Embedded System Software 과제 3

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어
담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20171643, 박준혁
개발기간: 2023. 05. 18. - 2023. 05. 31.

최 종 보 고 서

I. 개발 목표

본 임베디드시스템소프트웨어 과제3는 지난 과제2와 유사하게 FPGA Board 상에 명세서 요구 조건을 만족시키는 Device Driver를 개발하고 해당 Driver를 호출하는 Application Program을 만들어 검증하는 것이 목표이다. 본 과제에선 Stopwatch를 구현한다. **사용자의 Key 입력에 따라 시간 측정을 시작하거나 멈출 수 있는, 또는 다시 Reset할 수 있는, 그러한 일반적인 기능을 모두 갖춘 Stopwatch를 개발**하는 것이 목표이다.

이를 위해 지난 과제1, 2에서 제공된 FPGA Board I/O Devices Default Driver Module 코드를 활용해 본 과제에서 필요한 기능을 제공할 수 있도록 I/O 제어 루틴을 구축한다. 여기선 FND Device만 사용하며, 해당 Device에 대한 I/O를 Stopwatch Driver Module에서 적극 활용한다. 배부된 'The Linux Kernel Module Programming Guide' 교재에 나와 있는 Module 예제들을 토대로 Stopwatch Driver의 틀을 만들고, 해당 Module에서 FND Device 제어 루틴들을 호출하는 것이다. 이때, 본 과제3에선 Key 입력을 받아들이는데, 이는 이전과 다르게 Device Driver가 아닌 Interrupt 방식으로 처리한다. 해당 Button 입력에 대한 Interrupt Handler를 각각 설치하고, Interrupt 발생 시 알맞은 Handler가 호출되어 일을 처리하는 것이다. 이때, Handler의 업무가 길어짐에 따라 Top & Bottom Half 개념이 도입될 것이다. 이러한 Timer, Interrupt 처리 부분은 본 과목 강의 시간에 학습한 내용을 토대로 어렵지 않게 구현할 수 있다.

한편 위와 같은 과정을 통해 만들어진 Stopwatch Driver Module을 Cross-Compile 후 보드로 전송 후, 'insmod' Command를 통해 Module을 설치한다. 'mknod'를 통한 Driver File을 생성하는 것도 중요하며, 이를 통해 **FPGA Board 내에 설치된 Stopwatch Driver를 이용하는 Application을 별도로 만들어 Stopwatch Driver의 정상 동작을 검증**한다.

이런 흐름을 거쳐 수행된 본 과제3의 제출 코드는 PDF 명세서에서 요구하는 모든 조건을 만족하고 있다.

II. 개발 범위 및 내용

가. 개발 범위

본 항목에선 과제3 수행 간에 고려해야 할 개발 범위 및 내용에 대해 다룬다. 총 세 Phase('나' 항목부터는 두 번째 Phase에 한해 두 개의 하위 Phase를 추가해 설명할 예정)로 나누었으며, 각각 **FPGA FND Device Control, Stopwatch Driver Module with Timer & Interrupt Handling, Application for Verification**이다.

(1) FPGA FND Device Control

본 과제3에서 가장 간단한 부분이다. 지난 과제1, 2에서 사용했던 FND Device Driver 코드를 기반으로 간단하게 FPGA FND Device I/O Controlling Routines를 하나의 소스 코드로 제공하면 된다. 기존 과제들과 다른 부분이 없기에 더 이상의 설명은 생략한다.

(2) Stopwatch Driver Module

이어, 본 과제의 핵심인 Stopwatch Driver를 Module 형태로 개발한다. Stopwatch Driver는 (1)번 항목에서 언급한 FPGA FND Device Controlling Routines를 이용해 PDF 명세서에 언급된 서비스를 외부로 표시한다.

내부 핵심 기능인 '시간 측정 기능'은 지난 과제2와 유사하게 '연속적인 Timer 설치'를 통해 구현한다. 1초 간격으로 Timer를 설치하고 동작시킴으로써 시간을 측정해간다. 이를 위해 Module 내 특정 함수에서 Timer를 Register하고 제어하는 루틴이 필요하다.

또한 본 Driver에서 Key(Button) 입력은 Interrupt를 이용해 처리해야 하기 때문에 적절한 Interrupt Handling Routines도 설치한다. 이를 통해 입력된 Button 정보는 Stopwatch의 '시간 측정'을 시작하거나 멈추는 등의 통제 기능을 수행한다.

즉, **Stopwatch Driver는 FPGA FND Device 통제, Timer 통제, 그리고 Interrupt 통제를 유기적으로 수행**하며 동작해 PDF 명세서가 요구하는 서비스를 제공하게 된다. 자세한 개발 과정은 이하 항목들에서 소개할 예정이다.

(3) Application for Verification

(1)과 (2)를 통해 Stopwatch Driver 구축이 완료되면 해당 Driver Module을 개발 PC에서 Cross-Compile 후 FPGA Board로 ADB Push한다. 그렇게 Module이 보드로 이식되면 우리는 minicom을 통해 해당 Module을 실습 보드에 설치할 수 있다. 하지만 그렇게 설치한 Module을 테스트하는 Application이 없다. 따라서 우리는 **Stopwatch Driver를 Open해 Stopwatch를 동작시키는 Application을 설계**한다.

Application이 할 일은 많지 않다. PDF 명세서에 언급된 것처럼 Console 입력 없이 그냥 Stopwatch를 On시키는 일만 담당한다. 즉, Stopwatch Driver를 Open하고 Driver로 write Command를 날려 Stopwatch를 Trigger하면 된다. 자세한 내용은 마찬가지로 아래에서 확인하도록 하자.

나. 개발 내용

본 항목에선 상기 (가) 항목에서 제시한 개발 Phase (1), (2), (3)의 순서대로 구체적인 개발 내용을 기술한다. 우선, FPGA FND Device를 제어하기 위한 개발 과정부터 소개할 것이며, Code-Level Implementation은 제외한다.

(1) FPGA FND Device Control

FPGA Board I/O Device 통제 루틴 설계는 **지난 과제2에서 개발한 내용과 정확히 일치하고, 또한 본 과제3에선 FND Device만 사용하기 때문에 굉장히 간단**하다. 배부된 예시 Drivers 중 FND Device Driver를 참조해 간단히 Device Control Routine을 만들 수 있다. 과제2에서 언급했듯, FPGA 기본 제공 I/O Device Driver는 모두 공통적으로 다음의 순서로 동작한다.

1) "ioremap(Bus Address of Target Device, Size of Resource to map)"을 통해 FPGA Target Device와 연결. Bus Address 및 Size 정보는 예시 Driver를 그대로 참조.

2) 연결된 Device에 "outw(Data, Connetion Pointer)"을 통해 Output 명령 하달.

요약하면, "ioremap으로 연결하고, outw로 출력한다"이다. 따라서 이 Principle을 따라 FND Device 통제 루틴을 마련한다. 파일명은 'fpga_fnd.c'라 하자. 해당 파일에 정의된 루틴들은 Stopwatch Driver에서 호출할 예정이다. 지난 과제2와 마찬가지로 printk 함수를 통한 Logging도 마련한다.

