Embedded System Software 과제 2

(과제 수행 결과 보고서)

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

학번 및 이름: 20171643, 박준혁
개발기간: 2023. 05. 04. - 2023. 05. 17.

최 종 보 고 서

I. 개발 목표

본 임베디드시스템소프트웨어 과제2는 명세서에 주어진 대로 FPGA Board 상의 여러 I/O Device에 대해 **'시간, 반복 횟수, 계수 시작 Digit 입력'을 받아 다양한 입출력을 수행할 수 있도록 기능을 제공하는 Timer Device Driver 개발**이 목표이다. 즉, User-defined Operations를 수행하는 Device Driver의 개발이 주요 포인트이며, 이름 그대로 Timer를 활용해 입출력을 변화시킨다.

이를 위해 지난 과제1 및 이번 과제2에서 제공된 FPGA Board I/O Devices에 대한 Default Driver Module 코드를 활용해 본 과제에서 필요한 기능을 제공할 수 있도록 I/O 제어 루틴들을 취합한다. 이어, 취합된 루틴들을 호출하는 Timer Driver Module을 개발한다. 배부된 'The Linux Kernel Module Programming Guide' 교재에 나와 있는 Module 예제들을 토대로 Timer Driver의 틀을 만들고, 해당 Module에서 언급한 I/O 제어 루틴들을 호출해 기능을 제공한다. 이때, 본 과제2에선 Timer Interval, Timer Count, Timer Init를 입력받아 '시간에 따른 계수 반복'을 수행하므로 Interval과 Count를 통해 Timer를 세팅 및 Register해야 한다. 이 역시 본 과목 강의 시간에 학습한 내용을 토대로 어렵지 않게 수행할 수 있다.

한편 위와 같은 과정을 통해 만들어진 Timer 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 Timer Driver를 이용하는 Application을 별도로 만들어 이를 통해 Timer Driver를 검증**한다.

이런 흐름을 거쳐 수행된 본 과제2의 제출 코드는 PDF 명세서에서 요구하는 모든 조건을 만족하고 있다.

II. 개발 범위 및 내용

가. 개발 범위

본 항목에선 과제2 수행 간에 고려해야 할 개발 범위 및 내용에 대해 다룬다. 총 세 Phase로 나누었으며, 각각 **FPGA I/O Devices Control, Timer Device Module, Application for Verification**이다.

(1) FPGA I/O Devices Control

우선 배부된 Device Driver 코드를 기반으로 우리가 본 과제에서 사용할 FPGA I/O Devices(FND, LED, TEXT_LCD, DOT Device)의 I/O Controlling Routines와, 해당 Devices를 일괄적으로 한 Code 내에서 통제하기 위해 I/O Device Mapping 방법을 학습한다. 이어 해당 학습 내용을 토대로 FPGA I/O Devices 기능을 한 번에 제공할 수 있는 통일된 코드 파일을 작성할 것이며, 해당 작성 과정은 아래 '개발 내용'에서 소개한다.

(2) Timer Driver Module

이어, 본 과제의 핵심 요소인 Timer Driver를 Module 형태로 개발한다. Timer Driver는 (1)번 항목에서 언급한 FPGA I/O Devices Controlling Methods를 이용해 PDF 명세서에 언급된 개발 조건들을 만족시킨다.

한편, Driver 이름에서도 나와 있듯, 그러한 Controlling은 System 내에 등록된 Timer를 통해 이뤄지며, Driver로 입력된 Timer Interval, Timer Count, Timer Init(String)을 기반으로 Timer가 동작한다. 이를 위해 Module 내 특정 함수에서 Timer를 Register하고 제어하는 루틴이 필요할 것이다.

즉, **Timer Driver는 FPGA I/O Devices 통제 및 Timer 통제를 유기적으로 수행**할 것이며, 자세한 개발 과정은 이하 항목들에서 소개할 예정이다.

(3) Application for Verification

(1)과 (2)를 통해 Timer 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 **Timer Driver를 Open한 후 Driver 내 기능을 요청 및 검증하는 Application도 설계**한다.

Application이 할 일은 많지 않다. PDF 명세서를 따라 Interval, Count, Init String을 사용자로부터 입력받고 Timer Driver를 Open 후 Driver로 ioctl Command를 통해 Data Setting 및 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 I/O Devices를 제어하기 위한 개발 과정부터 소개할 것이며, Code-Level Implementation은 제외한다.

