IoTSW_16_최낙관(2016101124)

- Device driver

1. 실습 예제

이번 실습은 LED를 제어하는 device driver(이하 DD)를 만들어보는 실습이었습니다.

DD를 file형식의 module로 만들어, kernel에 추가하는 형식으로 구현이 됩니다. 예제의 DD는 character device 방식으로 구현이 되었기 때문에 사용자 process에 interface를 이용해, 바로 읽기쓰기를 하여 구현합니다. DD를 위한 code를 보면, 우선 insmod를 통해, kernel에 module이 적재가 되면, module_init이 호출되고, led_init 함수가 호출이 됩니다.

```
static int led_init(void){
    printk("[led_dd] led_init()\n");
    register_chrdev(MAJOR_NUMBER, DEVICE_NAME, &fops);

    return 0;
}
```

led_init 함수에서는 register_chrdev가 호출되어, kernel에 DD의 주번호, 이름, operation 구조체를 등록합니다. Operation 구조체는 character DD와 application 연결에 사용되는 구조체로,

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = led_open,
    .release = led_release,
    .write = led_write,
};
```

위와 같이 선언, 정의되어, 해당하는 동작과 함수를 연결시킵니다. Open은 DD가 처음 열렸을 때 H/W를 초기화 시켜주는 함수입니다.

```
static int led_open(struct inode *inode, struct file *filp){
    int i;

    initGpioAddr();
    for(i = 0; i < 16; i++){
        pinMode(Led[i], OUTPUT);
        digitalWrite(Led[i], LOW);
    }

    printk("[led_dd] led_open\n");

    return 0;
}
```

위와 같이 각 LED와 gpio 주소를 초기화해줍니다.

Write는 사용자 영역에서 Device에 data를 쓰기 위한 포인터입니다.

```
static int led_write(struct file *filp, const char *buf, size_t len, loff_t
*f_pos){
    int i;
    char state;

    for(i = 0; i < len; i++){
        copy_from_user(&state, &buf[i], 1);

        if(state == '0'){
            digitalWrite(Led[3-i], LOW);
        }else{
            digitalWrite(Led[3-i], HIGH);
        }
    }

    printk("[led_dd] led_write\n");

    return len;
}
```

예제에서는 위와 같이 정의되어, 사용자 영역에서 buf라는 data를 가져와, Led를 제어하는데 쓰였습니다.

Release는 device file이 close될 때 호출되는 포인터입니다.

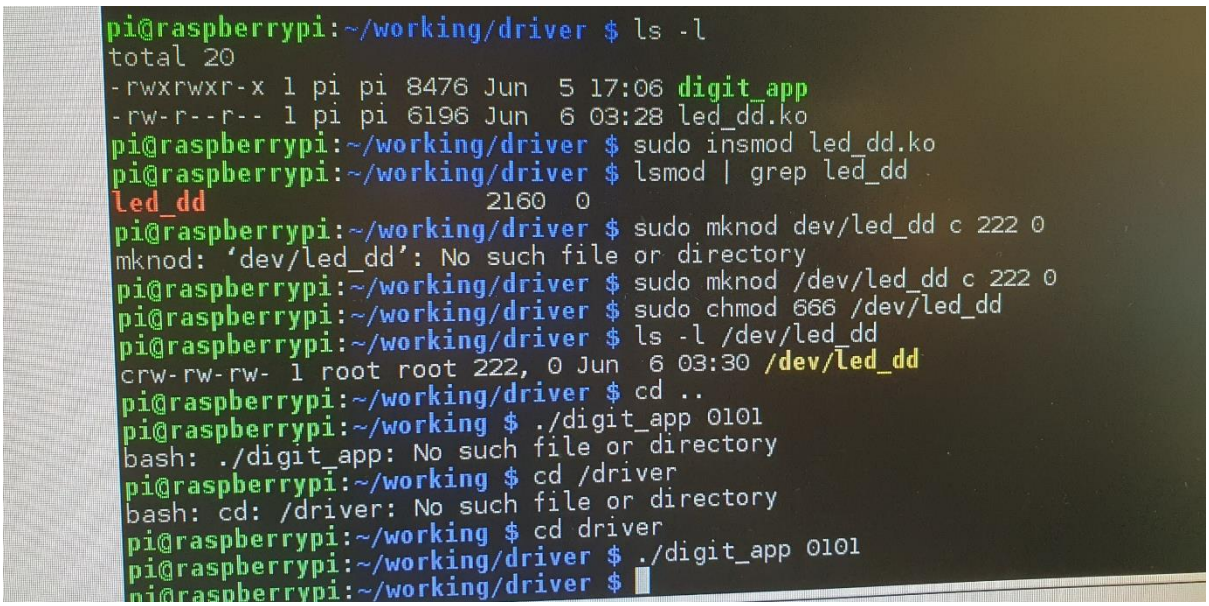
```
static int led_release(struct inode *inode, struct file *filp){
    iounmap(gpioAddr);

