Embedded System Software 과제 2

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20211584 장준영

개발기간: 2024. 5. 9. - 2024. 5. 14.

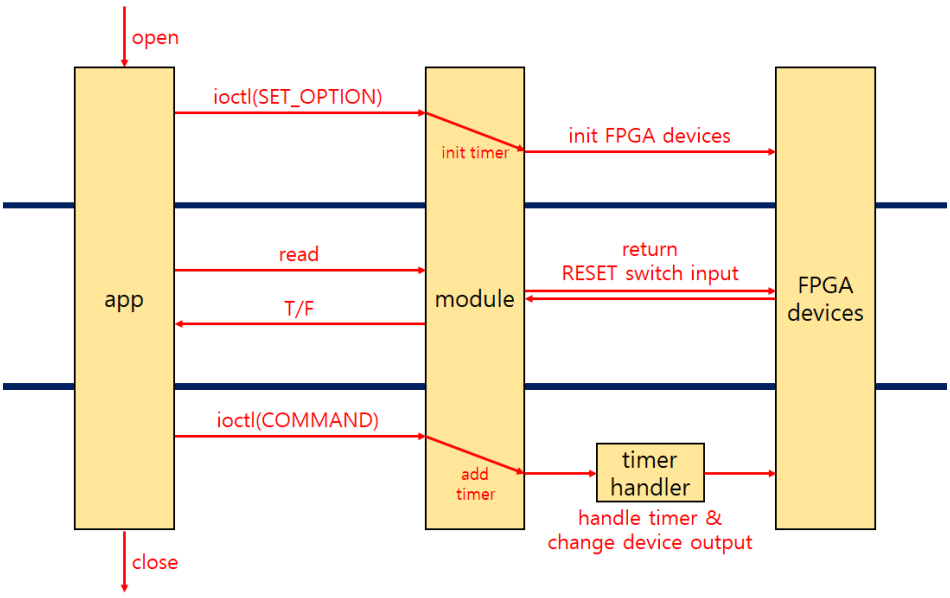
최 종 보 고 서

I. 개발 목표

이번 프로젝트에서는 FPGA 디바이스와 함께 운용되는 타이머 디바이스 드라이버를 구현하고, 해당 디바이스 드라이버를 테스트할 수 있는 유저 어플리케이션을 개발한다. 타이머 디바이스 드라이버에서는 타이머 자료구조를 관리하면서 디바이스 파일에 행해지는 file operation을 적절하게 처리하는 모듈을 구현해야 하고, 어플리케이션에서는 해당 모듈이 설계된 대로 적절히 디바이스 파일에 file operation을 호출해서 인터페이스를 통해 타이머를 이용하면 된다. 핵심적으로 ioctl이나 read같은 file operation 처리, timer_list 자료구조 활용, FPGA 디바이스의 address 매핑 등이 중요하게 이용된다.

II. 개발 범위 및 내용

가. 개발 범위



이번 프로젝트의 구조도는 위와 같다. app은 디바이스 파일에 대해 (1)open, (2)ioctl(SET_OPTION), (3)read, (4)ioctl(COMMAND), (5)close operation을 수행한다. module은 각각의 file operation에 대해 적절한 처리를 수행한다. 먼저 디바이스 파일이 open 되면, 사용되고 있다는 플래그를 설정하고 usage를 증가시킨다. 이후 사용자가 app으로 입력한 interval, count, init 등의 초기값을 ioctl(SET_OPTION)을 통해 module로 전송한다. module에선 이 값을 바탕으로 타이머와 FPGA 디스플레이를 초기화하고, 대기 상태가 된다. module에서 read operation은 RESET 스위치가 눌렸는지를 return 하기 때문에, app에서는 해당 결과값이 참이 될 때까지 계속해서 read를 호출한다. 이후 read를 통해 RESET 스위치가 눌린 것이 확인되면, ioctl(COMMAND)를 통해 타이머를 시작하라고 module에 전달한다. module은 타이머를 등록하고, 이후부터는 주기마다 호출되는 타이머 핸들러가 처리를 담당한다. 타이머가 종료되면, close(release)를 통해 사용 플래그를 해제하고 usage를 감소시킨다.

나. 개발 내용

[1] 타이머 모듈 프로그래밍

수업 시간동에 몇 주 동안 걸쳐 학습한 모듈 프로그래밍을 사용하여 타이머 디바이스 드라이버를 작성해야 한다. 리눅스 커널의 모듈은 시스템에 미리 설치된 커널 바이너리 이미지와는 별개로, 동적으로 연결하여(insmod, rmmod) 기능을 추가하거나 제거할 수 있도록 만든 오브젝트 파일이다. 커널 차원의 동작을 추가 및 제거하기 위해서 커널 이미지를 다시 컴파일하고 설치할 필요가 없다는 장점이 있다. 모듈은 다양한 동작을 수행할 수 있지만, 학습한 것과 프로젝트에 사용되는 내용은 캐릭터 디바이스 드라이버 구현이다. 디바이스 드라이버를 구현하기 위해선 디바이스 파일이 필요하고, 모듈에선 해당 파일에 수행되는 file operation을 원하는 동작(디바이스를 관리하는 동작)으로 대체한다. 이번 프로젝트에선 open, close, read, ioctl operation을 앞서 개발 범위에서 설명한 동작으로 대체한다. 특히 ioctl은 두 가지 다른 동작을 모듈 내에서 구별할 수 있도록 독립된 ioctl number를 설정하

고 사용한다는 점을 유념해야 한다.

