1. 개요

라즈베리 파이에서 PWM을 이용하여 모터 속도를 제어하는 디바이스 드라이버를 작성하였습니다.

PWM을 구현하는 방식으로는 루프문을 통해 직접 GPIO 단자의 출력을 제어하는 방법과 /boot/config.txt 파일을 수정하여 GPIO 단자를 PWM 통신에 이용하도록 설정한 후, “linux/pwm.h” 모듈을 이용하여 제어하는 방법 두 가지가 있음을 검색을 통해 알게 되었습니다. 두 번째 방법은 제출하는 파일 외에 시스템 파일을 별도로 수정해야 하기 때문에 첫 번째 방법을 선택하였습니다.

라즈베리 파이의 GPIO 단자에 모터를 직접 연결하여 사용할 시 라즈베리 파이의 프로세서가 손상될 수 있으므로 아두이노의 스텝 모터 드라이버에서 ULN2003을 분리하여 외부 전원을 공급하는 용도로 사용하였습니다. 다른 장치를 이용하여 전원을 공급하려 했으나, 구하기가 힘들어 비교적으로 손상되어도 괜찮은 아두이노를 이용하여 전원을 공급하였습니다.

PWM 방식으로 모터를 제어하기 위해, 처음에는 duty 값을 입력 받아 on, off time을 계산하여 사용하고자 하였습니다. 하지만 정밀한 모터를 사용하지 않아 전압에 따라 일정하게 동작하지 않는 점, 비율을 계산할 때 실수를 사용할 수 없어 정수를 이용해 계산식을 작성하니 한 눈에 이해되지 않는 점 등의 문제가 있어 그냥 on, off time을 직접 입력하도록 구현하였습니다.

1. 프로그램 구조 설명
   1. 함수에 대한 설명
2. motor\_driver.c

static int motor\_thread(void\* args);

* PWM을 이용한 모터 속도 제어 쓰레드

static ssize\_t motor\_write(struct file\* instance, const char\* user\_buffer, size\_t count, loff\_t\* offs);

* 문자열 형태로 입력 받은 숫자를 변환하여 usec\_on\_min에 저장
* 인터럽트 발생 빈도를 줄이기 위해 입력한 간격의 두 배까지 마진 값 설정 (usec\_on\_max = usec\_on\_min \* 2)

static long int motor\_ioctl(struct file\* instance, unsigned cmd, unsigned long arg);

* 더 확장된 기능을 위한 함수
* motor\_write 함수를 이용하여 입력했던 usec\_on을 포함한 usec\_off, state 등 모든 변수 입력
* motor\_write 함수만 사용 시, 루프문을 빠져나올 수 없는 문제 해결하기 위한 state 변수 입력

static int \_\_init motor\_init(void);

* 장치 번호 할당
* 모듈 생성
* 장치 파일 생성
* 위의 과정들을 통해 mknod 단계 생략
* GPIO 출력 단자 설정

static void \_\_exit motor\_exit(void);

* 할당받은 자원 해제

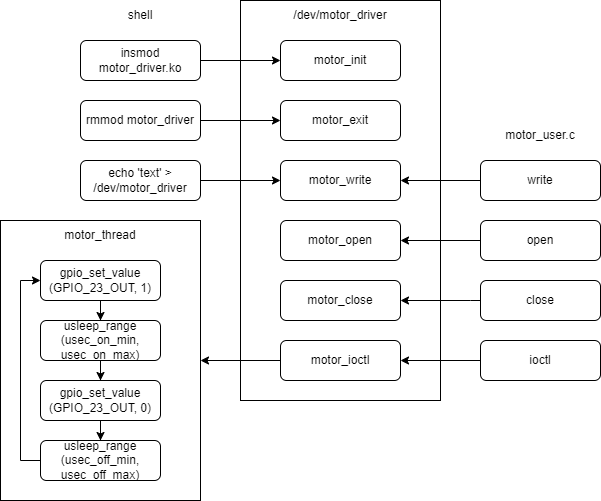
1. motor\_user.c

int main(int argc, char\* argv[]);

* 인자 하나만 입력 시, write 함수 이용하여 모터 속도 설정
* 인자 둘 입력 시, 첫 번째 인자를 통해 명령 설정, 두 번째 인자를 통해 값 설정
* ex) ./motor state 1, ./motor on 1000, ./motor off 500

1. cmd.h

* ioctl 함수를 위한 명령어 매크로
  1. 다이어그램



1. 실행 결과

실패 1)

텍스트이(가) 표시된 사진

자동 생성된 설명

이미 추상화되어 제공되고 있는 방식인 “linux/pwm.h” 모듈을 이용하여 구현하고자 하였습니다. 하지만 PWM 단자를 인식할 수 없는 문제가 생겼습니다. 좀 더 알아본 결과 /boot/config.txt 파일을 수정하여 GPIO 단자를 PWM 통신에 사용하겠다 미리 설정해줘야 한다는 것을 알게 되었습니다. 제출물 안에서 해결하기 위해 직접 루프문을 통해 PWM을 구현하기로 하였습니다.