(2) Stopwatch Driver Module - Module

Stopwatch Driver Module 구축은 앞서 언급한 것처럼 'The Linux Kernel Module Programming Guide'에 나와 있는 예시 Module을 적극적으로 차용해 개발한다. Module의 시작과 끝에서 자동으로 수행되는 **Module Init Routine, Module Exit Routine을 기본으로 두고, 거기에 추가로 File-related I/O Operation Commands(open, close, write)의 대체 Module Routine들을 정의**한다. 이어, 그렇게 정의된 '대체 Routines'를 File Operations Structure를 통해 '우회가 가능해지도록' 맵핑한다.

각 Module Function은 Convention을 준수하며 구축하면 된다. 예를 들어, Module Init / Exit Function의 경우 Character Device Register / Un-register와 Timer 설정 / 해제를 수행한다. 이때, (1)에서 정의한 FPGA FND Device Mapping / Un-mapping 루틴도 추가하도록 하자. open, close, write 대체 함수도 마찬가지로 Convention을 지키며 설계하도록 한다. 자세한 Code-Level Details는 아래 '개발 방법' 항목에서 소개한다.

한편, 이렇게 **Module Interface 구축이 완료되면 그러한 Interface 함수들이 사용하는 Internal Functions도 정의**하자. Internal Function이라 함은, 본 과제3에서 요구하는 I/O 조건, Flow를 수행하는 함수들을 의미하며, Timer Instance 초기화, Timer Structure에 Timer Instance 등록과 같은 Timer-related Routines와, Button(Key) 입력을 통제하는 Interrupt-related Routines를 정의한다. 이는 (2-1), (2-2)에서 추가 설명한다.

본 항목의 자세한 Code-Level Details는 이하 III 항목에서 소개할 예정이다.

(2-1) Stopwatch Driver Module - Timer-related Routines

본 Stopwatch Driver에서 Timer는 앞서 설명한 내용처럼 Module Init 함수에서 설정된다. 설정이 끝나면 Stopwatch는 (User Application의 open() 호출로 인한 Open 함수 동작 이후) 사용자의 Button 입력(Press)을 기다린다. 이후 사용자가 'Home' Button을 클릭해 Stopwatch를 동작시키면 '시간 측정'이 시작되어야 한다.

Timer는 바로 이러한 부분에서 제 역할을 수행한다. 1초 간격으로 시작 지점에서 Timer가 Register되고, 1초 뒤 Timer가 만료되면 다시 또 다른 1초 주기의 Timer를 등록해 연속적으로 Timer 등록/해제를 반복하는 것이다. 이 과정에서 특정 Metadata에 이 '등록 횟수'를 기록해두면 **그것이 바로 '시간 측정'**이 된다.

이를 위해 본인은 1차적인 Wrapper Function(ex. _stopwatch_start())을 만들고, 해당 Wrapper에서 Timer를 다루는 방식으로 프로그램 구조를 설계하였다. 예시로 언급한 Start 함수에선 최초로 Timer를 등록하는 역할을 할 것이다. 그리고 이러한 Start 함수가 아래 항목에서 소개할 Interrupt Handling Routines에서 호출되어 'Key 입력에 따른 Stopwatch 제어' 기능을 제공할 것이다.

한편, **'시간 측정'을 위한 Timer 외에도 본 Stopwatch Driver에선 '종료를 위한 Timer'도 필요**하다. 과제 명세서에 따르면 Stopwatch 종료를 위해선 '3초간의 Vol-Down Button 입력'이 필요하기 때문에, 이를 Timer로 조절하는 것이다. 자세한 내용은 후술한다.

(2-2) Stopwatch Driver Module - Interrupt-related Routines

Stopwatch Driver에서 Interrupt Handling을 하는 이유는 'Button 입력 처리' 때문이다. 이를 위해선 우선 '특정 신호'에 대한 Interrupt를 설치하는 것이 중요하다. 미리 강의자료 및 실습 배부 자료 등에서 확인한 **GPIO(General Purpose I/O)를 이용**해 상응하는 IRQ Number를 받아놓는다. 이어 해당 IRQ Number를 이용하여 Handler를 'request_irq' 함수를 이용해 등록한다. 등록이 완료되면 이제 해당 Interrupt 신호 발생 시 등록한 Handler가 동작하게 된다.

허나, 강의 시간에서 배운 것처럼, 이러한 Interrupt Handler는 너무 많은 일을 해선 안 된다. 하지만 앞서 (2-1)에서도 언급하였듯 Interrupt Handling 시 Stopwatch 동작을 제어하고 Timer 관련 작업, I/O 작업 등을 수행해야 한다. 이때 필요한 것은 바로 Top & Bottom Half 개념이다. Interrupt Handling을 두 Phase로 나누어, 'Interrupt가 걸리자마자 바로 수행되어야 하는 급박한 업무들'은 Top Half에서 수행하고, '그 외 나머지 시간이 오래 걸리는 일들'은 Bottom Half에서 수행하는 것이다. 주로 전자에선 Data 저장 및 Bottom Half Scheduling을 담당한다.

이러한 **Top & Bottom Half 처리에는 일반적으로 다음과 같이 3개의 방법이 존재**한다.

- **Softirqs** // Kernel Source를 고치고자 할 때 적합
- **Tasklets** // User Source 설계 적합 (Interrupt Context)
- **Workqueues** // User Source 설계 적합 (Process Context)

모두 각자의 장단점을 지니고 있는데, 본인은 이중 이미 유사하게 다뤄본 경험이 있는 Workqueue를 사용하기로 결정했다(지난 과제2와 현 과제3에서 모두 Module Open 시 Concurrency를 제공하는 과정에서 Waitqueue를 사용). 또한, Workqueue는 Schedulable하고 세 방식 중 가장 사용 방법이 직관적이다. 아무튼 결론적으로 Workqueue를 사용해 Bottom Half를 다룰 것이다. 어떻게? 바로 앞서 언급한 **stopwatch_start()와 같은 핵심 Stopwatch 기능을 담당하는 Internal Function들을 Top Half로** 만들고, **나머지 비-핵심 파트는 Bottom Half로 만들어줌**으로써 말이다. **비-핵심 루틴을 Work로 지정하고, 해당 Work를 Work Queue에 Runtime 삽입**하는 것이다. **Workqueue로 들어간 Bottom Half들은 추후 Kernel 내 특정 Thread가 자동으로 수행시킬 것**이다. (상세 내용 후술)

(3) Application for Verification

완성된 Stopwatch Driver를 Open하고 Trigger하는 Application은 상기 '개발 범위 - (3)'에서 설명한 그대로이다. 추가적인 설명은 필요치 않으며, 아래 '개발 방법' 항목에서 간략히 Code를 설명하겠다.

III. 추진 일정 및 개발 방법

가. 추진 일정

본인은 다음과 같은 일정을 통해 과제2를 수행하였다.