(1) FPGA I/O Devices Control

FPGA Board I/O Devices 통제 루틴 설계는 사실 간단하다. 배부된 예시 Drivers를 분석해보면 어렵지 않게 Device Control 방법을 알 수 있는데, 간단히 요약하면 다음과 같이 두 단계로 나눌 수 있다.

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.

이게 전부다. 요약하면, "ioremap으로 연결하고, outw로 출력한다"이다. 따라서 실제로 본인은 해당 Principle을 따라 각 I/O Devices(FND, LED, TEXT_LCD, DOT)의 통제 루틴을 따로따로 마련한 후 'fpga_device.c' 파일에 모아놓는다. 'fpga_device.c'에 정의된 루틴들은 Timer Driver에서 호출할 예정이다.

이와 별개로, 각 '연결', '출력' 루틴 내부에 printk 함수를 통한 Logging도 마련하면 좋을 것이다.

(2) Timer Driver Module

Timer Driver Module 구축은 앞서 언급한 것처럼 'The Linux Kernel Module Progra--mming Guide'에 나와 있는 예시 Module의 형태를 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, ioctl)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 I/O Devices Mapping / Un-mapping 루틴도 추가하도록 하자. open, close, ioctl 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 Functions**도 정의하자. Internal Function이라 함은, 본 과제2에서 요구하는 I/O 조건, Flow를 수행하는 함수들을 의미하며, Timer Instance 초기화, Timer Structure에 Timer Instance 등록, 일괄 I/O Printings, I/O Data Updates 등을 수행한다.

상기한 것처럼, 본 항목의 자세한 Code-Level Details 및 추가 설명은 이하 항목에서 소개할 예정이다.

(3) Application for Verification

완성된 Timer 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.

- 05/04~05/07: 과제 이해 및 사전 학습
- 05/07~05/12: Module & Application 프로그래밍 및 검증
- 05/12~05/17: 개발 완료 및 보고서 작성

지난 과제1에 비해 난이도가 높지 않았다고 느껴지며, 약 일주일의 개발을 통해 과제를 완성하였다. 이어 아래 '개발 방법' 항목에서 본 과제 제출 코드에 대한 Code-Level 설명을 이어간다.

나. 개발 방법

지난 과제1과 마찬가지로 Cross-Compile 환경에서 C Language를 이용해 Module과 Application을 개발 PC에서 작성하였으며, 작성된 Module과 Application을 순차적으로 소개한다. **'timer_driver.c'는 Module을 구성하며, 'fpga_device.c'는 FPGA I/O Device 제어 루틴을, 'timer.c'는 Application Program을 정의**한다.

(1) FPGA I/O Devices Control (fpga_device.c)

우선 'fpga_device.c'에서 사용할 **FPGA I/O Devices 관련 Metadata를 정의**한다. Meta-data에는 ioremap에 Parameter로 넘어갈 Bus Address와 Buffer Size부터, DOT에 Digit를 표시하기 위한 고정 배열, 표시할 Digit 개수 등 다양하다. 아래를 보자.

```c
/* # of FPGA I/O devices */
#define FPGA_DEV_CNT 4

/* Enum type for FPGA I/O devices */
typedef enum _FPGA_DEV {
    FND, LED, TEXT_LCD, DOT
} FPGA;

/* Output buffer size of TEXT_LCD */
#define TEXT_LCD_BUF_SIZE 32

/* # of dots in DOT device */
#define DOT_NUM 9

/* # of digits */
#define DIGIT_NUM 8

/* Error identifier */
#define FPGA_ERROR -1
```

```c
/* Metadata of FPGA I/O devices */
char *fpga_addr[FPGA_DEV_CNT];
static const int FPGA_DEV_ADDR[] = { 0x08000004, 0x08000016, 0x08000090, 0x08000210 };
static const int FPGA_DEV_DATA_SIZE[] = { 0x2, 0x1, 0x32, 0xA };

/* Fixed dot configurations for printing 1 ~ 8 and blank */
static const unsigned char dot_digit[DIGIT_NUM + 1][DOT_NUM + 1] = {
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},// ' '
    {0x0c, 0x1c, 0x1c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x1e},// '1'
    {0x7e, 0x7f, 0x03, 0x03, 0x3f, 0x7e, 0x60, 0x60, 0x7f, 0x7f},// '2'
    {0xfe, 0x7f, 0x03, 0x03, 0x7f, 0x7f, 0x03, 0x03, 0x7f, 0x7e},// '3'
    {0x66, 0x66, 0x66, 0x66, 0x66, 0x66, 0x7f, 0x7f, 0x06, 0x06},// '4'
    {0x7f, 0x7f, 0x60, 0x60, 0x7e, 0x7f, 0x03, 0x03, 0x7f, 0x7e},// '5'
    {0x60, 0x60, 0x60, 0x60, 0x7e, 0x7f, 0x63, 0x63, 0x7f, 0x3e},// '6'
    {0x7f, 0x7f, 0x63, 0x63, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03},// '7'
    {0x3e, 0x7f, 0x63, 0x63, 0x7f, 0x7f, 0x63, 0x63, 0x7f, 0x3e},// '8'
};
```