실패 2)

텍스트이(가) 표시된 사진

자동 생성된 설명

컴파일, 모듈 삽입, 디바이스 파일 생성 등 다른 과정들은 문제없이 잘 진행되었으나 테스트를 위해 echo 명령어를 이용하여 문자열을 입력하자 권한 문제로 실행되지 않는 문제가 발생하였습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

확인해본 결과, 디바이스 파일은 루트 유저만 접근할 수 있도록 권한이 설정되어 있음을 확인하였습니다. chmod 명령어를 사용하여 권한을 변경한 후 다시 값을 입력하자 정상적으로 입력되는 것을 확인하였습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

sudo su 명령어를 이용하여 루트 계정으로 접속 후 실행하여도 역시 값이 잘 입력되는 것을 확인하였습니다.

실패 3)

텍스트이(가) 표시된 사진

자동 생성된 설명

모듈 삽입 후, 모터를 작동시키기 위해 값을 입력하자 쉘이 멈추는 문제가 발생하였습니다. 무한 루프를 이용하여 작성하였기 때문에 시스템에 무리가 가서 멈췄다고 짐작하였지만, 모터의 속도가 설정한 값으로 잘 동작하고 있었기 때문에 시스템은 잘 돌아가고 있다고 판단하였습니다. 그래서 쉘을 하나 더 실행시키자 문제없이 잘 실행되었습니다.

별도의 쓰레드를 구성하지 않고 드라이버 파일 내에서 함수를 통해 무한 루프를 실행하기 때문에, 다른 쉘에서 state 변수 값을 수정하여 루프를 멈추기 전까지는 모터 구동 함수가 반환되지 않아 쉘에 결과를 반환하지 못해서 문제가 발생하였습니다. “linux/kthread.h” 모듈을 불러와 루프문을 실행하기 위한 별도의 쓰레드를 구성하여 문제를 해결하였습니다.

실패 4)

motor\_write 함수를 이용하여 usec\_on\_min, usec\_on\_max 값을 설정하여 모터의 회전 속도를 조절할 수 있었습니다. 모터의 시동, 정지를 위한 state 값 설정, 모터의 power-off 시간을 설정하는 usec\_off\_min, usec\_off\_max 값 설정 등을 위해 write 값을 수정하려 하자 함수가 너무 복잡해지는 문제가 있었습니다.

설정해야 할 인자의 수가 많았기 때문에, motor\_write 함수는 power\_on 시간을 설정하는 기능 그대로 남겨두고 motor\_ioctl 함수를 정의하여 모든 인자들을 설정할 수 있도록 하였습니다. 명령어는 USEC\_ON, USEC\_OFF, STATE 세 가지를 정의하였고 각각 정수 하나를 인자로 사용하도록 하였습니다.

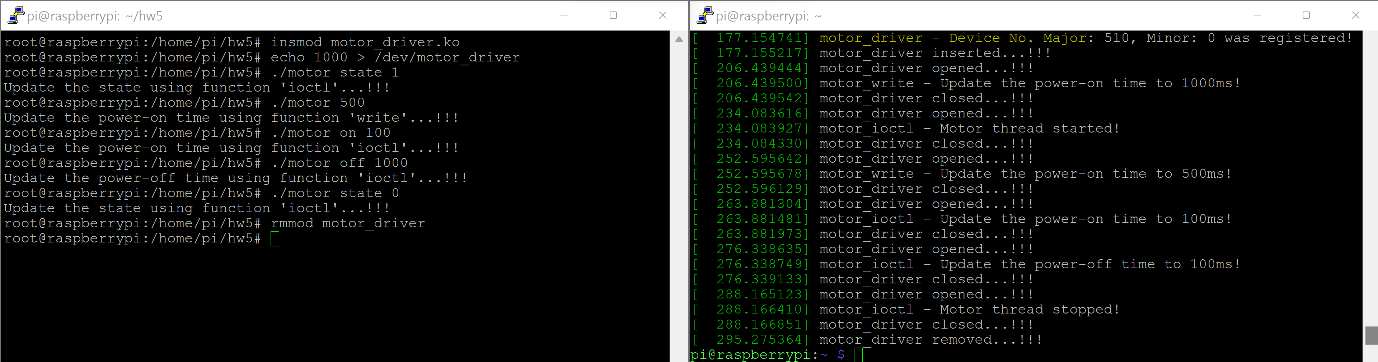
USEC\_ON, USEC\_OFF 명령어는 usleep\_range 함수에서 입력 받은 값을 대기 시간의 최솟값으로 사용하고 입력 받은 값의 두 배를 대기 시간의 최댓값으로 사용하였습니다. STATE 명령어는 처음에는 모터 구동 루프문을 종료하여 모터 동작 명령을 입력했던 쉘이 다시 동작하도록 하기 위해 state 변수를 수정하는데 사용하였지만, 쓰레드를 이용하여 구동하는 방식으로 바꾼 후에는 1 입력 시 쓰레드 시동, 0 입력 시 쓰레드를 종료하도록 변경하였습니다.

