, back 버튼은 일시정지/재시작, volup 버튼은 초기화, voldown 버튼은 3초 이상 누를 시 바로 종료를 담당한다. 위 버튼 입력 행동과 timer는 모두 interruupt로 구현한다. 또한 모든 interrupt는 Top Half, Bottom Half로 나눠져 핸들링하도록 한다.

      디바이스 드라이버의 이름은 /dev/stopwatch, major number은 242로 설정하고 ioctl명령을 통해 stopwatch가 실행되는 동안 프로세스는 sleep 상태로 대기한다.

   2) 소스코드 개발 내용

      위 명세서의 내용을 모두 담당하도록 한다. TH(Top Half)와 BH(Bottom Half), stopwatch 로직, timer 로직, 그리고 fpga 출력 로직들 모두 이 소스코드에 구현하도록 한다. 해당 로직들은 모두 interrupt를 중심으로 철저히 엮여 있기 때문이다.

3. app/app.c

   insmod가 되지 않은 상황이라면 execl() 함수를 통해 insmod를 따로 수행한다. 그리고 stopwatch device driver을 open한 다음 ioctl명령어를 통해 stopwatch를 작동시킨다. 이 때, 프로세스는 stopwatch가 사용자의 voldown 행동을 통해 종료되기 전까지 sleep상태로 유지된다.

III. 추진 일정 및 개발 방법

가. 추진 일정

- 2024/05/23: 프로젝트 분석 및 구조 구성
- 2024/05/24: user application 구현, TH handling
- 2024/05/25: BH handling
- 2024/05/26: timer interrupt TH/BH handling
- 2024/05/27: stopwatch 자원 컨트롤, stopwatch 로직 구현
- 2024/05/28: stopwatch device driver 구현, 로직 검증 및 테스트

나. 개발 방법

1. Top Half

```
104
105 /*
106  * It has been observed that interrupt handlers,
107  * despite being able to directly read from hardware registers whether the signal is FALLING or RISING,
108  * often retrieve incorrect values due to time delays.
109  * Therefore, if the interval between interrupts is too short, it should be ignored.
110  * This logic is included in top half interrupt handler.
111  */
```

<그림2: stopwatch_ctrl.c 104~111>

target board의 button click interrupt를 조사하는 과정에서 그림2에 언급된 바와 같이 interrupt가 직관적으로 오지 않음을 관찰하였다. 여기서 직관적으로 오지 않는다는 뜻은 falling edge, rising edge를 완벽하게 구분할 수 없다는 점뿐만 아니라 사용자가 의도하지 않은 interrupt또한 종종 발생함을 의미한다. 즉, 대부분의 상황에서 사용자가 의도한 인터럽트와 더불어 다른 인터럽트가 추가로 발생한다.

우선 사용자의 의도와 맞게 인터럽트를 핸들링하는 알고리즘이 필요하다. 잦게 잘못 발생하는 인터럽트 상황의 패턴을 분석한 결과 간단히 두가지 상황으로 추릴 수 있었다. 우선 falling edge 후 rising edge 없이 연속하여 falling edge가 발생하는 상황이다(연속된 rising edge도 마찬가지). 두번째로 의도하지 않은 falling edge와 rising edge가 매우 빠르게 오는 상황이다.

```
122     // FALLING: v = 0, RISING: v = 1
123     if(v == 0){
124         // FALLING -> FALLING ignored
125         if(home_prev_released < home_prev_pressed) return IRQ_HANDLED;
126         // Too fast RISING -> FALLING ignored
127         if(cur - home_prev_released <= DEBOUNCING) return IRQ_HANDLED;
128         home_prev_pressed = cur;
129     }
130     else{
131         // RISING -> RISING ignored
132         if(home_prev_pressed < home_prev_released) return IRQ_HANDLED;
133         home_prev_released = cur;
134         return IRQ_HANDLED;
135     }
```

<그림3: stopwatch_ctrl.c 122~135>

우선 모든 버튼은 누름과 동시에(falling edge가 옴과 동시에) 즉시 작동하도록 한다. <그림3>과 같이 각 버튼에 대해 이전에 falling edge가 발생한 시각, 이전에 rising edge가 발생한 시각을 기록해둔다(jiffies). 이를 통해 연속하여 같은 신호가 오는 상황을 해결한다. 또한 현실 세상에서 거의 불가능한 속도(DEBOUNCING)로 눌렀다 떼었다고 나타내는 상황은 인터럽트를 무시함을 통해 해결한다.