- 05/18~05/19: 과제 이해 및 사전 학습
- 05/19~05/28: Module & Application 프로그래밍 및 검증
- 05/28~05/31: 개발 완료 및 보고서 작성

본 과제3은 지난 과제2와 유사한 난이도이며, 약 열흘의 개발을 통해 과제를 완성하였다. 이어 아래 '개발 방법' 항목에서 본 과제 제출 코드에 대한 Code-Level 설명을 이어간다.

나. 개발 방법

지난 과제들과 마찬가지로 Cross-Compile 환경에서 C Language를 이용해 Module과 Application을 개발 PC에서 작성하였으며, 작성된 Module과 Application을 순차적으로 소개한다. **'stopwatch_driver.c'는 Module을 구성하며, 'fpga_fnd.c'는 FPGA FND Device 제어 루틴을, 'app.c'는 Application Program을 정의**한다.

(1) FPGA FND Device Control (fpga_fnd.c)

우선 'fpga_fnd.c'에서 사용할 **FPGA FND Device 관련 Metadata를 정의**한다. Meta--data에는 ioremap에 Parameter로 넘어갈 Bus Address와 Buffer Size가 대표적이다. 이어, 해당 Metadata를 활용해 필요한 I/O 제어 루틴(Map, Unmap, Write)을 설계한다. 아래를 보자. (write에서 시간을 '분/초'로 출력하는 부분 주목)

```c
/* Mapped address of FND device */
static char *fnd_addr;

/* Bus address of FND device */
#define FND_ADDR 0x08000004
#define FND_MAP_SIZE 0x2

/* Map the NFD device provided by our FPGA board */
int fpga_fnd_map(void) {
    printk("[FND DEVICE] Open the FND device\n");

    fnd_addr = ioremap(FND_ADDR, FND_MAP_SIZE);
    if (fnd_addr == NULL) {
        printk("[FND DEVICE] Failed to map the FND device\n");
        return FND_ERROR;
    }

    return 0;
}
```

```c
/* Unmap the FND device provided by our FPGA board */
void fpga_fnd_unmap(void) {
    printk("[FND DEVICE] Close all the open I/O devices\n");
    iounmap(fnd_addr);
}

/* Print the time(min:sec) to FND device */
void fpga_fnd_write(const int min, const int sec) {
    int i, time; unsigned short value = 0;
    time = 100 * min + sec;
    for (i = 0; i < 4; i++, time = time / 10)
        value += (time%10) << (4*i);

    outw(value, (unsigned int)fnd_addr);
    printk("[FND DEVICE] FND WRITE: (%02d:%02d)\n", min, sec);
}
```

배부된 예시 Device Driver와 같이 outw함수를 이용해 Write를 수행하고 있음을 주목하자. 이렇게 해서 FPGA FND Device Control Routine을 간단하게 구축할 수 있다. (2)번의 Stopwatch Driver Module에선 이를 적재적소에 호출하며 FND Device를 조정할 것이다.

(2) Stopwatch Driver Module - Module (stopwatch_driver.c)

이번엔 Stopwatch Driver Module을 설계하자. 우선 아래와 같이 **Stopwatch Driver에서 사용할 Metadata를 'stopwatch_driver.h'에 Structure로 정의**하자. '시간 측정에 쓰일 Timer'부터 현재까지 지나간 시간(sec 기준), Stopwatch가 Pause된 경우 해당 Paused 시간(0.xx초 단위) 등이 포함되어 있음을 주목하자.

```c
/* Timer to register, and some metadata of it */
typedef struct _timer {
    /* Timer to register */
    struct timer_list timer;

    /* Metadata of stopwatch */
    unsigned int elapsed_sec;    /* Elapsed seconds for the stopwatch */
    unsigned long prev_expires;  /* Time that the timer expired lastly (right before) */
    unsigned long paused_sec;    /* Time that the stopwatch paused lastly */
} Timer;
```

사실 Metadata는 이뿐만이 아니다. Stopwatch Driver에선 사용하는 루틴이 다양(Timer 뿐만 아니라 Interrupt Handling도 존재)하기 때문에 각 기능에 따른 Metadata를 별도로

정의해주어야 한다. 아래와 같이 말이다.

```c
/* Device file name and the major number (PDF conditions) */
#define DEV_NAME "stopwatch"
#define DEV_MAJOR_NUM 242

/* Main interfaces of this module */
static int __init stopwatch_driver_init(void);
static void __exit stopwatch_driver_exit(void);
static int stopwatch_driver_open(struct inode *, struct file *);
static int stopwatch_driver_close(struct inode *, struct file *);
static int stopwatch_driver_write(struct file *, const char __user *, size_t, loff_t *);

/* Redirect file I/O functions to user-defined handlers */
static struct file_operations stopwatch_driver_fops = {
    .owner = THIS_MODULE,
    .open = stopwatch_driver_open,
    .release = stopwatch_driver_close,
    .write = stopwatch_driver_write
};

/* Data structures for providing blocking */
static atomic_t already_open = ATOMIC_INIT(0);
static DECLARE_WAIT_QUEUE_HEAD(waitq);

/* Internal main functionalities of stopwatch driver */
void _stopwatch_init(void);
void _stopwatch_start(void);
void _stopwatch_pause(void);
void _stopwatch_exit(void);

/* Maximum seconds (3600sec == 1hour) */
#define TIME_LIMIT (60 * 60)   // "_ _:_ _" (min:sec)

/* Global timers we use here */
static Timer tm;
static struct timer_list exit_tm;

/* Internal functions to manage metadata attached to timer and perform operations */
static void _timer_handler(unsigned long);
static void _timer_register(void);
static void _timer_exit_handler(unsigned long);

/* Wait queue for interrupt handling (button press) */
static DECLARE_WAIT_QUEUE_HEAD(intrq);

/* Internal functions to manage interrupts */
void _intrs_register(void);
void _intrs_free(void);
/* # of buttons we use here */
#define BUTTON_CNT 4

/* Enum type for button selection (interrupt handling) */
typedef enum _button_type {
    INIT, HOME, BACK, VOL_UP, VOL_DOWN
} Button;

/* Internal handlers for each interrupt (button pressing) */
static irqreturn_t _home_button_handler(int, void*);
static irqreturn_t _back_button_handler(int, void*);
static irqreturn_t _volup_button_handler(int, void*);
static irqreturn_t _voldown_button_handler(int, void*);

/* Workqueue infrastructure for top and bottom half handling */
static struct workqueue_struct *workq;     // work queue
typedef struct {
    struct work_struct _work;              // work to be registered
    Button button;                         // which button is pressed
} Work;
Work *work;
static void work_function(struct work_struct *work);  // handler

/* Record the lastly pressed button for proper interrupt handling */
static Button prev_pressed = INIT;

/* Stopwatch paused flag */
static int stopwatch_paused = 0;

/* Volume down pressed flag */
static int vol_down_pressed = 0;

/* GPIO Number creator for each button */
static unsigned buttons_gpio[BUTTON_CNT] = {
    IMX_GPIO_NR(1, 11),  // HOME
    IMX_GPIO_NR(1, 12),  // BACK
    IMX_GPIO_NR(2, 15),  // VOLUME UP
    IMX_GPIO_NR(5, 14)   // VOLUME DOWN
};
```

```c
/* Name strings for each button */
static char *buttons_name[BUTTON_CNT] = {
    "home", "back", "vol+", "vol-"
};

/* Flags used for requesting IRQ number (for each button) */
static int button_flags[BUTTON_CNT] = {
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_RISING,
    IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING
};

/* Pointers for each button interrupt handler */
void (*button_handlers[BUTTON_CNT])(int, void*) = {
    (void*)_home_button_handler,
    (void*)_back_button_handler,
    (void*)_volup_button_handler,
    (void*)_voldown_button_handler
};
```

그렇다. 매우 많다. 위에서부터 항목별로 하나하나 차례대로 설명하겠다.