실패 5)

usleep\_range 함수에서 처음에는 참고했던 코드를 따라 min, max 값을 같은 값으로 입력하였습니다. 하지만 모터 구동 시 시스템이 심하게 느려져 usleep\_range 함수에 대해 알아본 결과 입력한 min, max 값 사이의 시간만큼 대기하고 만약 max 값이 지나도 다음 명령어를 실행하지 못한 경우, 인터럽트를 발생시켜 max 시간 내에 실행되도록 동작한다는 것을 알게 되었습니다.

따라서 인터럽트의 발생 빈도를 줄이기 위해 max 값을 기존의 값에서 입력한 값의 두 배만큼 변경해 주었습니다.

성공)



모듈 삽입 → 쉘에서 echo 명령어를 이용해 power-on time 설정 → 유저 모드 프로그램에서 ioctl 함수를 이용해 모터 구동 → 유저 모드 프로그램에서 write 함수를 이용해 power-on time 설정 → 유저 모드 프로그램에서 ioctl 함수를 이용해 power-on time 설정 → 유저 모드 프로그램에서 ioctl 함수를 이용해 power-off time 설정 → 유저 모드 프로그램에서 ioctl 함수를 이용해 모터 동작 정지 → 모듈 제거

회로 구성)

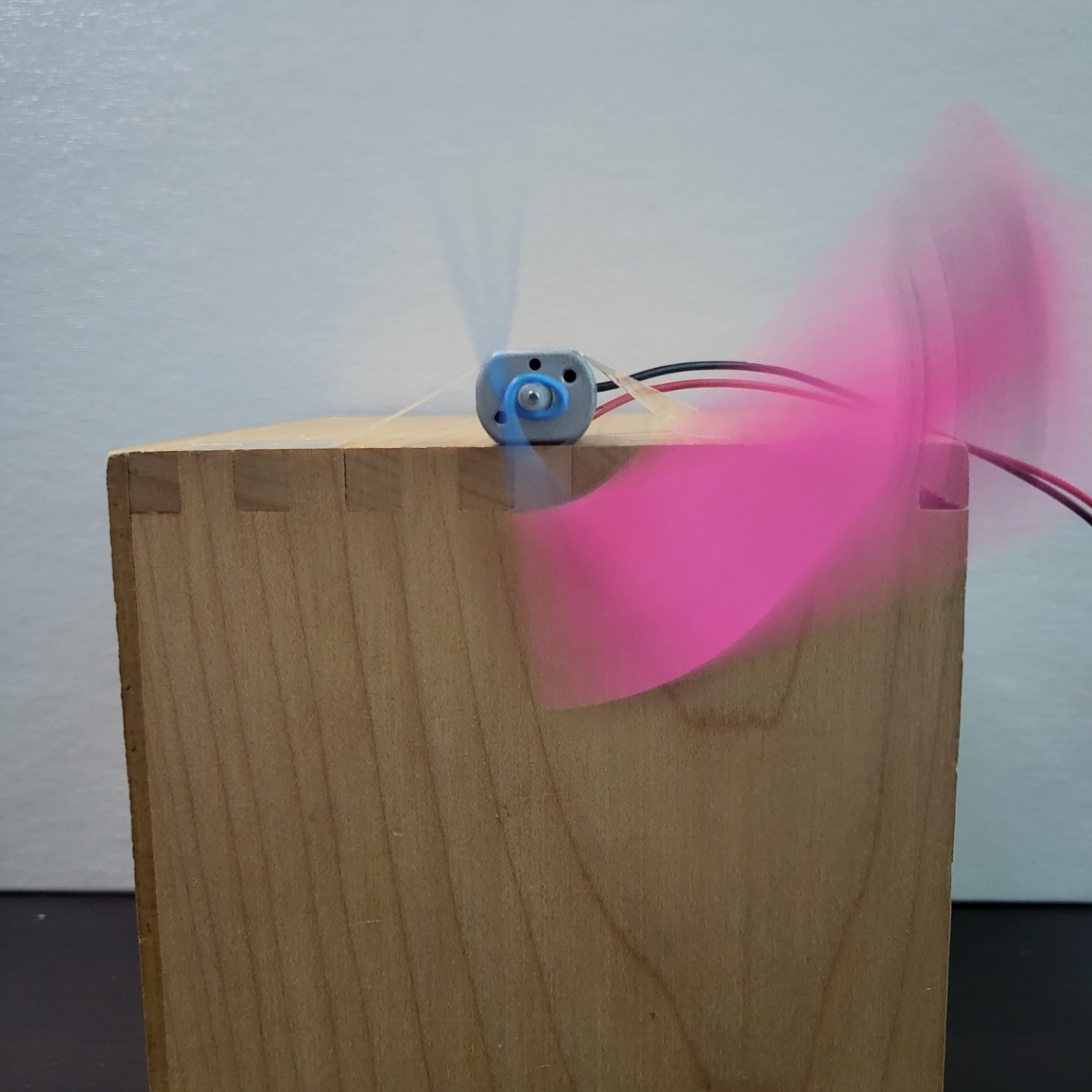
텍스트이(가) 표시된 사진

자동 생성된 설명

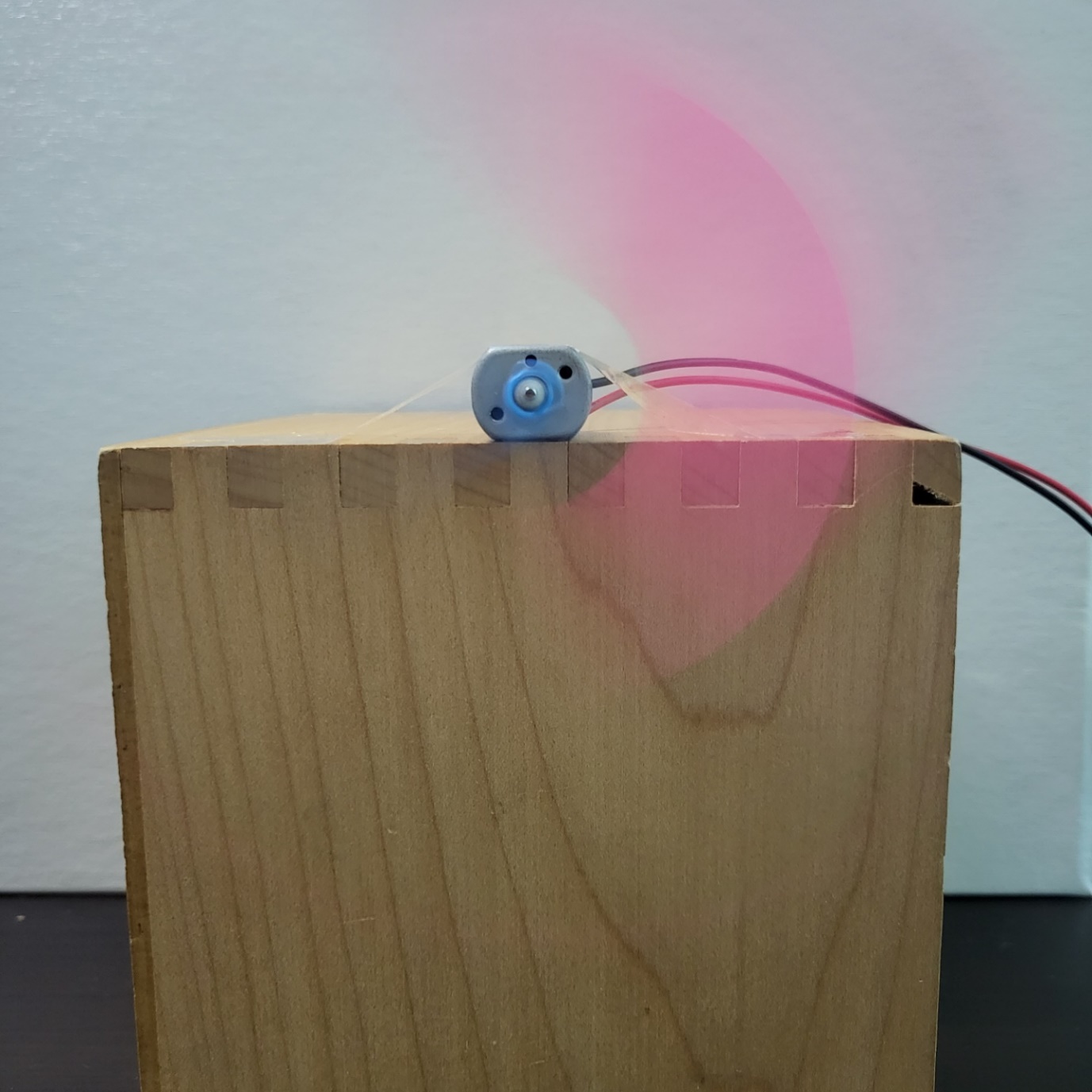
외부 전원으로 모터를 구동할 수 있도록, ULN2003을 이용해 회로 구성