그렇게 정의한 Metadata를 이용해 우리가 본 과제에서 사용할 모든 I/O Device를 일괄적으로 Mapping하고 Un-map하는 함수를 다음과 같이 마련하자. 이들은 각각 Timer Driver의 시작과 종료 시에 호출될 것이다.

```c
/* Map all the I/O devices provided by our FPGA board */
int fpga_map_devices(void) {
    FPGA dev;
    printk("[FPGA DEVICE] Open all the necessary I/O devices\n");

    for (dev = FND; dev <= DOT; dev++) {
        fpga_addr[dev] = ioremap(FPGA_DEV_ADDR[dev], FPGA_DEV_DATA_SIZE[dev]);
        if (fpga_addr[dev] == NULL) {
            printk("[FPGA DEVICE] I/O Device mapping fails\n");
            return FPGA_ERROR;
        }
    }

    return 0;
}

/* Unmap all the I/O devices provided by our FPGA board */
void fpga_unmap_devices(void) {
    FPGA dev;
    printk("[FPGA DEVICE] Close all the open I/O devices\n");

    for (dev = FND; dev <= DOT; dev++)
        iounmap(fpga_addr[dev]);
}
```

이어, 배부된 예시 Device Driver와 같이 outw함수를 이용해 Write를 수행하는 루틴을 간단하게 정의한다. FND, LED, TEXT_LCD, DOT에 대해 각각 만들며, 우측은 FND, LED Device에 대한 Write 루틴이다. 굉장히 간단한 구조를 가짐을 주목하자.

```c
/* Print the digit to FND device */
void fpga_fnd_write(const int idx, const int digit) {
    unsigned short value = digit << (12 - 4*idx);
    outw(value, (unsigned int)fpga_addr[FND]);
    printk("[FPGA DEVICE] FND WRITE: %d (idx: %d)\n", digit, idx);
}

/* Print the digit to LED device */
void fpga_led_write(const int digit) {
    unsigned short value = ((digit > 0) ? (1 << (8 - digit)) : 0);
    outw(value, (unsigned int)fpga_addr[LED]);
    printk("[FPGA DEVICE] LED WRITE: %d (LED%d)\n", digit, digit);
}
```