또, 커널 타이머를 사용해 타이머를 관리해야 한다. 리눅스 커널의 struct timer_list는 커널 타이머를 관리하는데 사용되는 구조체로, 내부 주요 필드로 expires, data, function이 존재한다. expires는 타이머가 언제 종료될 지 결정하는 필드로, 커널 내부 시간 정보인 jiffies를 통해 설정한다. data는 타이머 핸들러로 전달될 값이다. function은 타이머 만료 시에 호출할 타이머 핸들러 함수를 지정한다. 이러한 값을 설정하고 타이머 관리 함수(init_timer, add_timer, del_timer, ...)를 사용하여 커널에 타이머 관리를 위탁할 수 있다. 이번 프로젝트에선 유저가 지정한 interval마다 타이머 핸들러가 호출되도록, 또 지정한 cnt만큼만 호출되고 타이머가 종료될 수 있도록 설정하면 된다. 문양이 회전하는 것 또한 핸들러 내에서 처리해야 한다.

[2] FPGA device 관리

이번 프로젝트에선 전처럼 구현된 디바이스 드라이버 모듈을 사용하는 것이 아니라, 타이머 디바이스 드라이버 모듈에서 사용하는 모든 FPGA 디바이스를 관리할 수 있도록 새로 구현해야 한다. 이 구현 과정은 제공된 디바이스 드라이버 코드를 참고하면 쉽게 알 수 있다. FPGA 디바이스의 physical address를 알고, 이를 커널 프로세스의 virtual address와 연결하는 매핑 과정과 I/O 포트로 데이터를 읽고 쓰는 과정을 거친다. 주소 매핑/언매핑은 각각 ioremap/iounmap 함수를 사용하면 되고, 데이터 읽고 쓰기는 inw, outw 함수를 사용하면 된다. 주소 공간에 입력하는 크기나 방법은 제공된 코드를 꼼꼼히 읽고 그대로 따라 구현한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

2024. 5. 9. ~ 2024. 5. 10.	본격적인 모듈 프로그래밍에 앞서, 모듈을 호출하는 app(.c) 프로그램을 먼저 구현하였다. 모듈에 실행할 file operation이 각각 어떤 역할을 할지 구상

	하는데 도움이 되었다.
2024. 5. 11.	device.c 코드에 FPGA 디바이스를 관리하는 함수들을 작성하였다. 첨부된 디바이스 드라이버 코드를 참고하여, 주소를 매핑하고 값을 입출력하는 함수를 작성했다. 특히 저번 과제에서 LCD 드라이버를 호출할 때 가독성이 떨어지는 코드를 작성한 것이 아쉬워서, LCD 디스플레이를 세 부분으로 나눠 컴팩트하게 작성하기 위해 노력했다.
2024. 5. 12. ~ 2024. 5. 13.	timer.c 에서 실질적인 타이머 디바이스 드라이버를 구현하였다. file operation에 따른 타이머 처리와 FPGA 디바이스 인터페이스 호출이 주요 구현이다.
2024. 5. 14.	디버깅을 마친 완성된 코드에서 디버깅 루틴을 지우고, 세밀하게 주석을 작성하였다. 또, 직접 작성한 모듈 Makefile에 오류가 많아 자료를 참고하여 수정하였다.

나. 개발 방법

[1] app

유저 레벨에서 단순히 디바이스 파일을 통해 모듈에 명령어를 전달하는 프로그램이다. 유저의 입력을 예외처리하고, 때에 따라 필요한 file operation을 호출할 수 있도록 구현한다.

```c
/* ARGUMENT COUNT VALIDATION ROUTINE */
if (argc != 4)
{
    printf("usage: ./app TIMER_INTERVAL[1-100] TIMER_CNT[1-200] TIMER_INIT[0001-8000]\n");
    return -1;
}

/* ARGUMENT VARIABLES VALIDATION ROUTINE */
int i;
int interval = atoi(argv[1]),
    cnt = atoi(argv[2]),
    init = atoi(argv[3]);
if (interval < 1 || interval > 100)
{
    printf("ERROR(app.c) : TIMER_INTERVAL[1-100] out of range\n");
    return -1;
}
if (cnt < 1 || cnt > 200)
{
    printf("ERROR(app.c) : TIMER_CNT[1-200] out of range\n");
    return -1;
}
```