TH에선 위처럼 적절한 인터럽트인지 판별하는 로직이 들어간다. 적절하지 않은 인터럽트가 발생하는 경우는 하나 더 있는데 이는 다음 문단의 BH부분에서 설명한다. 그리고 TH는 프로그램 실행과정 중 대부분 캐쉬에 존재할 stopwatch 관련 변수를 수정한 후 BH로 stopwatch의 데이터를 복사하여 전달한다.

```
17 struct stopwatch{
18     unsigned int cur_time; // 0<=time<TIME_MAX
19     int is_started; // boolean
20     int is_paused; // boolean
21     int is_over_timer_running; // boolean
22     int is_over; // boolean
23 };
```

<그림4: stopwatch_ctrl.c 17~23>

위 stopwatch 자료구조의 내용을 수정한다. timer interrupt TH의 경우엔 cur_time을 1씩 증가시킬 것이다. start interrupt TH의 경우 is_start를 1(true)로 바꿀 것이다. back interrupt TH의 경우 is_paused를 toggle할 것이다. volup interrupt TH의 경우 cur_time을 초기화하고 is_start를 false로 바꿀 것이다. voldown interrupt TH의 경우 falling edge에서 is_over_timer_running을 true로 바꾸고 rising edge에서 false로 바꿀 것이다.

```c
struct single_thread_wq_elem{
    struct work_struct work;
    struct stopwatch data;
};
```

```c
    sw.cur_time = 0;
    sw.is_started = 0;
    sw.is_paused = 0;

    w = kmalloc(sizeof(struct single_thread_wq_elem), GFP_ATOMIC);
    if(!w){
        LOG(LOG_LEVEL_INFO, "Heap lack");
        return IRQ_HANDLED;
    }
    memcpy(&w->data, &sw, sizeof(struct stopwatch));
    INIT_WORK(&w->work, volup_do_wq);
    queue_work(single_thread_wq, &w->work);
    return IRQ_HANDLED;
}
```

<그림5, 6: work and stopwatch_ctrl.c volup interrrupt handler TH>

이렇게 전역적인 값(sw)을 설정하고 그 값을 정의한 single_thread_wq_elem 자료구조에 복사해서(memcpy) workqueue를 사용해 BH로 나머지 작업을 넘긴다.

2. Bottom Half

BH에서는 dot, fnd 장치 출력, 타이머 설정(세팅, 삭제, 등록), 대기중인 커널 프로세스 깨우기와 같은 다소 무거운 작업들을 담당한다. BH로 넘어간 작업들은 넘어간 순서대로 작동해야한다. 예시로 fnd가 0150(1분 50초)인 상황에서 갑자기 이전에 BH로 넘어온 작업이 남아있어 fnd가 0149(1분 49초)인 상황이 되선안된다. softirq의 작업은 한 작업이 여러 cpu에서 동시에 돌아갈 가능성이 있다. tasklet의 작업은 다른 작업의 경우 여러 cpu에서 동시에 돌아갈 가능성이 있다. workqueue의 작업은 여러 cpu에서 동시에 돌아갈 가능성이 있다.

```c
/*
 * Called once by module_init.
 * Return 0 on Error.
 */
int stopwatch_init(void){
    // init MMIO
    iom_fpga_dot_addr = ioremap(IOM_FPGA_DOT_ADDRESS, 0x10);
    iom_fpga_fnd_addr = ioremap(IOM_FND_ADDRESS, 0x4);
    // init timer
    init_timer(&fpga_timer);
    init_timer(&over_timer);
    // init wq
    single_thread_wq = create_singlethread_workqueue("single_thread_wq");
    if(!single_thread_wq){
        LOG(LOG_LEVEL_INFO, "Heap lack");
        return 0;
    }

    init_completion(&over);
    return 1;
}
```