    printk("[led_dd] led_release\n");

    return 0;
}
```

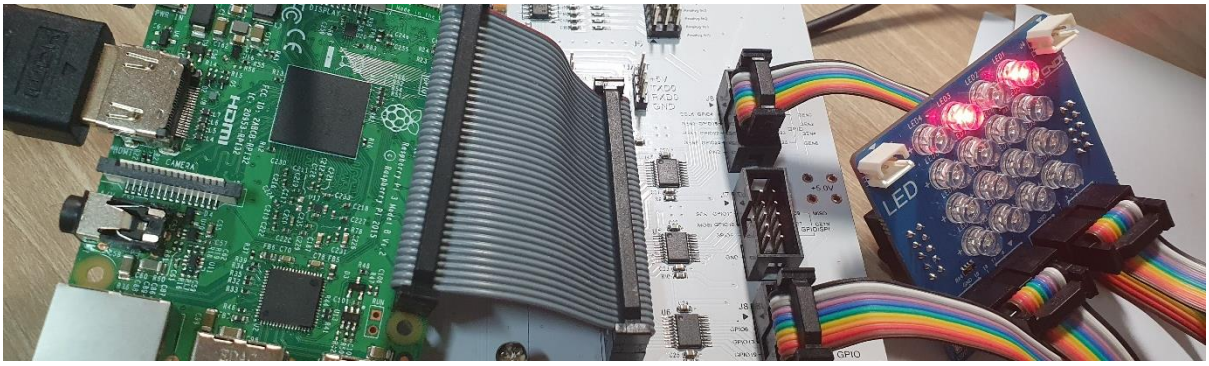
위와 같이 open에서 할당해준 data를 해제해줍니다.

이 외에도 여태까지 Library를 받아와썼던 pinMode와 digitalWrite같은 함수도 H/W와 사용자의 입력에 따라, GPFSEL, GPSET, GPCLR의 주소를 정의하여, 입력에 따라 변경시켜주어, led를 제어하는 방식으로 구현이 되었습니다.

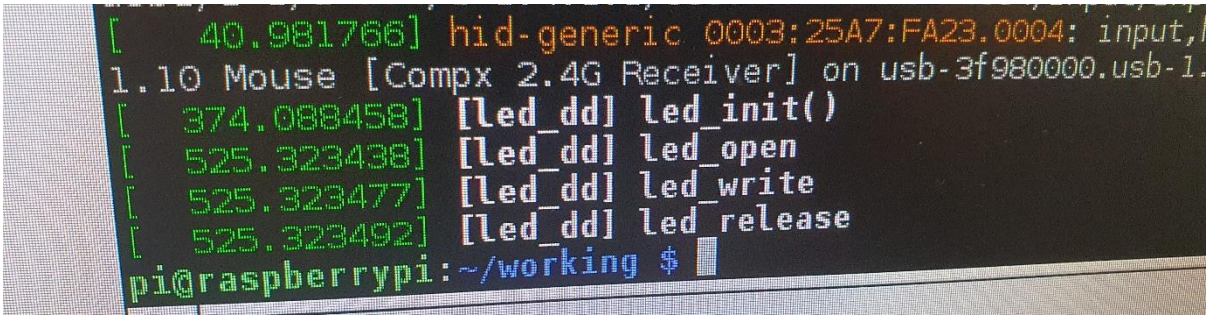
application에서는 open함수로 해당 DD를 열고, write함수로 입력 값을 DD에 써, LED를 제어하는 것을 확인할 수 있습니다.

Raspberry pi에서 해당 DD를 insmod를 통해 kernel에 적재하고, mknod를 통해, 사용자가 접근하기 위한 문자 파일을 만들고, rw가 허용되는 상태로 만듭니다. 그 다음 해당 application을 실행시켜보면

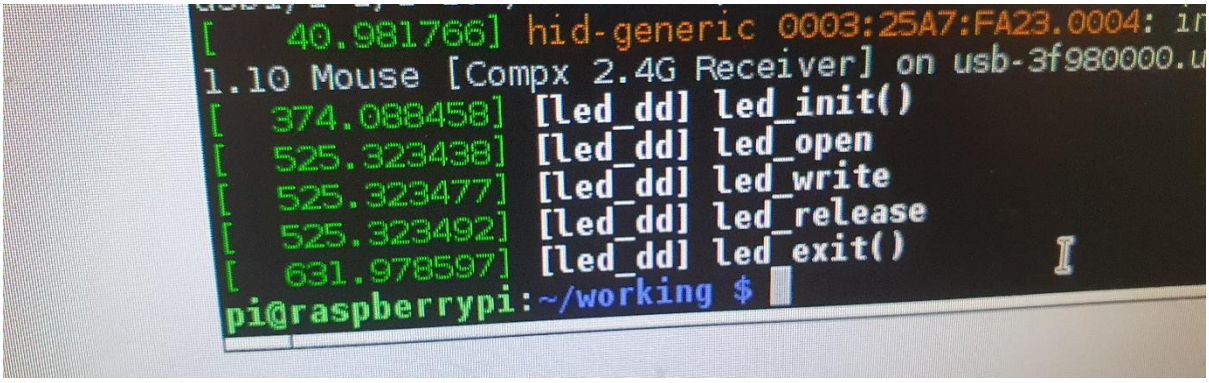




위와 같이 입력에 따라 device가 제어되는 것을 확인할 수 있습니다.



kernel에서도 init을 통해 등록, open통해 접근, write통해 제어 후 release를 통해 실행이 종료되는 것을 확인할 수 있습니다.



rmmod로 해당 DD가 kernel에서 제거되면, led_exit이 실행되어, kernel에서 제거되는 것을 확인할 수 있습니다.

2. 실습 과제

실습 과제는 ioctl() 함수를 추가하는 것이었습니다. ioctl은 driver 전용 파일 입출력 함수입니다. Read와 write함수를 대체하여 사용할 수도 있지만 과제에서는 led 색 제어에만 사용하였습니다.

Ioctl은 appliacation 부분에서는 ioctl(int fd, int request, char* arg)로 이루어집니다. fd에는 해당 file descriptor의 번호가 들어가고, request에는 command, arg에는 읽고 쓸 데이터의 주소값 헤드가 들어갑니다. Command는 4bytes로, 읽기 쓰기 명령 구분을 위한 2bits, arg를 통해 전달될 data의 크기인 14bits, 다른 driver와의 구분을 위한 magic number 8bits, ioctl의 명령을 구분하는 구분 번호 8bits로 이루어집니다. 과제에서는 1을 넣어주었지만, 따로 매크로 함수인 _IOW를 이용해서 device에 write를 하려 했지만 잘되지 않아, 추후에 추가적인 작업을 해주겠습니다. 우선 과제로 구현한 code를 보면,

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = led_open,
    .release = led_release,
    .write = led_write,
    .unlocked_ioctl = led_ioctl
};
```

