Embedded System Software 과제 2

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20171664, 이상윤

개발기간: 2024. 05. 08. - 2024. 05. 13.

최 종 보 고 서

I. 개발 목표

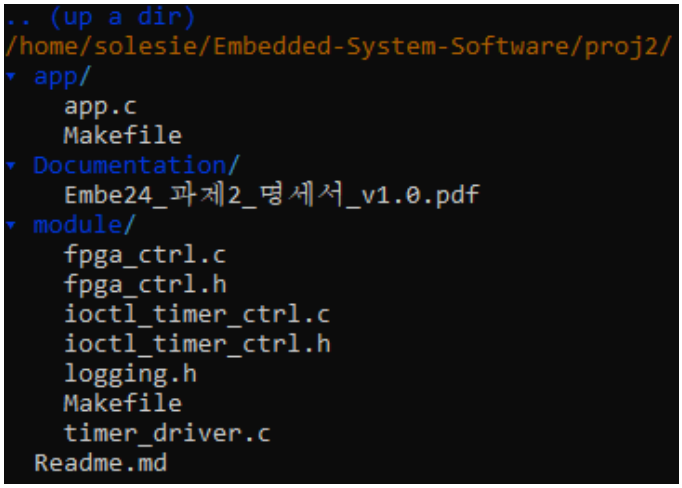
디바이스 드라이버의 구조와 fpga 장치를 register로 조작하는 법을 확실하게 파악한다. 그리고 리눅스 상 kernel timer와 이와 연관된 timer interrupt context 개념 등을 파악한다. 이를 바탕으로 위 내용을 응용하는 타이머 디바이스 드라이버 모듈을 구현한다. 또한 타이머 디바이스 드라이버를 테스트하는 사용자 레벨 응용 프로그램을 개발한다.

II. 개발 범위 및 내용

가. 개발 범위

1. 기존 huins fpga device driver 코드를 적절히 참고하여 fpga device register를 적절히 조작하여 장치 입출력을 구현한다.
2. Timer interrupt context를 적절히 이해하여 fpga device를 timer 로직에 맞게 작동시키는 timer controller를 구현한다.
3. Module interface에 맞게 timer controller을 작동시키는 timer device driver을 구현한다.
4. 해당 타이머 디바이스 드라이버를 사용하는 사용자 레벨 응용 프로그램을 작성한다.

나. 개발 내용



<그림1: 프로젝트 개발 디렉토리 구조>

1. module/fpga_ctrl.c(장치 입출력)

   출력 디바이스는 dot, fnd, led, text lcd가 있고 입력 디바이스에는 dip switch가 있다. 해당 소스코드에서는 우선 입출력 디바이스 레지스터를 ioremap과 같은 함수를 통해 메모리에 매핑하거나 해제하는 역할을 한다. 또한 timer 로직이 진행됨에 따라 규격에 맞도록 출력하는 역할과 blocking하게 dip switch를 읽는 역할도 담당한다.

2. module/loctl_timer_ctrl.c(타이머 컨트롤러)

   fpga_ctr.cl를 사용하는 소스코드이다. 이는 몇초마다 timer를 작동시킬 것인지, 몇번 timer를 작동시킬 것인지, 인자가 유효한지 등과 같은 타이머 핵심 로직을 담당한다. ioctl(SET_OPTION)로부터 사용자로부터 주어진 timer_interval, timer_cnt, timer_init의 값들이 timer controller에 세팅된다. 또한 timer callback 함수들이 위치한다. ioctl(COMMAND)로부터 호출되는 함수를 통해 타이머를 작동시킬 수 있다.

3. module/timer_driver.c(타이머 디바이스 드라이버 모듈)

   insmod, rmmod 될때의 동작을 정의하고, open, close, read, ioctl과 같은 file_operations들을 정의한다. 또한 user level에서 ioctl(SET_OPTION)을 사용하지 않고 ioctl(COMMAND)를 사용한다거나, 여러 프로세스에서 open하려 한다거나 등 정의되지 않은 행동을 방지한다.

4. app/app.c(사용자 레벨 타이머 디바이스 드라이버 응용 프로그램)

   사용자로부터 주어진 argc, argv의 유효성을 검증한다. 또한 insmod 되어있지 않은 경우 execl() 함수를 통해 insmod를 수행한다. 그리고 timer device driver을 open한 다음 두 ioctl을 통해 작동시킨다.