가장 먼저 매개변수의 개수가 알맞게 입력되었는지를 체크한다. 이후, 입력된 세 개의 인자(TIMER_INTERVAL, TIMER_CNT, TIMER_INIT)에 대해 예외처리를 진행한다. 먼저, TIMER_INTERVAL과 TIMER_CNT는 값의 범위가 정해져 있기 때문에 그 범위를 벗어나면 프

로그램을 종료한다.

```
char nonzero = 0;
for (i = 0; i < 4; i++)
{
    if (argv[3][i] == '\0')
    {
        printf("ERROR(app.c) : TIMER_INIT[0001-8000] must be four characters\n");
        return -1;
    }
    if (argv[3][i] < '0' || argv[3][i] > '8')
    {
        printf("ERROR(app.c) : TIMER_INIT[0001-8000] out of range\n");
        return -1;
    }
    if (argv[3][i] > '0')
    {
        if (nonzero)
        {
            printf("ERROR(app.c) : TIMER_INIT[0001-8000] multiple non-zero characters\n");
            return -1;
        }
        nonzero++;
    }
}
```

TIMER_INIT은 조건이 조금 복잡하다. (1)네 글자여야 하고, (2)세 개의 0과 한 개의 0이 아닌 숫자로 이루어져야 하고, (3)입력되는 문양은 0부터 8까지이다. 해당 조건을 만족하도록 예외처리한다.

```
/* TIMER DEVICE ROUTINE */
/* Open timer device file. */
int timer_fd = open(DEV_FILE_LOC, O_RDWR);
if (timer_fd == -1)
{
    printf("ERROR(app.c) : timer device file open failed\n");
    return -1;
}
/* Set ioctl parameter and send it to the timer device. */
char option[11];
snprintf(option, 11, "%03d%03d%04d", interval, cnt, init);
ioctl(timer_fd, IOCTL_SET_OPTION, option);
/* Start timer when the RESET signal comes in. */
while (!read(timer_fd, NULL, 0)) /* In device driver, .read fop returns true
                                  * when the RESET signal comes in.
                                  */
    usleep(100000);
ioctl(timer_fd, IOCTL_COMMAND);

/* EXIT ROUTINE */
close(timer_fd);
return 0;
```

이제 디바이스 파일을 open 하고, 사용자의 입력을 10글자의 문자열로 합쳐 ioctl(SET_OPTION)의 인자로 호출한다. 이후 RESET 스위치의 입력을 기다리면서 read를 반복한다. read로 RESET 스위치의 입력을 감지하면, ioctl(COMMAND)로 타이머를 시작한다. 타

이머가 만료되고 모든 동작이 끝나면 ioctl(COMMAND)이 return 되므로, 파일을 close 하고 종료한다.

[2] device

```c
/* FPGA DEVICE DEFINITIONS */
#define DEV_NUM 5 /* Number of devices supported */
/* SWITCH */
#define SWITCH 0
#define SWITCH_PA 0x08000000 /* Device physical address for SWITCH */
/* FND */
#define FND 1
#define FND_PA 0x08000004
/* LED */
#define LED 2
#define LED_PA 0x08000016
/* DOT */
#define DOT 3
#define DOT_PA 0x08000210
/* LCD */
#define LCD 4
#define LCD_PA 0x08000090

/* DOT MATRIX FONT */
static const unsigned char dot_number[10][10] = {
    // ...
};
```

먼저 제공된 디바이스 드라이버 코드에서 필요한 정보를 모두 가져온다. 여기에는 디바이스의 physical address, DOT matrix 값 등이 포함된다.

```c
/* Connect the physical address of the device to the virtual address of process
 * via the 'ioremap' function(map_device()) and store.
 */
static unsigned char *dev_addr[DEV_NUM];

/* All interface functions following do what their names say! */
int map_device()
{
    int i;
    /* Map each device's physical address to a virtual address, */
    dev_addr[SWITCH] = ioremap(SWITCH_PA, 0x01);
    dev_addr[FND] = ioremap(FND_PA, 0x04);
    dev_addr[LED] = ioremap(LED_PA, 0X01);
    dev_addr[DOT] = ioremap(DOT_PA, 0x10);
    dev_addr[LCD] = ioremap(LCD_PA, 0x32);
    /* and do check routine for mapping error if any. */
    for (i = 0; i < DEV_NUM; i++)
    {
        if (dev_addr[i] == NULL)
        {
            printk("ERROR(device.c) : device [%d] ioremap failed\n", i);
            return -1;
        }
    }
    return 1;
}
```

map_device 함수에선 ioremap을 통해 프로세스 주소 공간에 모든 디바이스의 주소를 매핑한다. 매핑된 주소는 dev_addr[] 배열에 저장되고, 이후 해당 배열의 값을 확인하면서 ioremap이 정상적으로 이루어지지 않은 디바이스가 있는지 확인한다.

```
void unmap_device()
{
    int i;
    /* Unmap all device virtual addresses. */
    for (i = 0; i < DEV_NUM; i++)
        iounmap(dev_addr[i]);
}
```