- **DEV_NAME, DEV_MAJOR_NUM** : Driver Register 시 사용할 Device Driver File 명과 사용할 Major Number를 의미한다.

- **file_operations 부분** : 본 Device Driver에서 제공하는 File Operations 대체 함수들의 Mapping 관계를 보여준다.

- **already_open과 waitq** : 본 Device Driver를 Multiple Applications가 Concurrent하게 Open하는 상황을 방지하기 위한 루틴으로, Kernel Programming Guide를 기반으로 작성하였다.

- **'_stopwatch_init'와 같은 함수**는 Internal Function으로, 본 Driver가 Interrupt를 Handling하는 과정에서 부를 함수들이다. 이들은 모두 이미 언급한 바와 같이 Timer를 다루며 Stopwatch의 핵심 서비스를 제공한다.

- **Timer인 tm과 exit_tm**은 각각 '시간 측정을 위한 Timer'와 '3초 종료 조건을 만족하기 위한 Timer'이다.

- **intrq**는 User Application을 잠재우기 위한 Queue이다.

- **_home_button_handler와 같은 함수**는 Button 입력에 의한 Interrupt를 처리하는 Hanlder로서, Top Half에 해당한다. 상기한 'stopwatch_init'과 같은 함수들도 Top Half에 해당한다. 나머지 비-핵심 기능은 Bottom Half화 하는데, 자세한 내용은 후술한다.

- **workq와 work 등**은 Bottom Half Scheduling을 위한 Workqueue 구축 과정에서 필요한 Global Data들이다. ('button'이란 Field에 대해선 후술)

- **prev_pressed 변수**는 '직전에 입력된 Button이 무엇인지'를 나타내어 올바른 Button 입력 처리를 하기 위해 존재한다. (**stopwatch_paused, vol_down_pressed**도 유사한 목적)

- **buttons_gpio 배열**은 각 Button에 대한 Interrupt를 의미한다. 이들을 통해 Handler를 등록할 것이다. **button_flags 배열**은 그러한 GPIO를 이용해 request_irq 함수 호출 시 호출 간에 사용할 옵션들을 의미하며, **button_handlers**는 '_home_button_handler'와 같은 Top Half들을 가리킨다. 이들은 모두 Interrupt 등록/해제 간에 간단히 반복문을 사용하고자 도입되었다.

이렇게 Metadata 설명을 마친다. 굉장히 많지만, 하나하나 전부 Stopwatch Driver 동작 간에 필요한 Data들이다. Timer, Interrupt 부분을 제외하고 나머지 Module 구성에 대해서도 간단히 더 알아보자. (Timer와 Interrupt는 별도의 소항목으로 소개)

우선 Module Init 함수부터 보자(아래). Character Device Register, FPGA I/O Devices Mapping, Timer Install, Workqueue 생성 등의 행위가 이뤄지고 있음을 주목하자.

```
/* Register the device driver provided by this module file, triggered by 'insmod' cmd */
static int __init stopwatch_driver_init(void) {
    int ret_val;
    if ((ret_val = register_chrdev(DEV_MAJOR_NUM, DEV_NAME, &stopwatch_driver_fops)) < 0 ) {
        printk("[STOPWATCH DRIVER] Module register failed\n");
        return ret_val;
    }
    printk("[STOPWATCH DRIVER] Device created on file(/dev/%s, major: %d)\n",
        DEV_NAME, DEV_MAJOR_NUM);

    /* Map the FND device in our FPGA board */
    if (fpga_fnd_map() == FND_ERROR)
        return FND_ERROR;
    /* Be prepared to use timer */
    init_timer(&(tm.timer));
    init_timer(&exit_tm);

    /* Create the work queue for bottom half handling */
    workq = create_workqueue("my_queue");
    printk("[INTERRUPT] Workqueue is created\n");

    return ret_val;
}
```

마찬가지로 Module Exit 함수를 Convention에 맞게 구축할 수 있다. 자, 이제 이어서 Open 대체 함수를 보자. **Concurrent Open 시 'I/O Blocking'을 제공하기 위해 Atomic Operations를 이용해 Open Flag를 제어하고, 동시에 Wait Queue 관리를 수행**하고 있음을 알 수 있다. O_NONBLOCK Flag를 통해 Open한 Process는 앞서 Timer Driver를 호출한 Process가 이미 존재할 경우 Open에 실패할 것이며, 그 외의 Process는 Open 시 앞선 Process가 있을 때 Wait Queue로 들어가 자신의 순서를 기다리게 될 것이다. 이때, 무엇보다 주목할 지점은 바로 **'_intrs_register()' 함수 호출하여 Interrupt Handler를 등록**하고 있음이다. Interrupt Handling이 가능해짐을 의미한다.

```
/* open() Handler for this driver */
static int stopwatch_driver_open(struct inode *inode, struct file *file) {
    /* If file's flag include O_NONBLOCK and is already open, then decline it */
    if ((file->f_flags & O_NONBLOCK) && atomic_read(&already_open))
        return -EAGAIN;

    /* Notify that this module is used now */
    try_module_get(THIS_MODULE);

    while (atomic_cmpxchg(&already_open, 0, 1)) {
        int i, is_sig = 0;

        /* If the module is used (when BLOCK mode), then put cur process to sleep */
        wait_event_interruptible(waitq, !atomic_read(&already_open));

        /* If the slept process is woke up by getting unpredictable signals */
        for (i = 0; i < _NSIG_WORDS && !is_sig; i++)
            is_sig = current->pending.signal.sig[i] &
                ~current->blocked.sig[i];

        if (is_sig) {
            module_put(THIS_MODULE);
            return -EINTR;
        }
    }

    /* Allow the access */
    printk("[STOPWATCH DRIVER] %s called\n", __func__);
    /* Register all the interrupts that this driver uses */
    _intrs_register();

    return 0;
}
```

한편, open 함수에 맞추어 Close 대체 함수도 적절히 구성한다. Close와 동시에 Open Flag를 Unset해 Close 여부를 알리고, Wait Queue에서 기다리고 있는 모든 Process를 깨워 Race를 시키고 있음을 주목하자.

```
/* close() Handler for this driver */
static int stopwatch_driver_close(struct inode *inode, struct file *file) {
    /* Set already_open to zero, so one of the processes in the waitq will be
       be able to set already_open back to one and to open the file. All the
       other processes will be called when already_open is back to one, so they
       will go back to sleep */
    atomic_set(&already_open, 0);

    /* Wake up all the processes in waitq, so if anybody is waiting for the file
       they can have it */
    wake_up(&waitq);

    module_put(THIS_MODULE);

    printk("[STOPWATCH DRIVER] %s called\n", __func__);

    /* Release all the interrupts used */
    _intrs_free();

    return 0;
}
```