<그림7: stopwatch_ctrl.c stopwatch 초기화 부분>

이에 <그림7>에서 보이는 바와 같이 single thread workqueue를 사용한다. 이는 하나의 스레드만을 사용하여 작업을 처리하고 순차적으로 실행되므로 동기적으로 실행된다고 볼 수 있다. 또한 TH에서 언급하였듯 stopwatch의 상태를 나타내는 전역적인 값이 따로 정의한 struct work 자료구조에 복사되므로 다른 인터럽트가 TH에서 값을 미리 바꾼 경우는 고려하지 않아도 된다.

```
332 /*
333  * Work of volup_inter_handler().
334  * This function deletes fpga_timer and reset fpga.
335  */
336 static void volup_do_wq(struct work_struct* work){
337     struct single_thread_wq_elem* w = container_of(work, struct single_thread_wq_elem, work);
338     del_timer_sync(&fpga_timer);
339     fpga_print_fnd(w->data.cur_time);
340     fpga_print_dot(ZERO);
341     kfree(w);
342 }
```

<그림8: stopwatch_ctrl.c volup BH>

동시성 문제가 BH에서 interrupt disable을 하지 않고도 어느정도 해결되었지만 timer interrupt와 관련된 문제가 남아있다. 예시로 <그림8>과 같이 BH에서 기존에 등록된 타이머를 삭제하는 경우, volup TH가 실행되고 해당 volup BH가 스케줄 되기 전에 timer interrupt가 발생해버릴 수도 있다.

이를 해결하기 위해 BH는 불변하는 구조로 설계한다. 즉, workqueue에 이미 들어간 작업은 모두 수행됨을 보장한다. 들어가지 말아야할 작업은 애초부터 TH에서 BH로 작업을 넘기지 않는다. <그림4>의 is_over_timer_running, is_over과 같은 값들은 이를 위해 존재한다.

```
254 /*
255  * It is a callback function that is called every 0.1 seconds and modifies the FPGA.
256  */
257 static void fpga_timer_callback(unsigned long unused){
258     struct single_thread_wq_elem* w;
259 
260     // Managing timers in the bottom half can lead to a situation
261     // where a timer that should not be executed might run
262     // if the bottom half has not yet been scheduled.
263     // Consider this race condition.
264     if(sw.is_over) return;
265     if(sw.is_paused) return;
266     if(!sw.is_started) return;
267 
268     sw.cur_time = (sw.cur_time + 1) % TIME_MAX;
```

```
281 /*
282  * It is a callback function that conducts a termination countdown.
283  */
284 static void over_timer_callback(unsigned long unused){
285     struct single_thread_wq_elem* w;
286 
287     // Consider the race condition.
288     if(!sw.is_over_timer_running) return;
289 
290     sw.is_over = 1;
```

<그림 9,10: stopwatch_ctrl.c timer callback functions>

<그림 9,10>은 timer interrupt TH로서 그 예시를 나타낸다. fpga의 스톱워치 값을 증가시키는 timer callback은 volup을 통해 is_started가 false로 설정된 경우 바로 종료된다. 또한 <그림 10>의 스탑워치 종료 타이머가 이미 작동한 경우 BH로 작업을 넘기지 않고 바로 종료한다. 즉, TH는 일관적으로 유지되는 stopwatch 전역변수의 값을 보고 그 행동을 결정하고, BH는 전달된 값에 따라 행동을 그대로 수행하는것에 불과하다.

또한 <그림 8>의 341번째 줄과 같이 이미 끝난 작업은 kfree를 통해 커널 자원을 확보하도록 한다.

3. stopwatch setting

1) user(kernel) process sleep

이는 completion 자료구조를 통해 sleep상태로 들어간 후 voldown interrupt BH에서 깨워줄 때까지 기다리도록 구현한다.

2) Memory Mapped IO

dot, fnd를 출력하기 위해 ioremap을 통해 적절히 맵핑하고 outw를 통해 값들을 출력하도록 한다.

```
96 static void fpga_print_fnd(unsigned int time){
97     unsigned char fnd[FND_CNT];
98     unsigned short int fnd_short;
99     fnd[0] = (time/600)/10, fnd[1] = (time/600)%10; // minutes
100    fnd[2] = ((time%600)/10)/10, fnd[3] = ((time%600)/10)%10; // seconds
101    fnd_short = fnd[0] << 12 | fnd[1] << 8 | fnd[2] << 4 | fnd[3];
102    outw(fnd_short, (unsigned int)iom_fpga_fnd_addr);
103 }
```

<그림11: stopwatch_ctrl.c fpga_fnd>

<그림11>의 함수 인자의 time은 일반적으로 <그림4> stopwatch struct의 cur_time을 받도록 한다. 59분을 0.1초단위(편의상 “01sec” 으로 정의하자)로 변환시 35400(01sec), 59초를 0.1초단위로 변환 시 590(01sec), 0.9초를 0.1초단위로 변환 시 9(01sec)이다. 이를 다 합하면 time은 최대 35999 까지의 값을 가질 수 있다. <그림11>의 코드는 이 time값을 fnd에 적절히 출력할 수 있도록 분해하는 과정을 나타낸다.