III. 추진 일정 및 개발 방법

가. 추진 일정

- 2024/05/08: 프로젝트 분석 및 구조 구성
- 2024/05/09: fpga I/O device 입출력 구현
- 2024/05/10: timer controller(타이머 로직) 구현
- 2024/05/11: module interface 구현
- 2024/05/12: 사용자 응용 프로그램 개발
- 2024/05/13: 로직 검증 및 테스트

나. 개발 방법

1. 명세서 분석

./app TIMER_INTERVAL TIMER_CNT TIMER_INIT

사용자 응용프로그램은 위 명령으로 시작된다. TIMER_INTERVAL은 1~100 사이의 정수이고 0.1 ~ 10초 까지의 타이머 interval을 나타낸다. TIMER_CNT는 1~100 사이의 정수이고 몇번 타이머를 작동시킬 것인지 나타낸다. TIMER_INIT은 4자리 문자열으로서 한자리 수만 1~8사이의 수이고 나머지는 0으로 채워져있다.

1~8사이의 문양이 time_interval마다 fnd에서 증가되며 로테이션마다 오른쪽으로 이동한다. 그때마다 led와 dot이 함께 바뀐다. fnd에서 제일 오른쪽이라면 1번째 자리로 이동한다. Text lcd는 time_interval마다 이름 영문 이니셜(LSY)가 두번째줄을 한칸씩 이동하며 끝에 다다른 경우 방향을 바꾼다. 또한 timer_cnt는 1씩 줄어든다.

Timer가 끝난 순간 countdown이 시작된다. 이때 text lcd 출력 디바이스만 켜져있고 나머진 꺼진다. 두번재 줄에 3초 카운트 다운을 표시하고 종료한다. 종료시 text lcd도 꺼진다.

./app 10 5 1000 시 예시를 분석해보면 아래와 같다. 각각 상태에서 1~2째줄은 text lcd, 3번째 줄은 fnd, 4번째 줄은 dot, 5번째 줄은 led의 값을 나타낸다.

<초기 상태>
20171664 5
LSY
1000
1
D1

이 상태에서 reset 버튼을 누르고 향후 5초간 아래와 같이 변한다.

<0~1초>(초기상태와 같음)
20171664 5
LSY

1000

1

D1

<1~2초>

20171664 4

LSY

2000

2

D2

<2~3초>

20171664 3

LSY

3000

3

D3

<3~4초>

20171664 2

LSY

4000

4

D4

<4~5초>

20171664 1

LSY

5000

5

D5

이제 5초가 지나자 마자 카운트 다운이 시작된다.

<0~1초>

Time's up!　　0

Shutdown in 3...

<1~2초>

Time's up!　　0

Shutdown in 2...

<2~3초>

Time's up!　　0

Shutdown in 1...

3초가 다 지나면 완전히 종료된다.

2. timer interrupt context

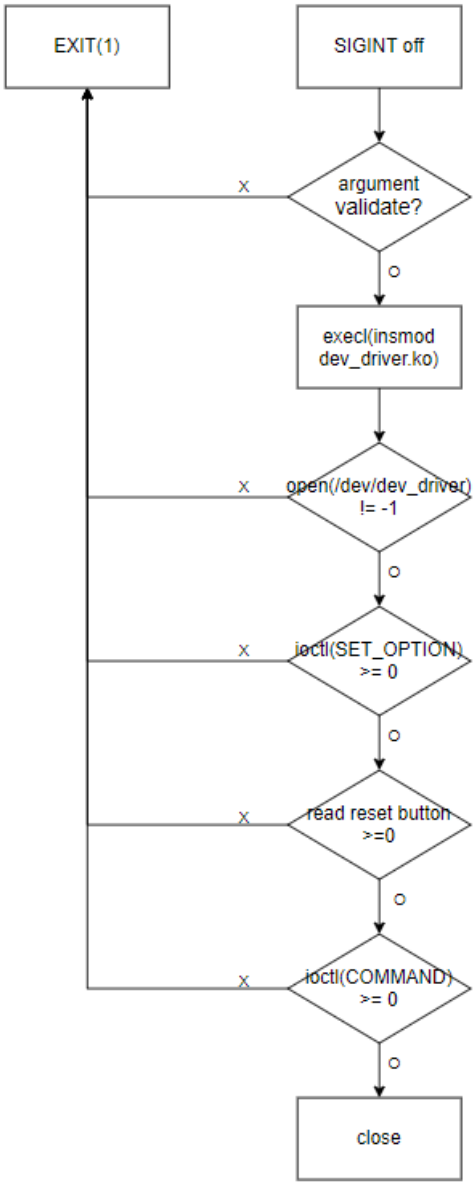
Interrupt context 에선 주의할 점이 있다. 내부적으로 sleep과 관련있는 자료구조를 최대한 배제해야 한다는 점이다. 이번 프로젝트에서 fpga device를 memory map 하고, 다시 unmap 하는 과정에서 ioremap, iounmap 과 같은 함수들을 사용한다. 이와 같이 새로운 메모리 영역을 매핑하는 함수들은 비용이 비싼 함수로서, 전형적으로 sleep할 가능성(blocking)이 있는 기능들을 내부적으로 사용한다. 따라서 카운트 다운이 진행되는 interrupt context 내부에서 카운트 다운이 종료되었을 때, iounmap을 사용하여 메모리 매핑을 해제한다던가 하는 행위는 커널을 다운시킬 수 있다. 따라서 이러한 상황이 일어나지 않도록 개발해야한다.