모듈이 종료될 때 호출되는 unmap_device는 map_device를 통해 매핑된 주소를 모두 언매핑한다.

```
int switch_read()
{
    unsigned char dip_sw_value;
    unsigned short _s_dip_sw_value;
    /* Read switch value, */
    _s_dip_sw_value = inw((unsigned int)dev_addr[SWITCH]);
    dip_sw_value = _s_dip_sw_value & 0xFF;
    /* and return 1 if switch is on, 0 otherwise. */
    if (!dip_sw_value)
        return 1;
    else
        return 0;
}

void fnd_write(const int idx, const int symbol)
{
    unsigned short int _s_value = 0 + (symbol << (12 - 4 * idx));
    /* Write value to FND. */
    outw(_s_value, (unsigned int)dev_addr[FND]);
}

void led_write(const int symbol)
{
    unsigned short _s_value = (symbol == 0 ? 0x00 : 0x01 << (8 - symbol));
    /* Write value to LED. */
    outw(_s_value, (unsigned int)dev_addr[LED]);
}

void dot_write(const int symbol)
{
    int i;
    /* Write symbol pattern to DOT matrix. */
    for (i = 0; i < 10; i++)
        outw(dot_number[symbol][i] & 0x7F, (unsigned int)dev_addr[DOT] + i * 2);
}
```

차례대로 switch, fnd, led, dot matrix에 대한 제어 인터페이스이다. 입출력 형식과 동작은 모두 제공된 코드와 완전히 동일하다. FPGA 디바이스의 align 단위가 unsigned short(2bytes)이고, 해당 단위에 맞게 적당한 비트에 값을 지정하여 inw/outw로 하드웨어에 읽고 쓸 수

있다고 추측하였다. 유일한 입력 장치인 RESET 스위치에 대해, switch_read는 입력이 감지되면 1, 아니면 0을 return 한다. 출력 장치의 경우 간결함을 위해 사용되는 문양(0~8)을 정수형으로 입력하면, 함수 내에서 알아서 형변환 후 출력까지 하도록 작성하였다.

```
void lcd_write(const char *left_up, const int right_up,
               const char *down)
{
    int i;
    unsigned char value[33]; /* Buffer to hold the formatted string to be displayed on LCD. */
    /* value[33] :
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     * | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     * | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |'\0'|
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     *
     * left_up   : [00, 12] (string)
     * right_up  : [13, 15] (≤3 digit right-aligned integer)
     * down      : [16, 31] (string)
     */
    /* Initialize the buffer to empty. */
    memset(value, ' ', 32);
    value[32] = '\0';
```

LCD 출력의 경우는 구현이 조금 복잡하다. 저번 프로젝트에서 LCD 출력을 설정하는 부분이 간결하지 못했기 때문에, 이번 프로젝트에서 개선해보았다. 이번 프로젝트의 LCD 출력의 경우 왼쪽 위 문자열, 오른쪽 위 우측 정렬된 세 자리 이하 정수, 아래쪽 문자열로 나눌 수 있다(형식을 설명한 주석을 참고하면 이해하기 쉽다). 각각의 입력을 따로 받아, 크기가 33인 문자열의 알맞은 자리에 위치시킨다. 가장 먼저, 출력값을 저장할 value 배열을 공백으로 초기화한다.

```
/* Copy the left_up string into the value buffer. */
for (i = 0; i < 13 && left_up[i] != '\0'; i++)
    value[i] = left_up[i];
/* Format the right_up integer and store in right_up_buf.
 * Copy the right_up_buf into the value buffer,
 * starting at position 13(≤3 digit right-aligned integer).
 */
if (right_up != -1) /* If right_up is -1, it means the timer has ended,
                     * so nothing will be filled.
                     */
{
    char right_up_buf[4]; /* temporary buffer to hold the formatted right_up integer */
    snprintf(right_up_buf, 4, "%3d", right_up);
    for (i = 13; i < 16; i++)
        value[i] = right_up_buf[i - 13];
}
/* Copy the down string into the value buffer, starting at position 16. */
for (i = 16; i < 32 && down[i - 16] != '\0'; i++)
    value[i] = down[i - 16];
```

이제부터 입력된 세 가지의 값에 따라 value 버퍼를 채운다. 0부터 12까지의 인덱스에는 left_up 문자열이 입력된다. 13부터 15까지의 인덱스에는 "%3d"로 우측 정렬한 세 자리 이

하의 정수가 입력된다. 16부터 31까지의 인덱스에는 down 문자열이 입력된다. 이렇게 하면 value 버퍼에 형식을 맞춘 출력값을 저장할 수 있다.

```c
/* Write value to LCD. */
unsigned short int _s_value = 0;
for (i = 0; i < 32; i += 2) /* Combine two adjacent characters into
                             * a short int and write it to the LCD.
                             */
{
    _s_value = (value[i] & 0xFF) << 8 | value[i + 1] & 0xFF;
    outw(_s_value, (unsigned int)dev_addr[LCD] + i);
}
```

마지막으로 outw로 LCD 출력을 적용한다. 제공된 코드를 참고하니, LCD의 경우 다른 디바이스와 다르게 2 바이트에 인접한 두 개의 문자를 한 번에 합쳐 표현했다.