우선 file_operations에 ioctl를 추가해주었습니다.

```
static int led_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int mode = 0;
    copy_from_user((void *)&mode, (void *)arg, sizeof(int));
    if(mode){
        ledColor = 11;
    }else {
        ledColor = 3;
    }

    printk("[led_dd2] led_ioctl\n");
}
```

해당하는 led_iotcl함수는 application에서 입력으로 받은 arg를 우선 kernel 영역으로 복사합니다. Arg는 기본적으로 주소 값이기 때문에 따로 ampersand를 넣어주지 않았습니다. 받은 mode값에 따라 제어할 led가 1~4, 8~12이기 때문에 해당하는 3과 11을 ledColor로 선언된 전역변수에 할당해주었습니다.

```
static int led_write(struct file *filp, const char *buf, size_t len, loff_t
*f_pos){
    int i;
    char state;

    for(i = 0; i < SIZEOFLED; i++){
        digitalWrite(Led[i], LOW);
    }

    for(i = 0; i < len; i++){
        copy_from_user(&state, &buf[i], 1);

        if(state == '0'){
            digitalWrite(Led[ledColor-i], LOW);
        }else{
            digitalWrite(Led[ledColor-i], HIGH);
        }
    }

    printk("[led_dd2] led_write\n");

    return len;
}
```