3) General Perpose IO

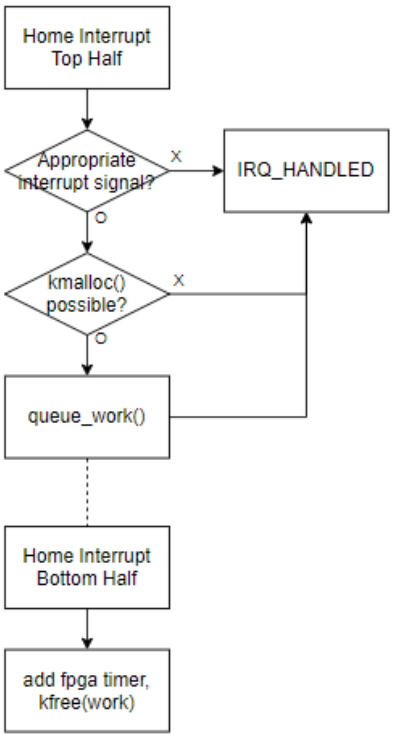
```
423 /*
424  * This function operates in a blocking(synchronous) manner and must be called after stopwatch_init.
425  * Please note that the existing interrupt handlers for home, back, volup, and voldown may be reset.
426  */
427 void stopwatch_run(void){
428     int irq, req;
429 
430     // It is initialized in advance just in case.
431     free_irq(gpio_to_irq(IMX_GPIO_NR(1, 11)), NULL);
432     free_irq(gpio_to_irq(IMX_GPIO_NR(1, 12)), NULL);
433     free_irq(gpio_to_irq(IMX_GPIO_NR(2, 15)), NULL);
434     free_irq(gpio_to_irq(IMX_GPIO_NR(5, 14)), NULL);
435 
436     // init home interrupt
437     gpio_direction_input(IMX_GPIO_NR(1,11));
438     irq = gpio_to_irq(IMX_GPIO_NR(1,11));
439     req = request_irq(irq, home_inter_handler, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "home", 0);
440     if(req) LOG(LOG_LEVEL_INFO, "Home button is not available.");
```

<그림 12: stopwatch_ctrl.c ioctl 실행 시 스톱워치 초기화 부분>

IMX_GPIO_NR(u,v) 매크로를 통해 i.MX 프로세서에서 u번 GPIO 컨트롤러의 v번 핀을 고유한 GPIO 번호로 변환한다. 그 후 gpio_to_irq() 함수를 통해 해당 고유 GPIO 번호에 해당하는 irq번호를 가지고 오게 한다. 그 후 request_irq() 를 통해 핸들러를 등록하는 과정을 거친다.