또한 ioctl(COMMAND)를 blocking하게 수행되도록 계획하였다. 따라서 타이머 프로그램 실행하고 있을 때, 커널 내 프로세스 컨텍스트는 interrupt context가 완료될 때까지 기다리고 있어야 한다. 이때 폴링과 같은 방식으로도 가능하지만, 가

능하면 커널 내 정의된 라이브러리를 활용하는 방향이 적절하다. 여기서 wait queue, wait event와 같은 방법들이 있다. 이번 프로젝트에서는 completion 자료구조를 채택한다. 이는 내부적으로 wait queue, spinlock들을 사용한다. 즉, waiting queue 자료구조를 직접 구현하고 관리하지 않아도 된다는 장점이 있다. 이를 통해 ioctl(COMMAND)가 blocking하게 수행되도록 한다.

Ⅳ. 연구 결과

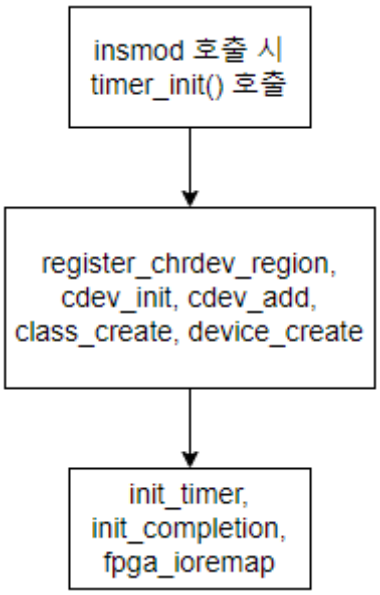
1. app.c 사용자 응용 프로그램 flow chart



<그림2: 사용자 응용 프로그램>

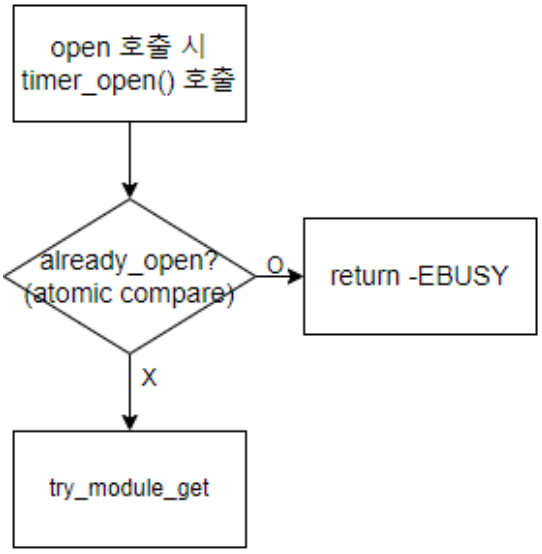
<그림2>는 app.c의 흐름도를 나타낸다. 사용자로부터 입력받은 command-line argument를 검사하여, 예외 처리를 철저히 한다. 그 후, module 디렉토리 내의 Makefile의 “make push” 명령어를 통해 보드의 data/local/tmp 디렉토리로 퓨징한 dev_driver.ko 파일을 execl 함수를 통해 insmod한다. 여기서 주의할 점은, app실행파일과 dev_driver.ko 파일은 같은 디렉토리에 위치해야 한다는 점이다. 그 후 파일을 열고, ioctl(SET_OPTION)을 통해 타이머를 세팅한 후, dip switch(reset button)을 동기적으로 읽고 ioctl(COMMAND)를 통해 타이머 프로그램을 실행한다. 이 타이머 프로그램 또한 동기적(blocking)하게 실행되고, 사용자 레벨에서 따로 기다릴 필요는 없다. 마지막으로 close를 하고 프로그램을 종료한다. execl까지 정상적으로 진행되었다면, 뒤의 단계에서 오류가 발생할 가능성은 거의 존재하지 않는다.