다음은 Write 대체 함수이다. User Application에서 write() 함수 호출 시 동작한다. Stopwatch를 초기화한 후 User Application을 'interruptible_sleep_on' 함수로 잠재우고 있음을 주목하자.

```
/* write() Handler for this driver */
static int stopwatch_driver_write(
    struct file *file, const char __user *buf, size_t count, loff_t *f_pos
) {
    printk("[STOPWATCH DRIVER] %s called\n", __func__);

    /* Initialize the stopwatch */
    _stopwatch_init();
    /* Put the user application on a sleep */
    interruptible_sleep_on(&intrq);
    printk("[STOPWATCH DRIVER] Make user application sleep\n");

    return 0;
}
```

(2-1) Stopwatch Driver Module - Timer-related Routines (stopwatch_driver.c)

지금부터 Stopwatch Driver에서 'Service 제공'을 담당하는 Internal Functions와 Timer 관리 과정에 대해 알아보자. 여기엔 **Timer Metadata 초기화, Timer Instance 등록, FPGA I/O Devices 출력, Timer 만료 시 수행되는 Handler 등의 다양한 작업이 수행**된다. 우선 아래의 Internal Functions부터 확인해보자.

```
/* Internal functionalities of stopwatch */
/* Initialize all the metadata of stopwatch */
void _stopwatch_init(void) {
    /* Reset the FND device */
    fpga_fnd_write(0, 0);
    /* Initialize the metadata */
    tm.elapsed_sec = tm.paused_sec = 0;
}
```

```
/* Start the stopwatch device */
void _stopwatch_start(void) {
    /* Delete possibly existing timer */
    del_timer(&(tm.timer));

    /* Register the next timer */
    _timer_register();
}
```

'_stopwatch_init' 함수를 보면, 시작과 동시에 FND Device를 00:00으로 초기화한다. 이

어, Metadata에서 Elapsed 초를 0으로 초기화한다. PDF 명세서 조건을 그대로 구현하는 것이다. 한편 '_stopwatch_start' 함수를 보면 해당 함수에선 기존 Timer를 없애고 새 Timer를 등록하고 있음을 알 수 있다. 말 그대로, Stopwatch를 시작하고 있음을 알 수 있다. 이제 아래의 남은 두 함수도 확인하자.

```
/* Pause the stopwatch device */
void _stopwatch_pause(void) {
    int paused_time = get_jiffies_64() - tm.prev_expires;

    /* Delete the current timer */
    del_timer(&(tm.timer));

    /* Record the paused time, the function below
       will register the timer after the re-start */
    tm.paused_sec = paused_time;
}

/* Exit the stopwatch, that is, start the exit timer */
void _stopwatch_exit(void) {
    /* Remove the previous one */
    del_timer(&exit_tm);

    /* Set the exit timer (3 seconds of pressed state) */
    exit_tm.expires = get_jiffies_64() + 3 * HZ;  // 3 seconds
    exit_tm.function = _timer_exit_handler;

    /* Register to the list */
    add_timer(&exit_tm);
}
```

Pause 함수에서도 마찬가지로 Timer를 삭제하고 있다. 하지만, 이때는 삭제와 더불어 삭제된 시간을 기록하고 있다. 이때 시간을 반환하는 get_jiffies_64() 함수를 사용하고 있다. 이는 강의 시간에 다룬 것처럼 (1/HZ)초에 한 번씩 Increment가 되는 값을 반환하는데, 우리 실습 Board 환경에선 **"0.xx" 단위**의 초를 반환함을 의미한다. 우리는 이를 기반으로 PDF 명세에 적힌 '버튼 입력 시 소수점 1번째 자리까지의 시간이 유지되어야 한다'는 조건을 충족시킬 것이다. (어떻게 충족시키는지는 아래서 설명)

'_stopwatch_exit' 함수에서는 Exit Timer를 등록하고 있음에 주목하자. Exit Timer는 앞서 언급한 바와 같이 3초를 기다린 다음 Stopwatch를 종료시키고 User Application을 깨우는 역할을 담당한다. 이 처리가 여기서 이루어지는 것이다. 즉, 엄밀히 말하면 'Exit을 시도하는 함수'라고 보면 되겠다(Button이 3초가 흐르기 전에 놓아질 수 있으므로).

이러한 **Internal Functions는 보다시피 결국 Button 입력에 따라 호출 여부가 결정될 것이기 때문에 후술할 Interrupt Handler에서 다시 한번 등장할 것이다. 이미 한 번 언급한 바와 같이, 이들은 Top Half Routines의 일부**이다. 우선 본 항목에선 이를 잠시 제쳐두고 Timer 관련 부분을 더 살펴보자.

아래(다음 장) Timer 등록 함수이다. Timer 등록 함수에선 1초를 기준으로 Timer 시간을 설정하고, 만료 시 _timer_handler 함수가 돌아가도록 설정하고 있음을 주목하자. 여기서 주목해야 할 부분은 바로 're_start_time'을 Expiration Time 측정 간에 제외하고 있다는 것인데, 이는 위에서 언급한 '버튼 입력 시 소수점 1번째 자리까지의 시간이 유지되어야 한다'는 조건을 충족시키기 위함이다. 그렇다. 즉, Pause된 이후 다시 Resume할 때 이 부분이 제 역할을 다할 것이다.

이렇게 설치된 Timer는 연속적인 Chain처럼 동작해 '시간 측정'을 수행한다.

```c
/* Internal Handlers Definitions */
/* Register the next timer for stopwatch functionality */
static void _timer_register(void) {
    int re_start_time = tm.paused_sec;

    /* Set the timer */
    tm.timer.expires = get_jiffies_64() + HZ - re_start_time;
    tm.timer.data = (unsigned long)&tm;
    tm.timer.function = _timer_handler;

    /* Register to the list */
    add_timer(&(tm.timer));

    tm.paused_sec = 0;
    printk("[TIMER] Register the next timer (restart time: 0.%dsec)\n", re_start_time);
}

/* Callback handler function that is expected to be called whenever the timer expires */
static void _timer_handler(unsigned long timeout) {
    Timer *tm = (Timer*)timeout;
    printk("[TIMER] Timer Handler is called (elapsed_sec=%d)\n", ++(tm->elapsed_sec));
    /* Increment the seconds and record the callback function call time */
    tm->elapsed_sec = tm->elapsed_sec % TIME_LIMIT;
    tm->prev_expires = get_jiffies_64();

    /* Print the time to the FND device */
    fpga_fnd_write((tm->elapsed_sec / 60), (tm->elapsed_sec % 60));

    /* Register the next timer */
    _timer_register();
}
```

매 1초가 만료될 때마다 FND Device에 현재 시간을 출력하고 다시 Timer를 등록하고 있음을 주목하자. (_timer_handler 함수 내부)