[3] timer

```c
/* TIMER DEVICE DEFINITIONS */
#define MAJOR_NUM 242
#define DEV_NAME "dev_driver"
#define DEV_FILE_LOC "/dev/dev_driver"
/* IOCTL NUMBERS */
#define IOCTL_SET_OPTION _IOW(MAJOR_NUM, 1, char *)
#define IOCTL_COMMAND _IO(MAJOR_NUM, 2)
```

먼저 모듈에서 사용할 정보들을 정의한다. 여기엔 디바이스 드라이버의 메이저 넘버, 디바이스 이름, 디바이스 파일 위치, ioctl number 등이 포함된다.

```c
/* DATA(TIMER) STRUCTURE */
struct metadata
{
    /** constant **/
    int interval; /* TIMER_INTERVAL */
    int cnt;      /* TIMER_CNT */

    /** inconstant(variable) **/
    int elapsed; /* elapsed time - increasing by 1 each time the timer expires */
    /* LCD */
    char left_up[14];
    int right_up;
    char down[17];
    /* FND */
    int fnd_idx;
    int symbol;
};
typedef struct _TIMER
{
    struct timer_list timer;
    struct metadata info;
} TIMER;
```

다음으로 타이머의 정보를 관리하는 구조체를 정의한다. 모듈 내에서 관리되는 TIMER 구조

체는 커널 타이머(struct timer_list)와, 정보를 담고 있는 메타데이터(struct metadata) 필드로 나뉜다. 관리하는 메타데이터에는 interval(TIMER_INTERVAL), cnt(TIMER_CNT), elapsed(핸들러가 호출된 횟수, 소요 시간), 가변적인 FPGA 출력 디바이스 값들이 있다.

```c
/* TIMER */
static TIMER timer_data;
/* 'already_open' flag for I/O(open) blocking */
enum
{
    NOT_USED = 0,
    EXCLUSIVE_OPEN = 1
};
enum
{
    TIMER_PROG = 0,
    TIMER_EXPIRE = 1
};
static atomic_t already_open = ATOMIC_INIT(NOT_USED);
/* Mutex semaphore to prevent ioctl command from exiting before the timer expires */
struct semaphore TIMER_END;
```

이후 모듈에서 사용할 TIMER인 timer_data를 전역변수로 선언한다. 파일이 열려있는지를 확인하는 already_open 플래그의 열거형 값, 타이머의 진행 상태를 나타내는 열거형 값 또한 미리 선언한다. 마지막으로 already_open 플래그를 NOT_USED로 초기화하고, 타이머가 종료될 때 프로그램이 종료될 수 있도록 하는 TIMER_END mutex semaphore도 선언하면 실질적인 구현을 위한 준비가 끝난다.

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

    return 0;
}
```

insmod를 통해 커널에 해당 모듈이 추가되면 호출되는 timer_init이다. 디바이스 정보와 함께 캐릭터 디바이스를 등록하고, TIMER_END semaphore를 0으로 초기화하고, 사용할 타이머를 초기화하고, FPGA 디바이스의 주소를 매핑한다.

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

rmmod를 통해 커널에서 해당 모듈이 제거되면 호출되는 timer_exit이다. FPGA 디바이스를 언매핑하고, 타이머를 리스트에서 제거하고, 디바이스를 등록 해제한다.

```c
/* Replaced fop functions for device file HEADER */
static int timer_open(struct inode *, struct file *);
static int timer_close(struct inode *, struct file *);
static int timer_read(struct file *, char __user *, size_t, loff_t *);
static long timer_ioctl(struct file *, unsigned int, unsigned long);
static struct file_operations timer_fops = {
    .owner = THIS_MODULE,
    .open = timer_open,
    .release = timer_close,
    .read = timer_read,
    .unlocked_ioctl = timer_ioctl,
};

/* Interface/Utility functions for managing timer HEADER */
static int timer_atoi(const char *);
static void timer_metadata_init(const char *);
static void timer_display();
static void timer_add(int);
static void timer_handler(unsigned long);
```

소스 파일에는 두 종류의 함수가 존재한다. 디바이스 파일에 수행되는 file operation을 대체하는 함수들, timer를 관리하기 위한 유틸리티 함수들이다. 순서대로 설명하겠다.