Ⅳ. 연구 결과

1. Home Interrupt Handling Flow Chart

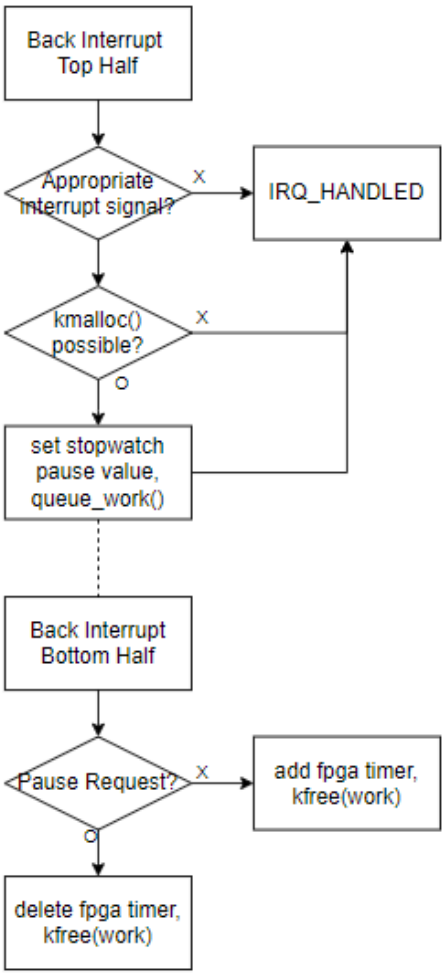


<그림 13: Home Interrupt Handling Flow Chart>

<그림13>은 home interrupt handling의 전체적인 flow chart를 나타낸다. 두번째 단계의 Appropriate interrupt signal을 확인하는 부분은 해당 interrupt가 적절한지 TH에서 판별하는 부분을 나타낸다. 또한 queue_work()단계 이후 점선으로 이어진 부분은 BH로 넘어감을 나타낸다. 이는 이후 flow chart에서도 동일하다.

실질적으로 fpga_timer를 시작시키는 역할만 수행한다.

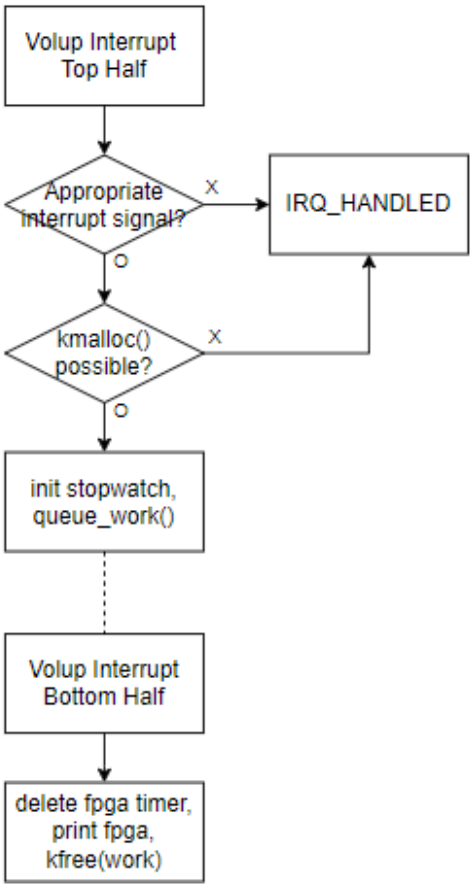
2. Back Interrupt Handling Flow Chart



<그림 14: Back Interrupt Handling Flow Chart>

TH에서 전역 stopwatch 값을 변경한다. BH에서는 복사된 값을 인자로 받아 정지할지, 다시 시작할지 결정한다.

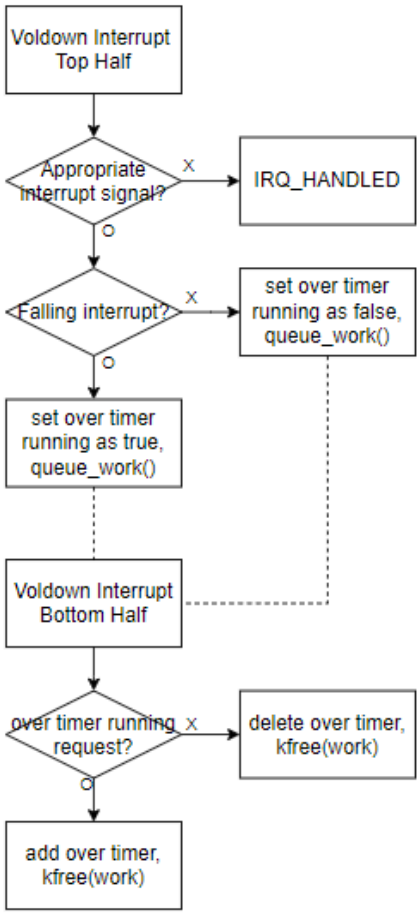
3. Volup Interrupt Handling Flow Chart



<그림 15: Volup Interrupt Handling Flow Chart>

물론 타이머 인터럽트에서 무시되겠지만 BH에서 명시적으로 타이머를 삭제해준다. 그 후 초기값으로 fpga에 출력한다.