아래는 '_stopwatch_exit' 함수가 Trigger할 **Exit Timer가 Expire했을 때 수행되는 Exit Timer Handler**이다. 별다른 부분은 없고, FND Device를 초기화하고 User Application을 깨우고 있음을 주목하자.

```c
/* Callback handler function for the exit timer */
static void _timer_exit_handler(unsigned long timeout) {
    /* Ends the stopwatch. */
    del_timer(&(tm.timer));
    printk("[TIMER] Stopwatch ends here (exit timer expires now)\n");

    /* Reset the I/O of FPGA */
    fpga_fnd_write(0, 0);

    /* Wakeup the application */
    printk("[INTERRUPT] Wake up user application (retrieve from WaitQ)\n");
    __wake_up(&intrq, 1, 1, NULL);
}
```

(2-2) Stopwatch Driver Module - Interrupt-related Routines (stopwatch_driver.c)

이제 대망의 **Interrupt Handling 부분**을 살펴보자. 앞서 정의한 Internal Functions들을 호출하는 Interrupt Handler들을 각각 Button에 대해 정의해야 할 것이며, 그들을 일괄적으로 등록 및 해제하는 루틴도 필요할 것이다. 우선 Interrupt 등록/해제 과정부터 확인하자. 아래를 보자. (III 항목 시작부의 button_xxx 배열들을 기억하라)

```c
/* Register all the interrupts that this driver uses */
void _intrs_register(void) {
    int i, irq, ret;
    /* Register */
    for (i = 0; i < BUTTON_CNT; i++) {
        gpio_direction_input(buttons_gpio[i]);
        irq = gpio_to_irq(buttons_gpio[i]);
        ret = request_irq(irq, (irq_handler_t)(*button_handlers[i]),
            button_flags[i], buttons_name[i], 0);
        printk("[INTERRUPT] IRQ number %d for '%s' button\n", irq, buttons_name[i]);
    }

    /* Intialize metadata */
    prev_pressed = INIT;
    vol_down_pressed = 0;
}

/* Free all the interrupts */
void _intrs_free(void) {
    int i;

    for (i = 0; i < BUTTON_CNT; i++) {
        free_irq(gpio_to_irq(buttons_gpio[i]), NULL);
        printk("[INTERRUPT] Button '%s' is freed now\n", buttons_name[i]);
    }
}
```

request_irq 함수를 사용해 각 Button 입력의 Interrupt 신호에 대해 알맞은 Interrupt Handler (Top Half)를 등록하고 있다. 앞서 초반에 확인한 것처럼 이들을 모두 배열화했기 때문에 위와 같이 간소한 코드를 작성할 수 있다. 해제 시엔 free_irq 함수를 사용한다.

아래는 'Home' Button Interrupt에 대한 Handler (Top Half)이다.

```c
/* Interrupt handler for 'home' button, it pauses the stopwatch */
static irqreturn_t _home_button_handler(int irq, void* dev_id) {
    int ret;

    /* Continuous pressing handling */
    if (prev_pressed == HOME) {
        printk("[INTERRUPT] Continuous 'home' button pressing ignored\n");
        return IRQ_HANDLED;
    }
    prev_pressed = HOME;
    _stopwatch_start();

    /* Make and schedule the bottom half (printks) */
    work = (Work *)kmalloc(sizeof(Work), GFP_KERNEL);
    if (work) {
        INIT_WORK((struct work_struct *)work, work_function);
        work->button = HOME;
        ret = queue_work(workq, (struct work_struct *)work);
    }
    stopwatch_paused = 0;

    return IRQ_HANDLED;
}
```

이전에 눌린 Button이 Home Button인지 확인해, Home Button이라면 '연속 입력 상황'이므로 무시한다. 이는 'Home' Button 뿐만 아니라 모든 Button에서 마찬가지로 필요한 루틴이다. 이어, '_stopwatch_start' Internal Function을 호출하여 시간측정을 시작한다. 이때, 그다음 부분에서 **Work를 할당해 work_function이란 것을 맵핑시킨 후 해당 Work를 Workqueue에 삽입해 Schedule**하고 있는데, 이는 추후 Kernel Thread에 의해 자동으로 실행될 것이다(work_function 설명은 후술). **Interrupt Handler가 너무 길어지는 것을 방지**하기 위한 조치이며, 본 Handler 외에 다른 Handler에서도 마찬가지이다.

```c
/* Interrupt handler for 'back' button, it pauses the stopwatch */
static irqreturn_t _back_button_handler(int irq, void* dev_id) {
    int ret;

    /* Continuous pressing handling */
    if (prev_pressed == BACK || stopwatch_paused) {
        printk("[INTERRUPT] Continuous 'back' button pressing ignored\n");
        return IRQ_HANDLED;
    }
    /* When the stopwatch is running, only 'back' button pressing is allowed */
    if (prev_pressed != HOME && prev_pressed != VOL_DOWN) {
        printk("[INTERRUPT] 'back' button pressing ignored (stopwatch isn't running)\n");
        return IRQ_HANDLED;
    }
    prev_pressed = BACK;
    _stopwatch_pause();

    /* Make and schedule the bottom half (printks) */
    work = (Work *)kmalloc(sizeof(Work), GFP_KERNEL);
    if (work) {
        INIT_WORK((struct work_struct *)work, work_function);
        work->button = BACK;
        ret = queue_work(workq, (struct work_struct *)work);
    }
    stopwatch_paused = 1;  // For vol-down & back pressing situation

    return IRQ_HANDLED;
}
```

이어 'Back' Button 입력 처리 부분이다. 마찬가지로 '연속 입력'과 'Bottom Half'에 대한 처리가 있음을 주목하자.

'Back' Button은 Stopwatch를 Pause하는 역할인데, 그에 따라 '직전에 눌린 Button'이 Home Button이거나 Vol-Down Button일 때만 Pause를 수행함을 주목하자(Vol-Down일 때 Pause를 허용치 않으면 종료 시도 이후 Back이 먹히지 않는 현상이 발생한다. 따라서 이런 루틴이 필요하다).

'Vol-Up' Button에 대한 Handler도 마찬가지 논리로 구현할 수 있다. 이제 마지막으로 'Vol-Down' Button Handler를 확인하자. 아래를 보자.

```c
/* Interrupt handler for 'vol-' button, it make the stopwatch
   terminate if 'vol-' button is pressed more than 3 seconds */
static irqreturn_t _voldown_button_handler(int irq, void* dev_id) {
    int ret;

    /* If pressed while already pressed, then stop counting 3 secs */
    if (vol_down_pressed) {
        printk("[INTERRUPT] 'vol-' button pressing handled (counting ends)\n");

        del_timer(&exit_tm);
        printk("[TIMER] Existing exit timer is now deleted\n");
        prev_pressed = VOL_DOWN;
    }
    /* 'vol-' button is pressed, it should be pressed until 3 seconds are elapsed */
    else {
        _stopwatch_exit();

        /* Make and schedule the bottom half (printks) */
        work = (Work *)kmalloc(sizeof(Work), GFP_KERNEL);
        if (work) {
            INIT_WORK((struct work_struct *)work, work_function);
            work->button = VOL_DOWN;
            ret = queue_work(workq, (struct work_struct *)work);
        }
    }

    vol_down_pressed = 1 - vol_down_pressed;
    return IRQ_HANDLED;
}
```