하지만 TEXT_LCD에 대한 Write Routine은 한 가지 생각해볼 지점이 있다. 본 과제2 명세서를 보면 TEXT_LCD Device의 화면을 두 줄로 구분하고, 윗줄엔 과제 수행자의 학번을, 아랫줄엔 과제 수행자의 영문 이름을 명시해야 한다. 그 와중에서 윗줄과 아랫줄이 서로 독립적으로 해당 줄 내에서 좌우로 움직이며 한쪽 벽에 닿을 때만 방향을 바꾸는 식으로 동작한다. 이를 위해선 우린 **TEXT_LCD에 대한 Write을 수행하기 전에 PDF 조건을 만족시키도록 문자열 처리를 해주어야 한다**. 다음과 같이 말이다. 각 줄에서 '현재 이동 위치'를 고려해 앞 뒤로 Padding 처리를 하고 있음을 주목하자.

```c
/* Print two text lines to TEXT_LCD device */
void fpga_text_lcd_write(const char *text1, const int text1_prev_idx, const char *text2, const int text2_prev_idx) {
    unsigned short value;
    unsigned char text_buf[TEXT_LCD_BUF_SIZE + 1] = {0};
    int i, text1_len, text1_idx, text2_len, text2_idx;
    text1_len = strlen(text1); text2_len = strlen(text2);
    text1_idx = text1_prev_idx; text2_idx = TEXT_LCD_BUF_SIZE / 2 + text2_prev_idx;

    /* Text at the first line */
    // Fill the front remainders of first line with spaces
    memset(text_buf, ' ', text1_idx);
    strncat(text_buf, text1, text1_len);
    printk("[FPGA DEVICE] TEXT_LCD WRITE: \"%-16s\" (line1)\n", text_buf);

    /* Text at the second line */
    // Fill the remainders of first line with spaces
    memset(text_buf + text1_len + text1_idx, ' ', text2_idx - (text1_len + text1_idx));
    strncat(text_buf, text2, text2_len);
    // Fill the remainders of second line with spaces
    memset(text_buf + text2_len + text2_idx, ' ', TEXT_LCD_BUF_SIZE - (text2_len + text2_idx));
    text_buf[TEXT_LCD_BUF_SIZE] = '\0';
    printk("[FPGA DEVICE] TEXT_LCD WRITE: \"%-16s\" (line2)\n", text_buf);

    /* Write to buffer */
    for (i = 0; i < TEXT_LCD_BUF_SIZE; i += 2) {
        value = (text_buf[i] & 0xFF) << DIGIT_NUM | (text_buf[i + 1] & 0xFF);
        outw(value, (unsigned int)fpga_addr[TEXT_LCD] + i);
    }
}
```

이렇게 해서 FPGA I/O Device Control Routine을 간단하게 구축할 수 있다. (2)번의 Timer Driver Module에선 이를 호출하여 I/O Devices를 조정할 것이다.

(2) Timer Driver Module (timer_driver.c)

이번엔 Timer Driver Module을 설계하자. 우선 아래와 같이 **Timer Driver에서 사용할 Metadata를 'timer_driver.h'에 Structure로 정의**하자. 사용자로부터 입력받은 Interval, Count, Init String뿐만 아니라 Text_LCD Device에서 출력할 두 Line String의 실시간 상황, 그리고 FND Device의 Counting 효과를 보여주는 데에 쓰일 두 정수 변수들도 있다. 무엇보다도, 해당 구조체 내부에 Timer Instance가 정의되어 있음을 주목하자.

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

    /* Metadata (related to timer) */
    int interval;        // Interval for countering
    int max_cnt;         // Max limit of counts
    int cnt;             // Total # of counts

    /* Metadata (related to text_lcd) */
    char text[LINE_NUM][TEXT_LEN + 1];// Two texts (for each line)
    short text_idx[LINE_NUM];     // Indexes to start printing each line
    int text_dir[LINE_NUM];      // Direction of forwarding of each text
    // 0 means right, and 1 means left

    /* Metadata (related to fnd & dot) */
    short digit;         // Current digit
    short digit_idx;     // Current digit's index (for fnd)
} Timer;
```

이어 'timer_driver.c' 파일을 보자. 코드의 시작부엔 다음과 같이 Module Function 선언부터 File-related System Call 대체 루틴, Internal Functions 선언, Concurrent Open을 Blocking할 Wait Queue, ioctl을 위한 Command 선언 등이 이뤄지고 있음을 주목하자.

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

/* Case definitions of user-defined ioctl() handler */
#define IOCTL_SET_OPTION _IOW(DEV_MAJOR_NUM, 0, char*)
#define IOCTL_COMMAND _IO(DEV_MAJOR_NUM, 0)

/* Current timer instance */
static Timer cur_timer;

/* Core routines of this module */
static int __init timer_driver_init(void);
static void __exit timer_driver_exit(void);
static int timer_driver_open(struct inode *, struct file *);
static int timer_driver_close(struct inode *, struct file *);
static long timer_driver_ioctl(struct file *, unsigned int, unsigned long);

/* Redirect file I/O functions to user-defined handlers */
static struct file_operations timer_driver_fops = {
    .owner = THIS_MODULE,
    .open = timer_driver_open,
    .release = timer_driver_close,
    .unlocked_ioctl = timer_driver_ioctl
};

/* Internal functions to manage metadata attached to timer and perform operations */
static void _timer_init_data(const int, const int, const char *);
static void _timer_register(Timer *tm);
static void _timer_print(const Timer*);
static void _timer_start(void);
static void _timer_handler(unsigned long);

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

/* Two text lines for TEXT_LCD */
static char *student_id = "2017643";
static char *student_name = "Junhyeok Park";
```