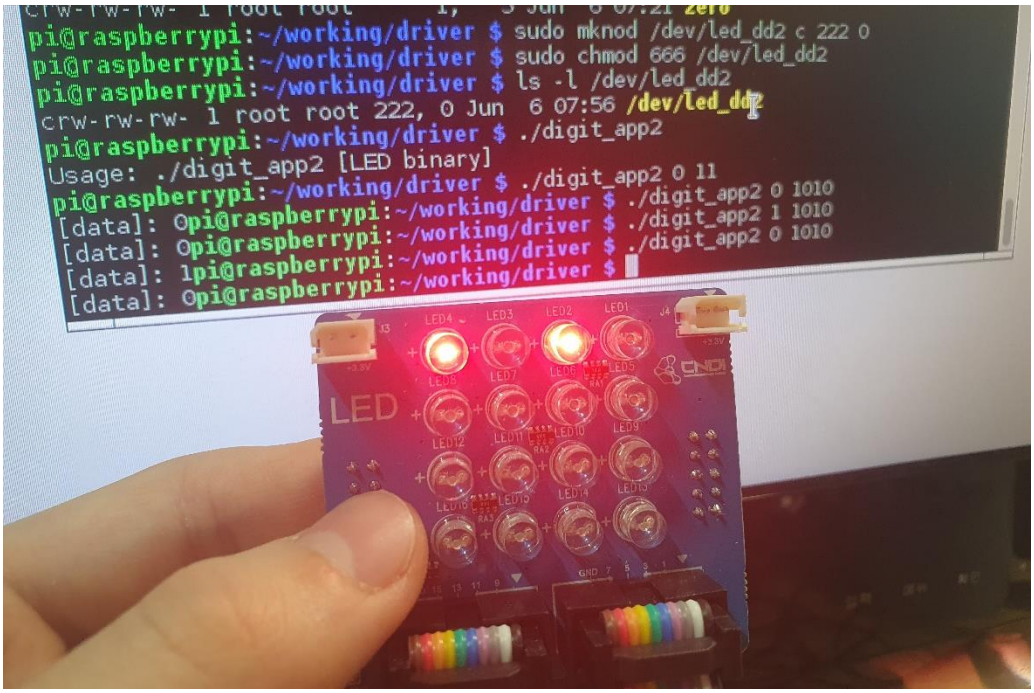
led_write함수는 우선 기본적으로 모든 led를 off 상태로 만들고, ioctl에서 정해주었던 ledColor부터 입력값에 따라 on과 off를 구현해주는 형식으로 예제에서 크게 바꾸지 않고 구현했습니다. 나머지 코드 부분도 예제와 다르지 않게 구현했습니다.