2. timer device module flow chart



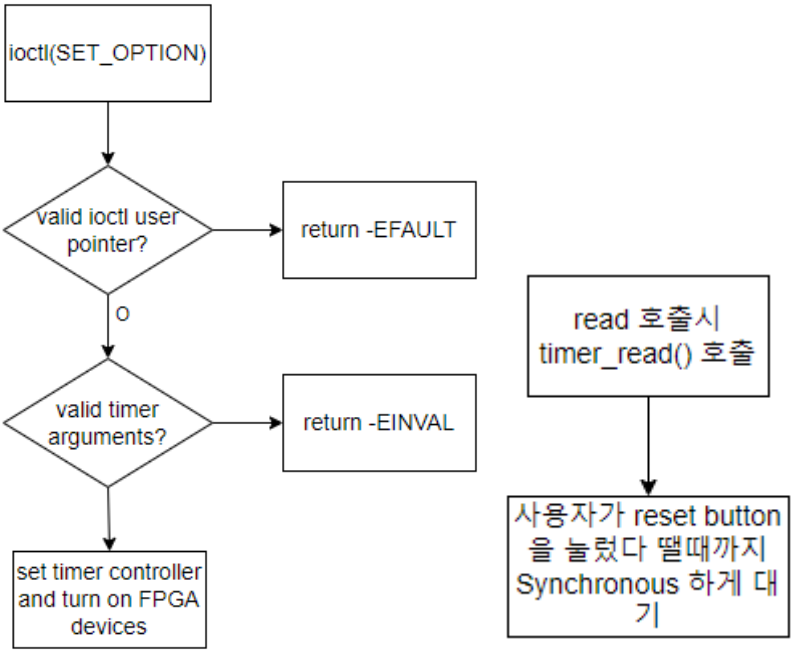
<그림3: insmod>

<그림3>은 insmod시 과정을 나타낸다. 여러 초기화 단계를 거침을 볼 수 있다. 여기서 register_chrdev_region, cdev를 통해, major, minor number를 할당하고 class_create, device_create를 통해 디바이스 파일을 /dev/dev_driver.ko에 생성한다.