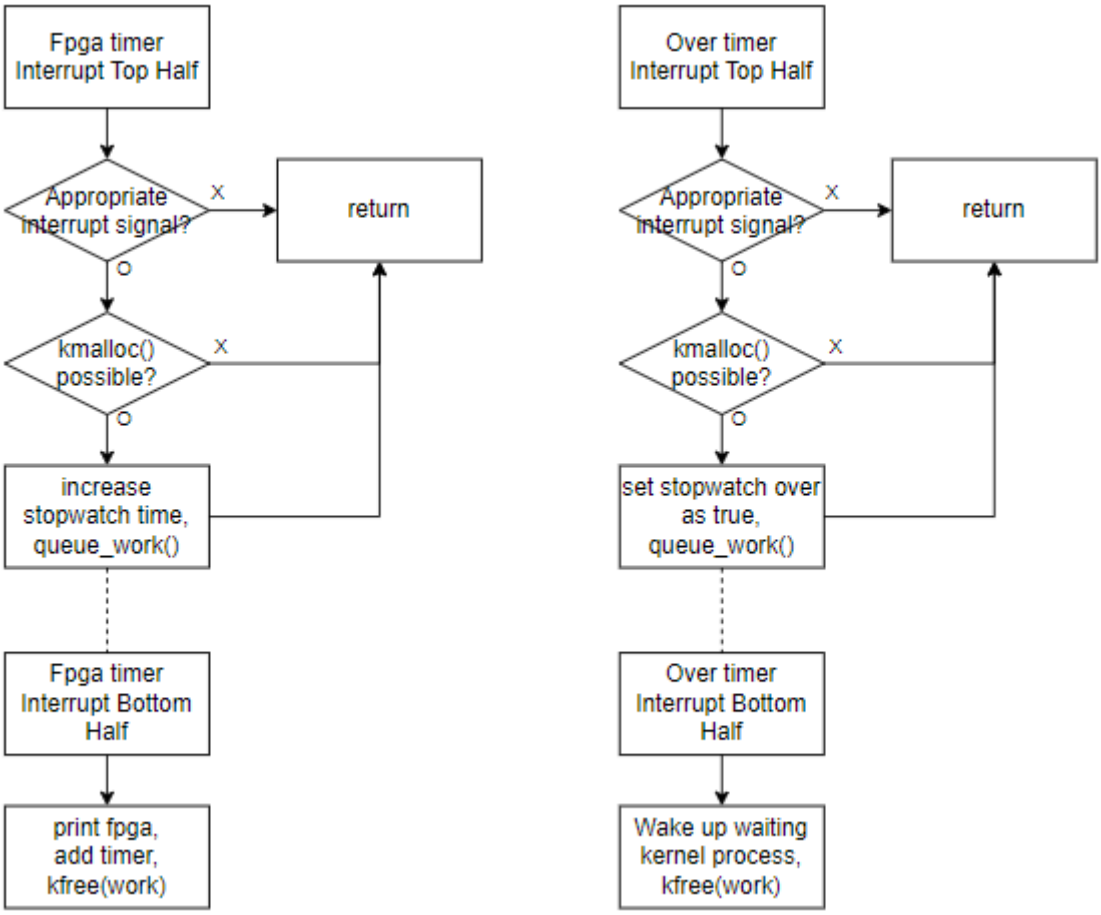
4. Voldown Interrupt Handling Flow Chart



<그림 16: Voldown Interrupt Handling Flow Chart>

이전 버튼 인터럽트에서는 rising edge는 적절한 인터럽트인지 판별할 때 외에 사용되지 않았지만 voldown에선 다르다. rising edge인 경우에는 설정된 타이머가 작동되지 않도록 전역 stopwatch 값을 변경해야한다. 이를 적절히 수행한다. 또한 BH에서 falling edge일 경우 종료 타이머를 3초로 설정하여 등록한다.