위 코드에서 **else문 부분이 '3초간 꾹' 누르는 부분**이다. **"if (vol_down_pressed)"에 걸리면 '꾹 눌리다 만' 상황**을 의미한다. 즉, 최초 Button 입력 시 vol_down_pressed Flag는 0으로 세팅되어 있기 때문에 else로 온다. 이어 Exit Timer를 설치하고 Flag를 Set한다. 만약, 여기서 꾹 누르던 것을 놓으면 다시 한번 해당 Handler가 동작하게 된다(초반에 소개한 button_flags를 확인하라. Vol-Down Interrupt는 누를 때와 놓을 때 모두 발생하도록 설정되어 있다). 그러면 이제 if문에 걸리게 된다. if문에선 위 코드에서 알 수 있듯 Exit Timer를 제거한다. 본 Stopwatch의 Exit Routine은 이렇게 동작하는 것이다.

자, 마지막으로 **Workqueue에 삽입된 Bottom Half Work의 Routine인 work_function**을 확인하자. 'work_function'은 아래와 같이 정의되어 있다.

```
/* Workqueue function (bottom half function)
   It only prints logging strings since these are not important for stopwatch functionalities */
static void work_function(struct work_struct *work) {
    Work *_work = (Work *)work;
    printk("[WORKQUEUE] Bottom half routine is now handled\n");
    switch(_work->button) {
    case HOME:
        printk("[TIMER] Stopwatch starts now\n");
        printk("[INTERRUPT] 'home' button pressing handled\n");
        break;
    case BACK:
        printk("[TIMER] Stopwatch is paused now (paused time: 0.%dsec)\n", tm.paused_sec);
        printk("[INTERRUPT] 'back' button pressing handled\n");
        break;
    case VOL_UP:
        printk("[STOPWATCH DRIVER] Reset the stopwatch now\n");
        printk("[STOPWATCH] Stopwatch is now initialized\n");
        printk("[INTERRUPT] 'vol+' button pressing handled\n");
        break;
    case VOL_DOWN:
        printk("[TIMER] Register the new exit timer (VOL_DOWN pressed)\n");
        printk("[INTERRUPT] 'vol-' button pressing handled (counting starts)\n");
        break;
    default: break;
    }
    printk("[WORKQUEUE] Bottom half routine is now finished\n");

    kfree((void *)work);
    return;
}
```

각 Button에 대한 Interrupt Handling 시 필요한 Logging Print문들이 모두 모여있음을 확인할 수 있다. 즉, 본 과제 수행 간에 Interrupt Handling의 Top Half / Bottom Half는 다음을 기준으로 분류된 것이다.

"Stopwatch 기능의 핵심(Timer, Interrupt, FND I/O) 부분을 담당하고 있는가?"

앞서 확인한 바와 같이 Button이 눌리면 해당 Button이 발생시킨 Interrupt에 대한 Interrupt Handler가 동작한다. 그리고 해당 Interrupt Handler는 몇 가지 예외 처리를 한 후 Internal Function을 호출해 Stopwatch 기능(시간측정 시작/중지/초기화/종료)을 각각 Timer와 FND Device I/O를 이용해 수행한다. 여기까지는 **Stopwatch 기능을 결정짓는 중요한 부분**이다. 이들의 처리는 Synchronous하게 그 Interrupt 발생 순간 긴박하게 처리되어야 한다. 따라서 **이들을 모두 Top Half로 처리**하는 것이다. **그러면, 나머지 'Stopwatch 기능에 핵심이 아닌 부분'은 무엇일까? 그렇다. Logging Print문들**이다. 특정 Interrupt

Handling 및 기능 수행이 잘 이뤄지고 있음을 개발자 및 User에게 알려주는 이 Logging Function은 굳이 Stopwatch Driver 동작과 Synchronous할 필요가 없다. 따라서 이들을 Bottom Half로 분류하는 것이다. 이들을 Work로 만들어주고, Workqueue에 삽입해 추후 Kernel Thread가 수행하도록 만든다. Stopwatch 핵심 기능 수행을 늦추지 않도록 하는 것이다.

따라서, work_function을 위와 같이 Logging Print문을 수행하는 역할로 정의하였다. 앞서 확인했던 '_home_button_handler' 함수를 다시 확인해보자. **Work 할당 후 Workqueue에 삽입하기 직전에 해당 work 변수의 Button Field에 '현재 발생한 Interrupt'가 무엇인지 기록**해두고 있다. 이어, 시간이 흐른 후 Kernel Thread가 해당 Work를 Workqueue에서 Dequeue 후 수행 시 그 Work의 Button Field를 확인해 어떤 Interrupt Handling 과정에서 만들어진 Work인지 확인 후 알맞은 Logging Print문들을 출력하는 것이다. 이렇게 함으로써 Logging Print문이 Stopwatch 핵심 기능을 저해하는 일이 발생하지 않도록 만들 수 있다.

이렇게 해서 Stopwatch Driver 설계가 마무리되었다. 다시 한번 복기해보면, **사전에 마련한 FND Device Routine과 Timer Handling Routines를 이용해 Internal Function을 설계한다. 이어, 해당 Internal Function을 Top Half Interrupt Handler의 일부로 하여 각 Button에 대해 적절히 Interrupt Handling을 수행**한다. 그리고 나머지 Interrupt Handling 시의 Logging Functions 호출은 **Bottom Half**로 하여 Stopwatch의 신속한 동작이 저해되지 않도록 처리한다. 그리고 이와 별개로 Driver Init, Exit 함수가 돌아간다. Stopwatch Driver는 이렇게 돌아가는 것이다.

이제 구축된 Driver Module을 Cross-Compile해 .ko 파일을 생성한 후, 해당 파일을 FPGA Board로 넘겨 설치하면 끝이다. 하지만 아직 한 작업이 더 필요하다. 검증을 위한 Application이 필요한 것이다.

(3) Application for Verification (app.c)

Application은 계속해서 설명하는 것과 같이 굉장히 간단하다. Board 내에서 Module을 'insmod'로 설치하고 나면 **'mknod'를 이용해 그에 상응하는 Device Driver File을 만드는 것이 중요한데, Application에선 해당 파일명을 Driver 이름으로 해서 Open해 Stopwatch Driver를 사용**할 수 있다. 그렇게 Open하고 나면 write 함수를 이용해 Timer Setting과 Timer Driver Service 요청을 PDF 명세서대로 수행하면 된다. Application은 이게 핵심이고 전부이다.

```c
/* Device file for our device driver */
#define DEV_NAME "/dev/stopwatch"

/* Simple application to run /dev/stopwatch */
int main(void) {
    int fd;
    if ((fd = open(DEV_NAME, O_WRONLY)) == -1) {
        perror("[APP] Error occurs in device file open\n");
        return -1;
    }
    /* Trigger the driver */
    write(fd, NULL, 0);

    close(fd);
    return 0;
}
```

남은 일은 이 Application을 Board로 Module과 함께 보내 Stopwatch 기능을 검증하는 것뿐이다. 이렇게 해서 본 과제3 프로그래밍을 마무리한다.