```c
/* timer_open - Replace 'open' fop on device file. */
static int timer_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);
    return 0;
}

/* timer_close - Replace 'close' fop on device file. */
static int timer_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);
    return 0;
}
```

먼저 open과 close를 대체하는 timer_open, timer_close이다. timer_open에선, 하나의 프로세스에서만 디바이스 파일을 열 수 있도록 already_open 플래그를 atomic하게 체크한다. 만약, 이미 파일이 열려 already_open이 NOT_USED가 아닌 경우 에러를 return 한다. 그렇지 않은 경우 already_open을 EXCLUSIVE_OPEN으로 설정한다. 이후 모듈의 usage를 증가시키기 위해 try_module_get을 호출한다. timer_close에선 정반대로 already_open을 NOT_USED로 설정하고 usage를 낮추기 위해 module_put을 호출한다. (사실 이번 프로젝트의 경우 app 프로세스에서만 타이머 디바이스 드라이버를 사용하기 때문에 플래그 체크 루틴은 필요가 없지만, 강의 자료로 사용된 LKMPG의 모듈 컨벤션을 참고해 그대로 따랐다.)

```c
/* timer_read - Replace 'read' fop on device file. */
static int timer_read(struct file *file, char __user *buffer, size_t length, loff_t *offset)
{
    /* Read the input from the RESET switch and return it. */
    return switch_read();
}
```

read를 대체하는 timer_read의 경우 앞서 여러 번 설명한 것처럼 RESET switch 입력을 return 한다.

```c
/* timer_ioctl - Replace 'ioctl' fop on device file. */
static long timer_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
{
    switch (ioctl_num)
    {
    case IOCTL_SET_OPTION:
    {
        /* Read the option from user pointer passed in. */
        char *u_option = (char *)ioctl_param;
        char tmp_buf[11] = {0};
        if (strncpy_from_user(tmp_buf, u_option, 10) < 0)
        {
            printk("ERROR(timer.c) : strncpy_from_user failed\n");
            return -1;
        }
        /* Initialize timer's metadata(info field) based on the option,
         * and display it on FPGA device.
         */
        timer_metadata_init(tmp_buf);
        timer_display();
        /* Timer's ready waiting for IOCTL_COMMAND! */
        break;
    }
```

ioctl의 경우 ioctl number에 따라 동작이 다르기 때문에, 함수 내에서 구별하여 독립적으로 동작하도록 한다. ioctl_num이 IOCTL_SET_OPTION인 경우, 유저 파라미터로 option이 입력되므로 해당 정보를 복사해온다. 타이머 초기값이 설정되어 있는 option으로부터 timer_data를 초기화하고, FPGA 디바이스에 그 값을 출력한다.

```
    case IOCTL_COMMAND:
    {
        /* Reset and register timer. */
        del_timer_sync(&(timer_data.timer));
        timer_add(TIMER_PROG);
        down_interruptible(&TIMER_END); /* blocked until timer expires */
        break;
    }
    default: /* unset numbers */
    {
        printk("ERROR(timer.c) : unknown ioctl number\n");
        return -1;
    }
    }
    return 0;
}
```

ioctl_num이 IOCTL_COMMAND인 경우 타이머를 리셋하고, 초기화된 타이머를 리스트에 등록한다. 이후, 타이머가 완전히 만료될 때까지 프로세스를 block 하기 위해 semaphore down operation을 한다.

```
/* timer_atoi - Simple kernel atoi. [ASCII string → (unsigned) int] */
static int timer_atoi(const char *str)
{
    int i;
    int res = 0;
    for (i = 0; str[i] != '\0'; i++)
        res = res * 10 + str[i] - '0';
    return res;
}
```

이제부턴 유틸리티 함수이다. 가장 먼저, timer_atoi는 기존 유저 레벨 라이브러리의 atoi를 커널에서 사용할 수 없어, 필요한 정도로만 유사하게 구현한 함수이다. 0 또는 짧은 양수를 나타내는 문자열을 10진수 정수로 변환한다.

```
/* timer_metadata_init - Initialize timer's metadata(info field) based on option. */
static void timer_metadata_init(const char *option)
{
    int i;
    char buf[5] = {0};
    int u_interval, u_cnt;
    char u_init[5] = {0};

    /* option[11] :
     * +----+----+----+----+----+----+----+----+----+----+----+
     * | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |'\0'|
     * +----+----+----+----+----+----+----+----+----+----+----+
     *
     * interval   : [00, 02] (001 - 100)
     * cnt        : [03, 05] (001 - 200)
     * init       : [06, 09] (0001 - 8000)
     */
    /* Tokenize option. */
    /* interval */
    memcpy(buf, option, 3);
    u_interval = timer_atoi(buf);
    /* cnt */
    memcpy(buf, option + 3, 3);
    u_cnt = timer_atoi(buf);
    /* init */
    memcpy(u_init, option + 6, 4);
```

timer_metadata_init은 option 문자열을 바탕으로 timer_data의 (struct metadata) info 필드를 초기화한다. 가장 먼저, option의 형식(주석 참고)에 따라 interval, cnt, init 값을 tokenize 한다.