<그림4: open>

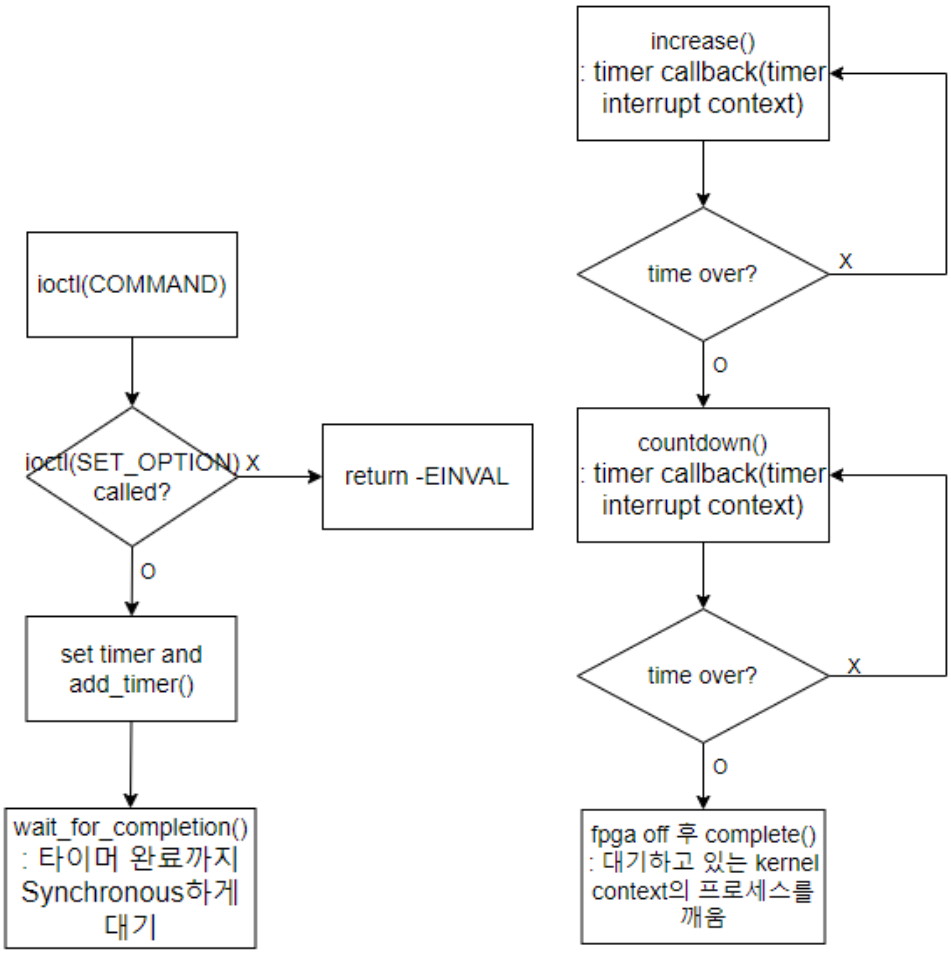
<그림4>는 open file operation 호출 시 과정을 나타낸다. 해당 디바이스 파일은 커널 상 단 하나의 프로세스나 스레드에 의해서만 열려질 필요가 있다. 따라서 atomic하게 열려져있는지 여부를 비교하고, 열려져있다면 -EBUSY를 반환하도록 한다. 또한 try_module_get을 통해 usage counter를 증가시킨다.



<그림5, 6: ioctl(COMMAND) 전처리 단계>

<그림5, 6>은 ioctl(COMMAND) 호출전 ioctl(SET_OPTION)로 타이머를 세팅하고 read()로 dip switch(reset button)를 읽기까지의 흐름을 나타낸다. 사용자의 포인터와 사용자의 타이머 인자값 유효성을 검사하고 유효하지 않으면 에러값을 반환한

다. 또한 dip switch를 눌렀다 떼었을 때 dip switch를 눌렀다 판단하도록 한다. 사용자 입장에서 read()를 호출 시, dip switch를 눌렀다 떼기 전까지 다음단계로 진행할 수 없다.

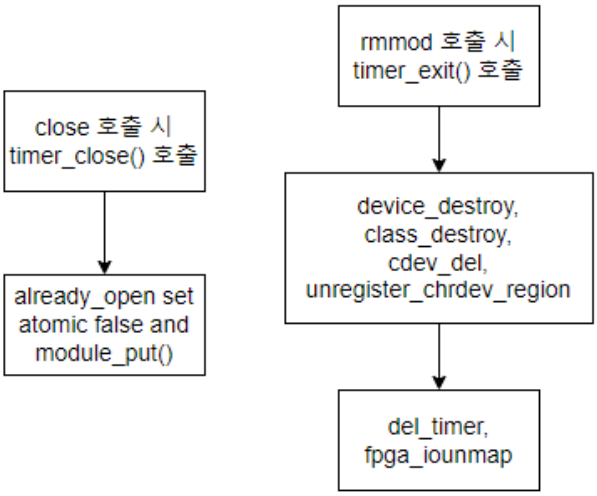


<그림7, 8: 타이머 실행>

<그림7>은 ioctl(COMMAND) 호출 시 흐름을 나타낸다. 타이머 디바이스 드라이버를 사용하는 사용자는 ioctl(SET_OPTION) 이후에 ioctl(COMMAND)를 호출해야 한다는 제약이 존재한다(이전에 open을 하였나는 기존 커널이 관리해준다). 즉, ioctl(COMMAND)를 먼저 호출해버리는 정의되지 않은 사용자 예외가 존재한다. 이를 확인하여 만약 타이머가 세팅되지 않았다면 -EINVAL 값을 반환한다. 그리고 이전에 언급한 바와 같이, completion 자료구조를 통해 타이머 완료까지 blocking하게 대기하는 것을 볼 수 있다.