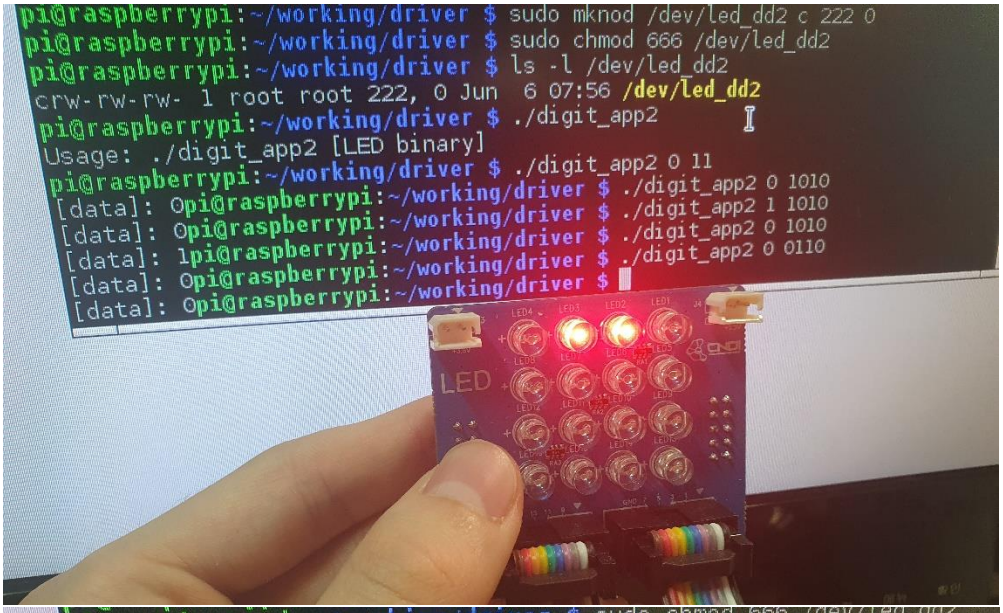
Application 부분은

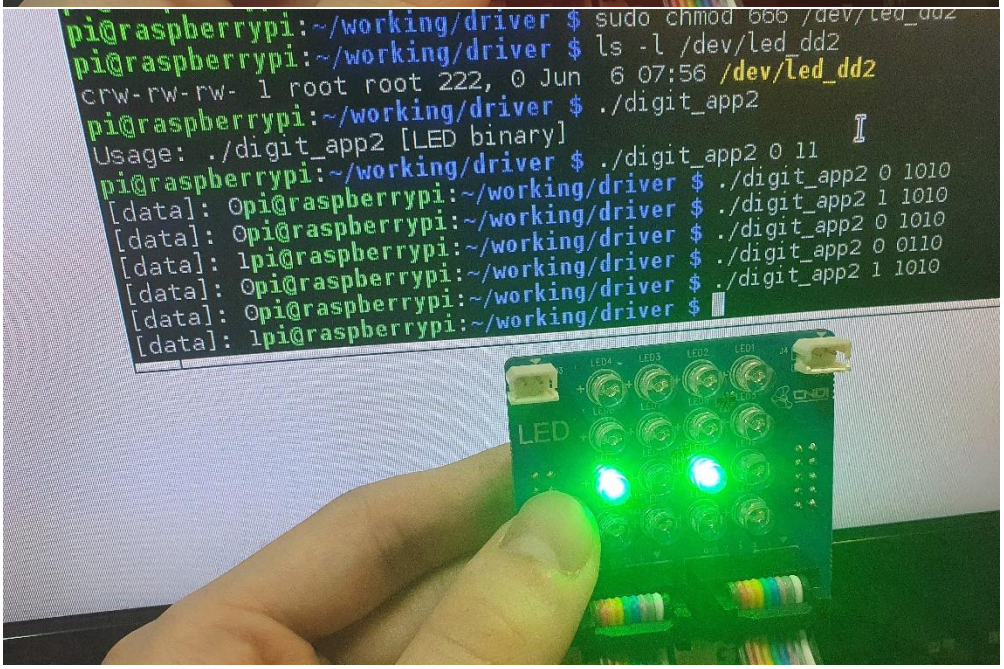
```
if(argc != 3){
    printf("Usage: %s [LED binary]\n", argv[0]);
    exit(1);
}

if((fd = open(_LED_PATH_, O_RDWR | O_NONBLOCK))<0){
    perror("open()");
    exit(1);
}
int data = atoi(argv[1]);

printf("[data]: %d", data);

ioctl(fd, 1, &data);

write(fd, argv[2], strlen(argv[2]));

close(fd);
```