아래를 보자. Module Init 함수이다. Character Device Register, FPGA I/O Devices Mapping, Timer Install 등의 행위가 이뤄지고 있음을 주목하자.

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

    /* Map all the I/O devices in our FPGA board */
    if (fpga_map_devices() == FPGA_ERROR)
        return FPGA_ERROR;
    /* Be prepared to use timer */
    init_timer(&(cur_timer.timer));

    return ret_val;
}
```

마찬가지로 Module Exit 함수를 Convention에 맞게 구축할 수 있다. 자, 이제 이어서 Open 대체 함수를 보자. **최근 수업 내용이었던 '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.

```
/* open() Handler for this driver */
static int timer_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("[TIMER DRIVER] %s called\n", __func__);
    return 0;
}
```

이에 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 timer_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("[TIMER DRIVER] %s called\n", __func__);
    return 0;
}
```

다음은 IOCTL 대체 함수이다. **User Application에서 ioctl 호출 시 어떤 Option을 넘겼느냐에 따라 Case를 구분해 일을 처리하고 있음**을 주목하자. SET_OPTION Flag의 경우엔 주어진 사용자 입력에 따라 Timer Metadata Intialization을, COMMAND Flag의 경우엔 Timer를 작동시켜 Service를 제공하고 있음을 어렵지 않게 포착할 수 있다.

```c
/* ioctl() Handler for this driver */
static long timer_driver_ioctl(
    struct file *file, unsigned int ioctl_num, unsigned long ioctl_param
) {
    int ret_val = 0;
    char *user_ptr;
    char read_buf[READ_BUF_SIZE + 1];
    char temp_buf[TEMP_BUF_SIZE] = {'\0'};
    char fnd_init[TEMP_BUF_SIZE + 1] = {'\0'}; long interval, cnt;

    /* Select the operation */
    switch (ioctl_num) {
    /* Case1: read all the parameters when the program starts */
    case IOCTL_SET_OPTION:
        printk("[TIMER DRIVER] %s called (SET_OPTION)\n", __func__);

        /* Get string data from the user application */
        user_ptr = (char*)ioctl_param;
        if (strncpy_from_user(read_buf, user_ptr, strlen_user(user_ptr)) >= 0) {
            /* Retrieve each variable from the read string */
            // First three characters indicate interval value
            strncpy(temp_buf, read_buf, 3);
            kstrtol(temp_buf, READ_BUF_SIZE, &interval);  // make int
            // Following three characters indicate count value
            strncpy(temp_buf, read_buf + 3, 3);
            kstrtol(temp_buf, READ_BUF_SIZE, &cnt);
            // Final four characters indicate the initial FND
            strncpy(fnd_init, read_buf + 6, TEMP_BUF_SIZE);

            /* Set metadata of timer as input */
            _timer_init_data(interval, cnt, fnd_init);
        }
        else ret_val = -EFAULT;
        break;
    /* Case2: Start the timer */
    case IOCTL_COMMAND:
        printk("[TIMER DRIVER] %s called (COMMAND)\n", __func__);
        _timer_start();
        break;
    /* Other cases */
    default:
        printk("[TIMER DRIVER] Undefined IOCTL command (%u)\n", ioctl_num);
        ret_val = -ENOTTY;
        break;
    }

    return ret_val;
}
```