<그림8>은 timer interrupt context 과정을 나타낸다. 설정된 timer interval이 지날 때 마다 timer cnt를 감소시키며 dot, fnd, text lcd, led와 같은 fpga 출력장치를 조작한다. 카운트 다운도 완료되었다면 대기하고 있는 커널 process context를

complete()를 통해 깨운다.



<그림9, 10: 종료단계>

close() 시에는 atomic하게 열려져있지 않다고 세팅하고 module_put을 통해 usage counter를 감소시킨다. rmmod 시에는 생성한 /dev/dev_driver.ko 파일을 삭제하고 디바이스 드라이버를 unload시킨다.

3. ioctl

```
19 struct ioctl_set_option_arg {
20     unsigned int timer_interval, timer_cnt;
21     char timer_init[FND_MAX + 1];
22 };
23
24 #define TIMER_MAJOR 242
25 #define TIMER_NAME "dev_driver"
26 #define IOCTL_SET_OPTION _IOW(TIMER_MAJOR, 0, struct ioctl_set_option_arg)
27 #define IOCTL_COMMAND _IO(TIMER_MAJOR, 1)
```

```
59         case IOCTL_SET_OPTION:
60             if(copy_from_user(&data, (struct ioctl_set_option_arg __user *)arg, sizeof(data)))
61                 return -EFAULT;
62             // invalid ioctl args
63             if(set_timer_ctrl(data.timer_interval, data.timer_cnt, data.timer_init) == 0)
64                 return -EINVAL;
65             return 0;
```

<그림11, 12: module/timer_driver.c>

사용자는 kernel의 ioctl_set_option_arg의 구조체와 일치하는 메모리 구조의 포인터를 ioctl의 arg로 넘겨주어야 한다. 또한 kernel에 메크로로 #define된 SET_OPTION, COMMAND와 일치하는 cmd를 ioctl에 넘겨주어야 한다. 위의 커널에 정의한 자료구조와 일치하지 않는 인자를 사용자가 넘겨줄 경우, -ENOTT(유저가 ioctl command가 정의되지 않음), -EFAULT(사용자의 ioctl arg가 유효한 pointer 영역이 아님), -EINVAL(사용자의 ioctl arg의 ioctl_set_option_arg 값이 유효하지 않거나 ioctl 호출 순서가 잘못됨)과 같은 에러를 반환한다.

4. timer controller

```c
15 static struct timer_list timer;
16 static struct completion over;
17
18 static unsigned int timer_interval;
19 static unsigned int timer_cnt;
20 static char timer_init[FND_MAX + 1];
```

```c
102 /* Return 0 on ERROR. */
103 int run_timer_ctrl(void){
104     if(!validate()) return 0;
105
106     LOG(LOG_LEVEL_INFO, "Start timer...");
107
108     // Let's run timer...
109     del_timer_sync(&timer);
110     timer.expires = get_jiffies_64() + (timer_interval * HZ / 10);
111     timer.data = 0;
112     timer.function = increase;
113     add_timer(&timer);
114
115     // Sleep until timer is completed.
116     wait_for_completion(&over);
117
118     // Turn off timer controller.
119     fpga_off();
120     timer_interval = 0;
121     timer_cnt = 0;
122     memset(timer_init, 0, sizeof(timer_init));
123
124     return 1;
125 }
```

<그림13, 14: ioctl_timer_ctrl.c>

```c
35 /* countdown callback function(per 1sec) */
36 static void countdown(unsigned long cur){
37     if(++cur >= COUNTDOWN){
38         complete(&over);
39         return;
40     }
41     fpga_countdown();
42
43     timer.expires = get_jiffies_64() + HZ; // 1sec
44     timer.data = cur;
45     timer.function = countdown;
46     add_timer(&timer);
47 }
48
49 /* increase callback function(per timer_interval) */
50 static void increase(unsigned long cur){
51     if(++cur >= timer_cnt){
52         fpga_set_countdown();
53
54         timer.expires = get_jiffies_64() + HZ; // 1sec
55         timer.data = 0;
56         timer.function = countdown;
57         add_timer(&timer);
58         return;
59     }
60     fpga_increase();
61
62     timer.expires = get_jiffies_64() + (timer_interval * HZ / 10);
63     timer.data = cur;
64     timer.function = increase;
65     add_timer(&timer);
66 }
```