맨 마지막 장에서 지금까지 설명한 Flow에 대한 Flow Chart를 첨부하였다.

IV. 연구 결과

지금까지 설명한 Stopwatch Driver는 정상적으로 동작하며 PDF 명세서의 요구 조건을 모두 충족하고 있다. FND Device를 요구사항대로 적시에 정확히 제어하고 있으며, Timer (시간측정)과 Button 입력(Interrupt) 처리도 정상적으로 동작하고 있다. 제출 Source Code의 모든 핵심 Function에는 Logging Print문이 포함되어 있기 때문에 printk가 Console창으로 출력을 뿌릴 수 있게 우선순위를 조정하고 Application을 동작시키면 Device 동작 과정을 minicom으로도 자세히 확인할 수 있다(과제 평가 시 조교님께서 이를 참조하시면 조금 더 편리한 평가를 하실 수 있습니다). 자세한 결과물은 직접 FPGA 보드 동작 및 Minicom을 통해 확인할 수 있다.

결론적으로 **과제 명세서에서 요구하는 모든 조건을 충족시켰다. 모두 잘 동작**하고 있다. 주요 평가 포인트는 다음과 같이 정리할 수 있다.

- Interrupt 사용 시 Top Half Interrupt와 Bottom Half Interrupt로 나누어 구현 (충족)

~> Top Half Routine과 Bottom Half Routine의 구분 기준 : 본 보고서에서 계속 'Internal Function'이라 칭하는 **'Timer 관련 작업을 호출해 Stopwatch Service를 제공하는 함수들'은 '빠르게 수행되어야 하는 Stopwatch의 핵심 기능'이다.** 따라서 이들은 모두 Top Half에 포함한다. 그리고 남은 각종 **Logging Print문들은 Stopwatch 동작 간에 핵심적인 기능은 아니다.** 따라서 본인은 이러한 Logging Print문들을 모두 Bottom Half화 하였다. 즉, *'Stopwatch Service의 핵심 기능이냐 아니냐'* 가 Half 분류 기준이며, 코드 레벨로는 'printk문을 호출하는 부분'이라고 할 수 있겠다.

- Module을 실행시키는 Application을 구현하고, Parameter는 받지 않음. (충족)

- Timer Device Driver 이름은 /dev/dev_driver로 하고 Major Number는 242이다. (충족)

- Key 입력은 Interrupt를 이용하여 수행 (충족)

- Home Button : Start -> 1초마다 FND의 정보를 갱신 (Timer 사용) (충족)

- Back Button : Pause -> 일시 정지 (충족)

~> 소수점 1번째 자리까지의 시간 유지도 충족. 본 프로그램에서 두 번째 자리까지 지원.

- VOL+ Button : Reset -> FND 출력 및 시간이 모두 초기 상태로 돌아감. (충족)

- VOL- Button : Stop -> 3초 이상 누르고 있을 시 Application을 종료, 그리고 FND를 0000으로 초기화 (충족)

- **Device Driver 이름은 /dev/stopwatch로 통일 (Major Number: 242)** (충족)

- **User Application은 Sleep 상태로 대기** (충족)

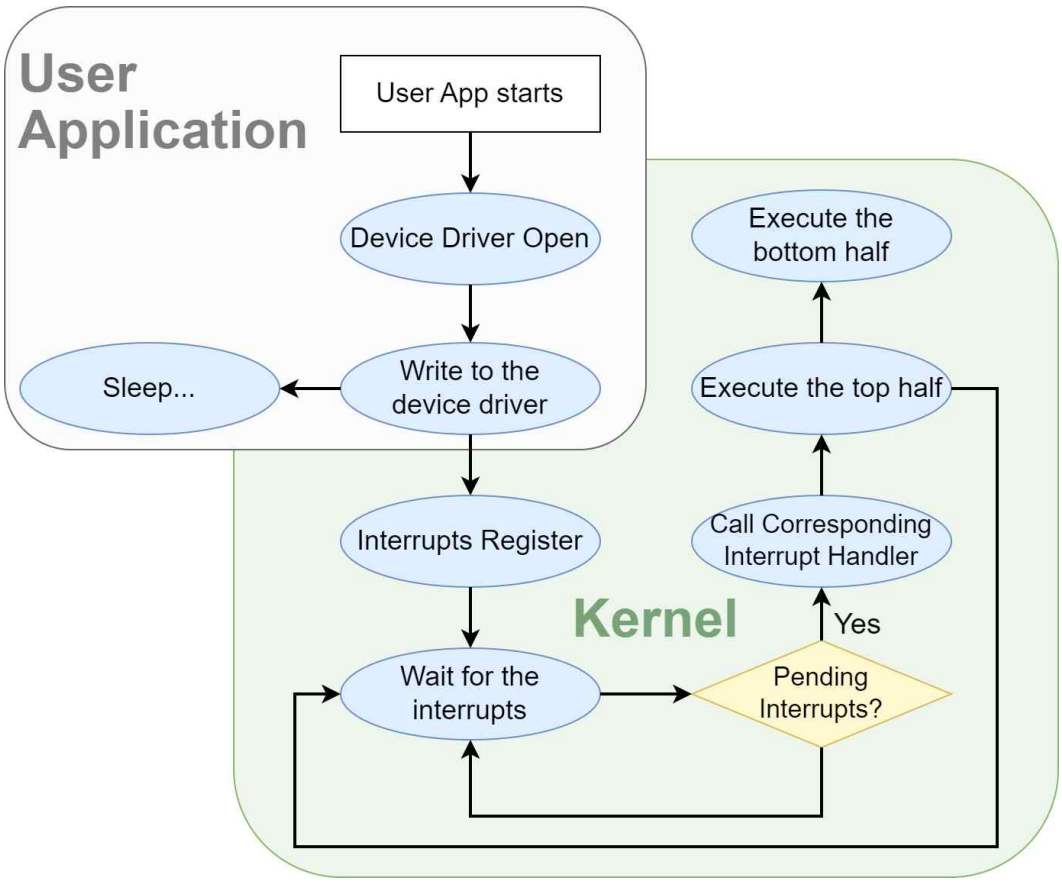
~> 구현 내용은 위 III 항목에서 소개하였음.

=> 이밖에도 명세서에 명시된 모든 조건 및 요구사항을 모두 충족

V. 기타

위와 같은 과정을 통해 과제3를 성공적으로 수행할 수 있었다. Device Driver Module을 만들어 Driver를 Kernel 내에 설치해 User Application에게 특정 Service를 제공하는 일련의 과정이 단순히 Embedded System 관점을 넘어 Linux Kernel 관리 및 운영 측면에서 굉장히 의미 있는 작업이었다고 생각한다. 다음 프로젝트에서 본 과제 수행 간에 경험한 기술들을 적극 활용해볼 수 있길 소망한다.

VI. Flow Chart



(Bottom Half 동작까지만 표현하였으며, 각 Button에 대한 Handler를 일일이 명시 x)