지금부터 Timer Driver에서 '실질적인 Service 제공'을 담당하는 Internal Functions에 대해 알아보자. Timer Metadata 초기화, Timer Instance 등록, FPGA I/O Devices 출력, Timer 만료 시 수행되는 Handler 등의 다양한 작업이 수행된다. 우선 아래의 Timer Met--adata 초기화 및 Timer 등록 함수들부터 확인하자.

```c
/* Internal Handlers Definitions */
/* Initialize metadata of our timer as given parameters */
static void _timer_init_data(const int interval, const int cnt, const char *init) {
    int i;
    printk("[TIMER_DRIVER} Inputs: (%d, %d, %s)\n", interval, cnt, init);

    /* Initialize data */
    for (i = 0; i < TEMP_BUF_SIZE; i++) {
        if (init[i] != '0') {
            cur_timer.digit = init[i] - '0';
            cur_timer.digit_idx = i;
            break;
        }
    }
    strncpy(cur_timer.text[0], student_id, strlen(student_id));
    strncpy(cur_timer.text[1], student_name, strlen(student_name));
    cur_timer.text_idx[0] = cur_timer.text_idx[1] = 0;
    cur_timer.text_dir[0] = cur_timer.text_dir[1] = 0;

    cur_timer.interval = interval;
    cur_timer.max_cnt = cnt;
    cur_timer.cnt = 0;
}

/* Register the timer structure to the timer list */
static void _timer_register(Timer *tm) {
    printk("[TIMER_DRIVER] Timer is registered\n");

    /* Set the timer */
    tm->timer.expires = get_jiffies_64() + tm->interval * (HZ / 10);
    tm->timer.data = (unsigned long)&cur_timer;
    tm->timer.function = _timer_handler;   // This will be called for every expiration

    /* Register to the list */
    add_timer(&(tm->timer));
}
```

Timer 등록 함수에선 주어진 Interval을 기준으로 Timer 시간을 설정하고, 만료 시 _timer_handler 함수가 돌아가도록 설정하고 있음을 주목하자.

아래는 Timer Driver가 I/O Device 출력을 수행하는 함수인 '_timer_print', 그리고 Timer Driver의 Service를 수행하는 함수인 '_timer_start'이다.

```c
/* Print metadata to the FPGA I/O devices */
static void _timer_print(const Timer *tm) {
    printk("[TIMER_DRIVER] Timer performs WRITEs\n");

    fpga_fnd_write(tm->digit_idx, tm->digit);
    fpga_led_write(tm->digit);
    fpga_text_lcd_write(tm->text[0], tm->text_idx[0], tm->text[1], tm->text_idx[1]);
    fpga_dot_write(tm->digit);
}

/* Start the timer, with resetting the device */
static void _timer_start(void) {
    printk("[TIMER_DRIVER] Timer starts now\n");

    /* Reset */
    del_timer_sync(&(cur_timer.timer));
    _timer_print(&cur_timer);     // Print the reset output
    _timer_register(&cur_timer);  // Register the timer
}
```

대망의 '_timer_handler' 함수를 보자. Timer Structure에 등록된 Timer가 세팅된 시간이 끝나 만료되었을 때 호출되는 루틴이다.

```
/* Callback handler function that is expected to be called whenever the timer expires */
static void _timer_handler(unsigned long timeout) {
    int i;
    Timer *tm = (Timer*)timeout;
    printk("[TIMER_DRIVER] Timer Handler is called (cnt=%d)\n", ++(tm->cnt));

    /* If the counter reaches the maximum count limit, then clear all */
    if (tm->cnt == tm->max_cnt) {
        tm->text[0][0] = '\0';
        tm->text[1][0] = '\0';
        tm->digit = 0;
    }
    /* Otherwise change the outputs of I/O devices in our FPGA board */
    else {
        // Advance in a circular manner
        if (tm->digit == ROTATE_CNT)
            tm->digit = 1;
        else tm->digit++;
        if ((tm->cnt % ROTATE_CNT) == 0) {
            tm->digit_idx++;
            tm->digit_idx %= 4;
        }

        // Update texts of TEXT_LCD, following conditions mentioned in PDF
        for (i = 0; i <= 1; i++) {      // 'i' means which line is considered
            if (tm->text_dir[i] == 0) {
                tm->text_idx[i]++;
                if (tm->text_idx[i] + strlen(tm->text[i]) > TEXT_LCD_BUF_SIZE / 2) {
                    tm->text_idx[i] -= 2;
                    tm->text_dir[i] = 1;
                }
            }
            else {
                tm->text_idx[i]--;
                if (tm->text_idx[i] < 0) {
                    tm->text_idx[i] += 2;
                    tm->text_dir[i] = 0;
                }
            }
        }

        // Register the next timer
        _timer_register(tm);
    }

    /* Print the updates */
    _timer_print(tm);
}
```

여기서 중요한 지점은 딱 하나이다. Timer Driver는 **'주기적으로 Counting하는 Service'**를 제공하는데, 그 Counting Service는 Interval을 기준으로 매번 Timer가 만료되어 가며 이뤄지는 것이다. 이를 위해선 Timer를 매 Interval 만료 시 새롭게 다시 설치해야 한다는 것이다. 즉, Timer가 연쇄적으로 다음과 같이 동작하는 것이다.