<그림15: ioctl_timer_ctrl.c, timer callback functions>

사용자가 넘겨준 ioctl(SET_OPTION,arg)에서 arg의 값이 유효한 경우, <그림

13>의 타이머와 관련된 static 자료구조 값들이 초기화된다. ioctl(COMMAND)로 run_timer_ctrl()가 실행되는데, 이전에 언급한 completion 자료구조가 있음을 볼 수 있다. 여기서 주의할 점은, 설정한 타이머 자료구조와 관련된 static 변수들을 모두 초기화 해주어야 한다는 점이다. 왜냐하면 이는 커널 영역이고, 해당 static 값들은 rmmod하지 않는 한 계속 남아있기 때문이다. 다시 open했을 때 해당 값이 남아있으면 정의되지 않은 행동을 일으킬 수도 있고 안전하지도 않다.

<그림 14, 15>에선 add_timer()로 타이머를 설정하는 것을 볼 수 있다. 여기서 카운트 다운의 경우 expires 값을 get_jiffies_64() + HZ로, 타이머 increase의 경우에는 get_jiffies_64() + (timer_interval * HZ / 10)으로 설정하는 것을 볼 수 있다. 명세서에 따르면 interval 1~100은 각각 0.1초~10초를 의미한다. (x초/(1/HZ)) = HZ*x번 jiffies 이므로, interval / 10 * HZ 가 그 expire 값이 되어야 한다.

5. fpga 입출력 장치 조작

```c
/* represent current fpga status */
static int dot;
static unsigned char fnd[FND_MAX];
static unsigned char pivot;
static int led;
static char text_lcd[TEXT_LCD_MAX_BUFF + 1];
static int name_loc;
static int timer_cnt;
static int countdown;
static int dir = 1;
```

```c
static void print(void){
    int i;
    unsigned short int fnd_short;
    unsigned short value;
    unsigned short int _s_value;

    // print dot
    for(i = 0; i < 10; i++)
        outw(dot_table[dot][i] & 0x7F, (unsigned int) iom_fpga_dot_addr + i * 2);

    // print fnd
    fnd_short = fnd[0] << 12 | fnd[1] << 8 | fnd[2] << 4 | fnd[3];
    outw(fnd_short,(unsigned int)iom_fpga_fnd_addr);

    // print led
    value = led > LED_MAX ? 0 : 1 << (LED_MAX- led);
    outw(value, (unsigned int) iom_fpga_led_addr);

    // print text lcd
    _s_value = 0;
    for(i = 0; i < TEXT_LCD_MAX_BUFF; i += 2){
        // 16bit
        _s_value = ((text_lcd[i] & 0xFF) << 8) | (text_lcd[i + 1] & 0xFF);
        outw(_s_value,(unsigned int)iom_fpga_text_lcd_addr + i);
    }
}
```

<그림16, 17: fpga_ctrl.c 자료구조와 출력장치 조작>

<그림16>은 현재 fpga 출력장치에 출력되고 있는 값들을 나타낸다. <그림17>은

메모리 매핑된 디바이스 레지스터에 outw()함수를 통해 해당 메모리 영역에다 값을 써 fpga 출력장치를 조작하는 것을 보여주고 있다.

```
165 /* Turn off fpga devices. */
166 void fpga_off(void){
167     dot = DOT_TABLE_MAX + 1;
168     memset(fnd, 0, sizeof(fnd));
169     led = LED_MAX + 1;
170     memset(text_lcd, 0, sizeof(text_lcd));
171     memset(text_lcd, ' ', TEXT_LCD_MAX_BUFF);
172 
173     pivot = 0;
174     name_loc = 0;
175     timer_cnt = 0;
176     countdown = 0;
177     dir = 1;
178 
179     print();
180 }
```

<그림18: fpga_ctrl.c 종료>

<그림18>은 <그림14>에서와 같이 커널영역의 코드이기 때문에 rmmod시 까지 값이 남아있고, static 값들을 초기화하지 않으면 위험하기 때문에 확실히 초기화해주는 코드를 나타낸다.

V. 기타

Interrupt context, linux kernel timer에 대해서 뿐만 아니라 Memory-Mapped-IO 구조에 대해서도 알 수 있어서 Linux kernel에 대해 많은 것들을 배워갈 수 있었던 2번 프로젝트였습니다.