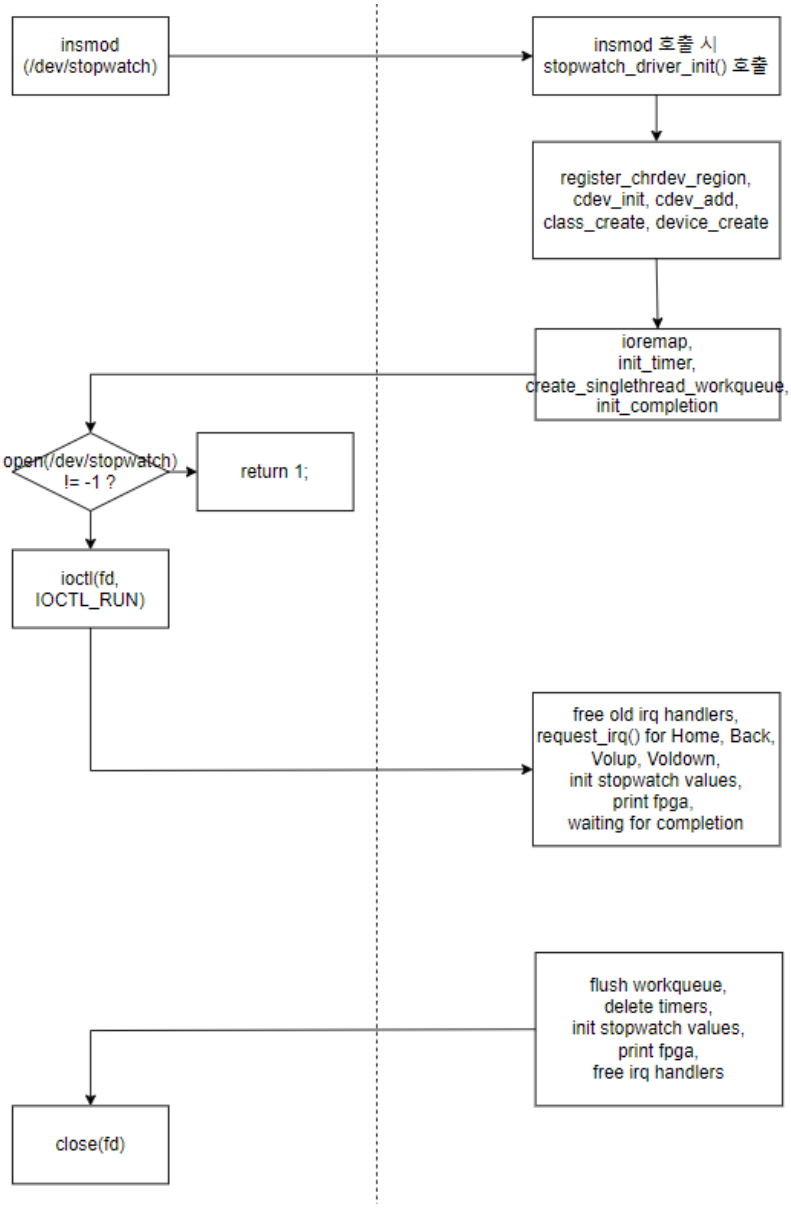
5. Timer Interrupt Handling Flow Chart



<그림 17: Timer Interrupt Handling Flow Chart>

좌우측 각각 flow chart는 fpga를 증가시키는 타이머와 voldown 종료조건 3초를 측정하는 타이머를 나타낸다. 종료 타이머의 경우 TH에서 전역 stopwatch 값에 종료가 되었다 표시한 후 BH에서 ioctl을 호출한 프로세스를 깨운다.

6. Kernel And app.c Flow Chart



<그림 18: Kernel And app.c Flow Chart>

점선 기준 좌측은 app.c 실행 후 프로세스가 user level 일때를, 우측은 kernel level 일때를 나타낸다. 프로세스는 waiting_for_completion을 통해 sleep한다. 이 다음 단계는 voldown interrupt handling 과정을 통해 깨워지지 않으면 진행되지 않는다. 깨워진 후에는 정의되지 않은 행동을 일으킬 수도 있으므로 혹시 남아있는 timer나 workqueue들을 삭제한다. 그 후 stopwatch 값들을 초기화하고 fpga도 종료상태로 만든다. 또한 혹시 다른 프로그램에서도 사용될 수 있을 가능성이 있으므로 등록된 irq핸들러들은 삭제한다.

V. 기타

학부과정 중 SWI를 다루는 경험은 종종 있었습니다. 그러나 커널 레벨에서 하드웨어 인터럽트를 설정하고 직접 다루며 Top Half, Bottom Half로 나누어 핸들링하는 것은 정말 새로운 부분이었고, 이를 통해 많은 점을 배울 수 있어서 정말 즐거운 경험이었습니다.