```
    /* Set timer's metadata. */
    struct metadata tmp;
    /* interval, cnt, elapsed */
    tmp.interval = u_interval;
    tmp.cnt = u_cnt;
    tmp.elapsed = 0;
    /* LCD fields */
    snprintf(tmp.left_up, 9, "20211584 ");
    tmp.right_up = tmp.cnt - tmp.elapsed;
    snprintf(tmp.down, 4, "JJY ");
    /* FND fields */
    for (i = 0; i < 4; i++)
    {
        if (u_init[i] != '0')
        {
            tmp.fnd_idx = i;
            tmp.symbol = u_init[i] - '0';
        }
    }
    timer_data.info = tmp;
}
```

이후 적절하게 필드를 채워주면 된다. 할당되는 값을 보면, 타이머가 시작하기 전 보여지는 가장 초기의 값이다. FND와 관련된 필드를 채울 땐, u_init을 모두 확인하여 0이 아닌 문양을 갖는 인덱스와 그 문양을 저장한다.

```
/* timer_display - Display timer's metadata on FPGA devices. */
static void timer_display()
{
    fnd_write(timer_data.info.fnd_idx, timer_data.info.symbol);
    led_write(timer_data.info.symbol);
    dot_write(timer_data.info.symbol);
    lcd_write(timer_data.info.left_up, timer_data.info.right_up, timer_data.info.down);
}
```

timer_display 함수는 단순히 timer_data의 메타데이터를 FPGA 디바이스에 출력해준다.

```
/* timer_add - Set timer field and register timer. */
static void timer_add(int flag)
{
    if (flag == TIMER_EXPIRE) /* If timer expires, */
        timer_data.timer.expires = get_jiffies_64() + HZ; /* 1 (sec) */
    else                      /* Otherwise, */
        timer_data.timer.expires = get_jiffies_64() +
                                   (timer_data.info.interval * HZ / 10); /* 0.1 * interval (sec) */
    timer_data.timer.data = (unsigned long)&timer_data;
    timer_data.timer.function = timer_handler;
    add_timer(&(timer_data.timer));
}
```

timer_add는 타이머의 필드를 적절한 값으로 채우고 add_timer로 리스트에 추가해주는 함수

이다. 이때 flag는 elapsed가 아직 cnt보다 작은 경우 TIMER_PROG이고, cnt 이상인 경우 TIMER_EXPIRE가 된다. flag가 TIMER_EXPIRE인 경우 3초 동안 종료 알림 메시지를 띄워야 하므로 expires를 1초 뒤로 설정하고, 아닌 경우 유저가 지정한 대로 설정한다. 핸들러로 넘어가는 data는 timer_data, 핸들러는 timer_handler로 지정하여 add_timer를 호출한다.

```c
/* timer_handler - Timer handler called whenever the timer expires(timer.expires == get_jiffies_64()). */
static void timer_handler(unsigned long timeout)
{
    struct metadata tmp = ((TIMER *)timeout)->info;

    /* elapsed proceeds :
     * [1              ][2    ][3       )
     * +---------------+------+-------->
     * 0               cnt    cnt+3    INF
     *
     * Interval 1 : Section proceeding by user-specified number 'TIMER_CNT'.
     * Interval 2 : 3-second notification section indicating that the timer will expire.
     * Interval 3 : Expire timer. End routine.
     */
    tmp.elapsed++;
```

타이머 핸들러인 timer_handler는 세 개의 구간에 따라 동작을 달리 한다. 주석을 참고하면, 유저가 지정한 cnt 까지 세고 있는 구간 1과, 유저가 지정한 cnt가 되어 3초의 종료 알림을 띄우는 구간 2와, 알림까지 마쳐 타이머를 완전히 종료하는 구간 3이 존재한다. elapsed는 핸들러가 호출된 횟수이므로, 핸들러 호출시마다 1씩 증가시킨다. 이 elapsed를 통해 구간을 지정하고 동작을 수행한다.