"Timer 설치 -> Timer 만료 -> I/O Updates -> Timer 설치 -> ..."

↓

밑줄 친 부분이 바로 Interval이 되는 것이다.

그렇다면, 이러한 Timer 설치는 언제 멈추는가? 그렇다. 입력 받은 Count 횟수가 다 차면 멈추는 것이다. 그 와중에 Count가 끝나기 전까진 계속해서 FND Device와 TEXT_LCD Device의 출력물을 Update(Modify)하고 있음도 잊지 말자.

이렇게 해서 Timer Driver Module 구축도 완료된다. 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 (timer.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, User Application에선 해당 파일명을 Driver 이름으로 해서 Open해 Timer Driver를 사용**할 수 있다. 그렇게 Open하고 나면 ioctl 함수를 이용해 Timer Setting과 Timer Driver Service 요청을 PDF 명세서대로 수행하면 된다. Application은 이게 핵심이고 전부이다.

```c
/* Open timer device */
if ((fd = open(dev_file_addr, O_WRONLY)) == -1) {
    perror("Error opening device file\n");
    return -1;
}

/* Set parameters and then pass them */
sprintf(ioctl_param, "%03d%03d%04d", timerInterval, timerCount, timerInit);
ioctl(fd, IOCTL_SET_OPTION, ioctl_param);  // Pass option parameters to device
ioctl(fd, IOCTL_COMMAND);                  // Start the timer device!

/* Close the device */
close(fd);
```

남은 일은 사용자로부터 Interval, Count, Init String을 입력받아 입력을 검증하는 것뿐이고, 이는 크게 중요한 부분은 아니기에 설명을 생략한다.

이렇게 해서 본 과제2 프로그래밍을 마무리한다.

맨 마지막 장에서 지금까지 설명한 Flow에 대한 Flow Chart를 첨부하였다.

Ⅳ. 연구 결과

지금까지 설명한 Timer Driver는 정상적으로 동작하며 PDF 명세서의 요구 조건을 모두 충족하고 있다. FND, LED, TEXT_LCD, DOT Device를 모두 요구사항대로 적시에 정확히 제어하고 있으며, Timer도 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.

- 두 개의 ioctl 명령어 사용 (충족)

~> ioctl(fd, SET_OPTION, ...), ioctl(fd, COMMAND, ...) 형식을 모두 만족

- ioctl을 통하여 전달된 옵션을 기준으로 4가지 Device(FND, LED, TEXT_LCD, DOT)의 요구사항에 따라 Device에 동시에 출력 (충족)

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

- FND Device 출력 조건 with Rotation을 모두 정상적으로 지원 (충족)

~> PDF 명세서에 주어진 세 가지 Example과 정확히 동일한 순서로 FND 출력 동작

- TIMER_CNT 횟수만큼의 출력이 끝나면 fpga_led의 불을 꺼준다. (충족)

- fpga_dot도 fpga_fnd와 같은 문양으로 함께 바뀐다. (충족)

- TEXT_LCD 초기 상태 조건 (첫 번째 줄은 학번, 두 번째 줄은 영문명, 두 줄 모두 Left Aligned) 모두 만족 (충족)

- TEXT_LCD에서 텍스트가 없는 나머지 칸은 빈칸이 출력되도록 한다. (충족)

- TEXT_LCD에서 두 줄은 독립적으로 좌우 이동을 한다. 한쪽 끝에 닿을 때 방향이 바뀌는 방식으로 동작한다. (충족)

- TIMER_CNT 횟수만큼의 출력이 끝나면 TEXT_LCD의 불을 꺼준다. (충족)

=> 이밖에도 명세서에 명시된 모든 조건 및 요구사항을 모두 충족

V. 기타

위와 같은 과정을 통해 과제2를 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. 향후 출제될 과제들도 기다려진다. 남은 학기 계속해서 실습과 과제에 열심히 임하겠다.