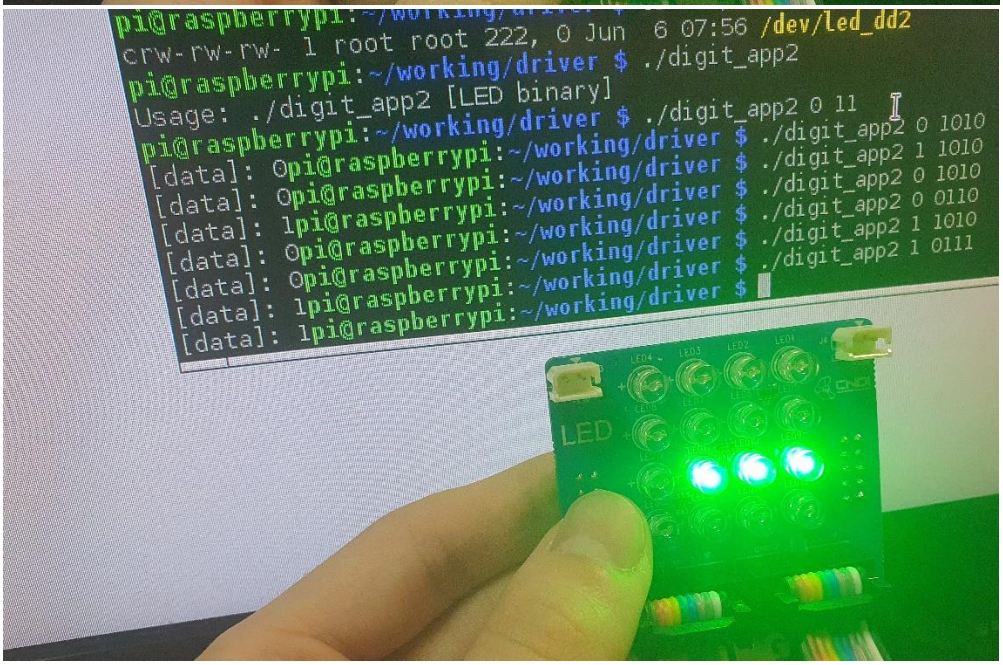
위와 같이 write이전에 led의 색을 정해주는 ioctl함수만 추가해주었습니다.

다음은 kernel에 module을 적재, file 생성 후 app을 구동시킨 결과입니다.

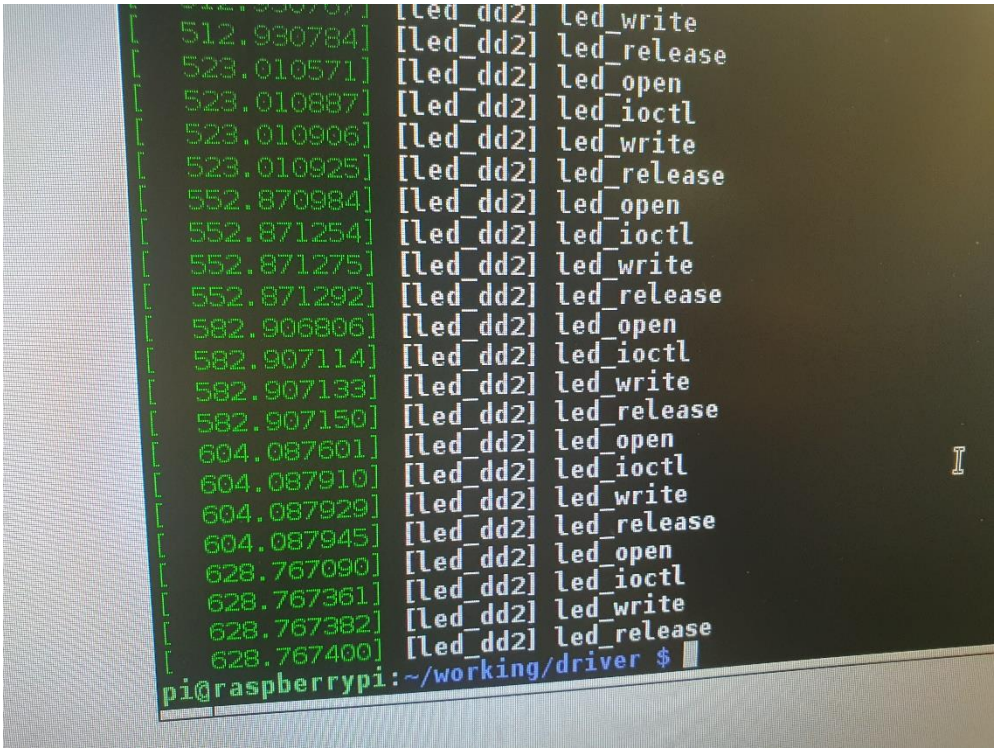








입력에 따라, led색과 켜지는 led 위치가 모두 잘 구동되는 것을 확인할 수 있습니다.



Kernel message를 통해 App의 구동에 따라 해당 DD가 잘 구동되는 것을 확인할 수 있었습니다.