```c
    /* Interval 1 */
    if (tmp.elapsed < tmp.cnt)
    {
        /* Update metadata (INCREASING TO TIMER_CNT). */
        tmp.right_up = tmp.cnt - tmp.elapsed;
        if (++tmp.symbol == 9) /* If symbol is out of range, rotate it. */
            tmp.symbol = 1;
        if (tmp.elapsed % 8 == 0) /* When symbol has completed one cycle,
                                   * index moves on to the next.
                                   */
            tmp.fnd_idx = (tmp.fnd_idx + 1) % 4;
        /* Display the updated information and add the timer again. */
        timer_data.info = tmp;
        timer_display();
        timer_add(TIMER_PROG);
    }
```

구간 1에선 타이머가 실질적으로 동작하고 있으므로, FPGA 디바이스 출력값이 변화한다. right_up은 cnt에서 elapsed를 뺀 값으로, symbol은 1부터 8을 순환하도록, fnd_idx는 elapsed가 symbol의 주기인 8의 배수가 되었을 때 한 칸씩 옆으로 움직이도록 설정한다. 이

후 변경된 값이 디바이스에 적용되도록 timer_display를 호출하고, 변경된 타이머를 TIMER_PROG 플래그와 함께 다시 등록한다.

```
/* Interval 2 */
else if (tmp.cnt ⩽ tmp.elapsed &&
         tmp.elapsed < tmp.cnt + 3)
{
    /* Update metadata (EXPIRATION NOTIFICATION). */
    snprintf(tmp.left_up, 11, "Time's up! ");
    tmp.right_up = 0;
    snprintf(tmp.down, 17, "Shutdown in %d... ", 3 - (tmp.elapsed - tmp.cnt));
    tmp.fnd_idx = 0;
    tmp.symbol = 0;
    /* Display the updated information and add the timer again. */
    timer_data.info = tmp;
    timer_display();
    timer_add(TIMER_EXPIRE);
}
```

구간 2에선 LCD의 left_up, down을 종료 알림 메세지으로 바꾼다. 나머지 FND, LED, DOT matrix는 꺼질 수 있도록 symbol을 0으로 설정한다. 변경된 값이 디바이스에 적용되도록 timer_display를 호출하고, 변경된 타이머를 TIMER_EXPIRE 플래그와 함께 다시 등록한다.

```
/* Interval 3 */
else
{
    /* Update metadata (TURN-OFF ALL FPGA DEVICES). */
    snprintf(tmp.left_up, 1, " ");
    tmp.right_up = -1;
    snprintf(tmp.down, 1, " ");
    tmp.fnd_idx = 0;
    tmp.symbol = 0;
    /* Display the updated information and do semaphore-up-operation
     * to notify that the timer has expired completely and unblock module process.
     */
    timer_data.info = tmp;
    timer_display();
    up(&TIMER_END);
}
```

구간 3에선 LCD를 포함한 모든 FPGA 출력 디바이스를 초기화하고 timer_display를 호출한다. 타이머가 완전히 종료한 시점이므로 timer_add를 호출해 타이머를 재등록하지 않고, TIMER_END semaphore에 up operation을 수행해 timer_ioctl에서 block 되어있는 모듈 프로세스를 깨워준다.

```
module_init(timer_init);
module_exit(timer_exit);

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

마지막으로 init, exit 함수를 지정하고 라이선스와 저자를 표기하면 모듈 프로그램이 완성된

다.

IV. 연구 결과

요구 사항을 모두 만족하였다. 지난 프로젝트의 채점 결과에 '명세서를 더 꼼꼼히 확인해야 한다'는 평가가 있었기 때문에, 이번에는 명세서를 복기하며 자가 채점을 해보고자 한다.

[1] 응용 프로그램(app)

- 응용 프로그램에서 두 개의 ioctl 명령어 사용 O
- 응용 프로그램 입력에 올바른 값이 들어갈 수 있도록 예외처리 O
- 구동 방식(ioctl, 프로그램 종료 시점) O

[2] 타이머 디바이스 드라이버(driver, timer)

- 디바이스 드라이버와 타이머 기능을 포함한 하나의 모듈 구현 O
- 모든 디바이스의 값들이 바뀌는 시간은 TIMER_INTERVAL을 기준으로 구현 O
- 타이머 디바이스 드라이버 이름 및 메이저 넘버 설정 O
- FND 출력 O
- LED 출력 O
- DOT 출력 O
- LCD 출력 O
- SWITCH 입력 O

V. 기타

이번 과제는 시간 투자를 확실하게 해서 지난 과제처럼 코드의 문제가 많지는 않은 것 같다. 문제점을 개선하기보단 추가로 구현할 수 있는 기능을 생각해보았다.

[1] 과제 마감 주 화요일의 수업시간에 I/O blocking에 대해 간략하게 알게 되었다. 중복

open에 대한 blocking 말고도, 모든 file operation에 대한 I/O blocking을 구현한다면 좀 더 안정적인 모듈이 될 것이다.

[2] 이번 프로젝트에서 타이머 모듈은 단일 프로세스에서만 실행되기 때문에, open/close에 있어 blocking이 제대로 되고 있는지 확인할 방법이 없다. app에 fork 루틴 등을 추가해서, 여러 프로세스가 해당 디바이스 파일에 접근하려고 할 때 오류가 없는지 확인해볼 수 있을 것이다.