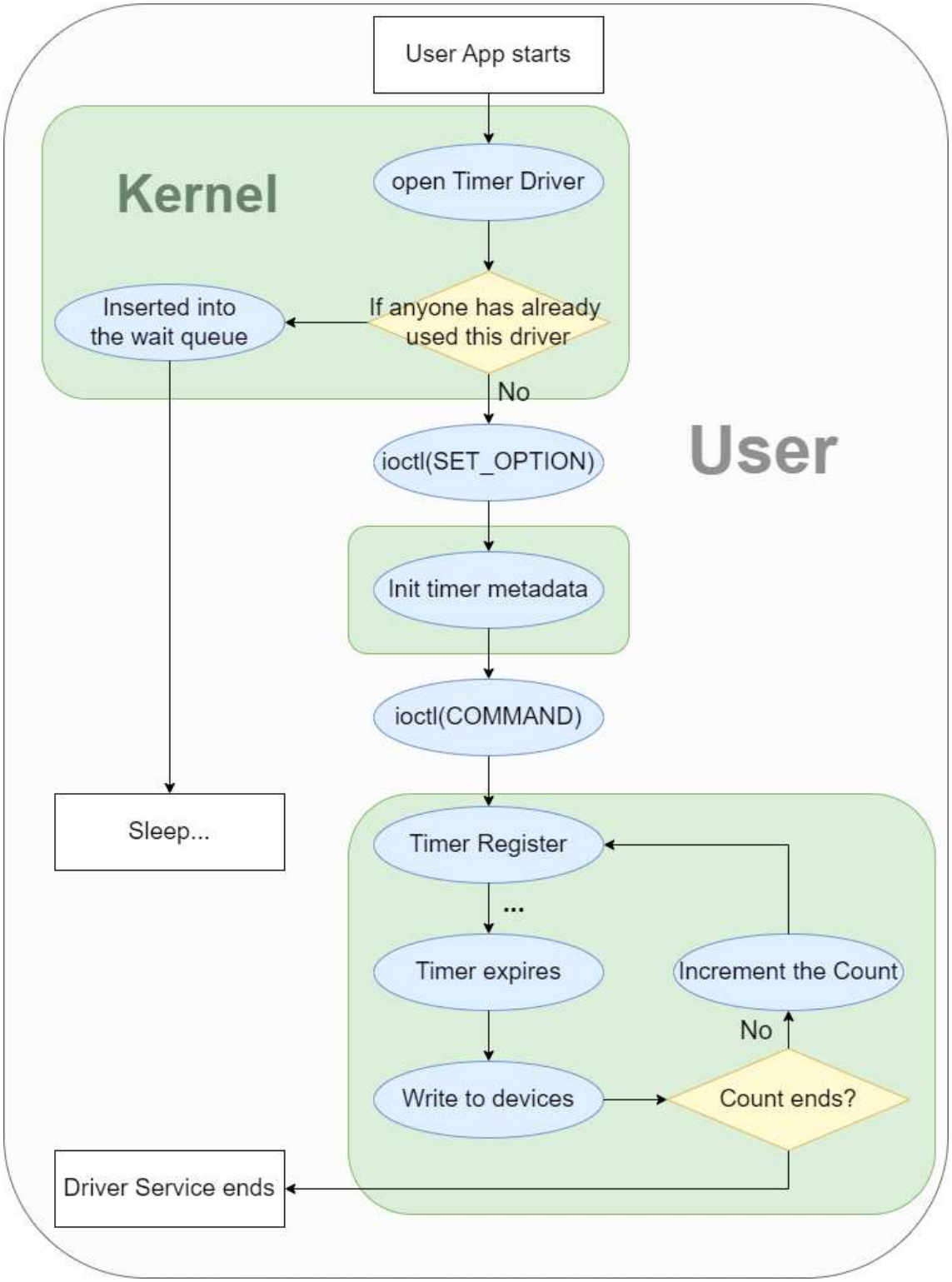

Figure 1 - Flow Chart of the communcation between Timer Driver and Application