Power-off time = 500us 고정)

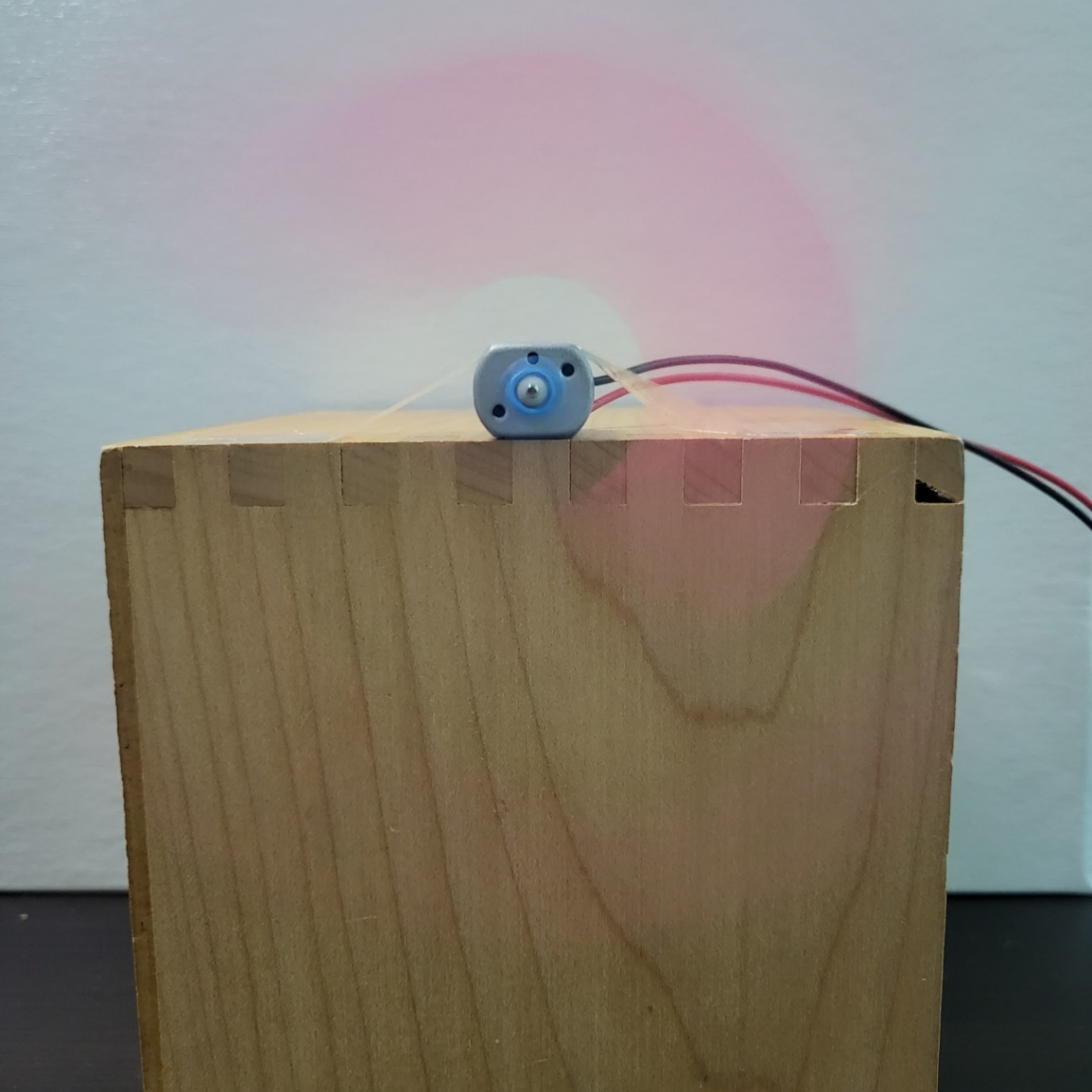
Power-on time = 100us)



Power-on time = 200us)

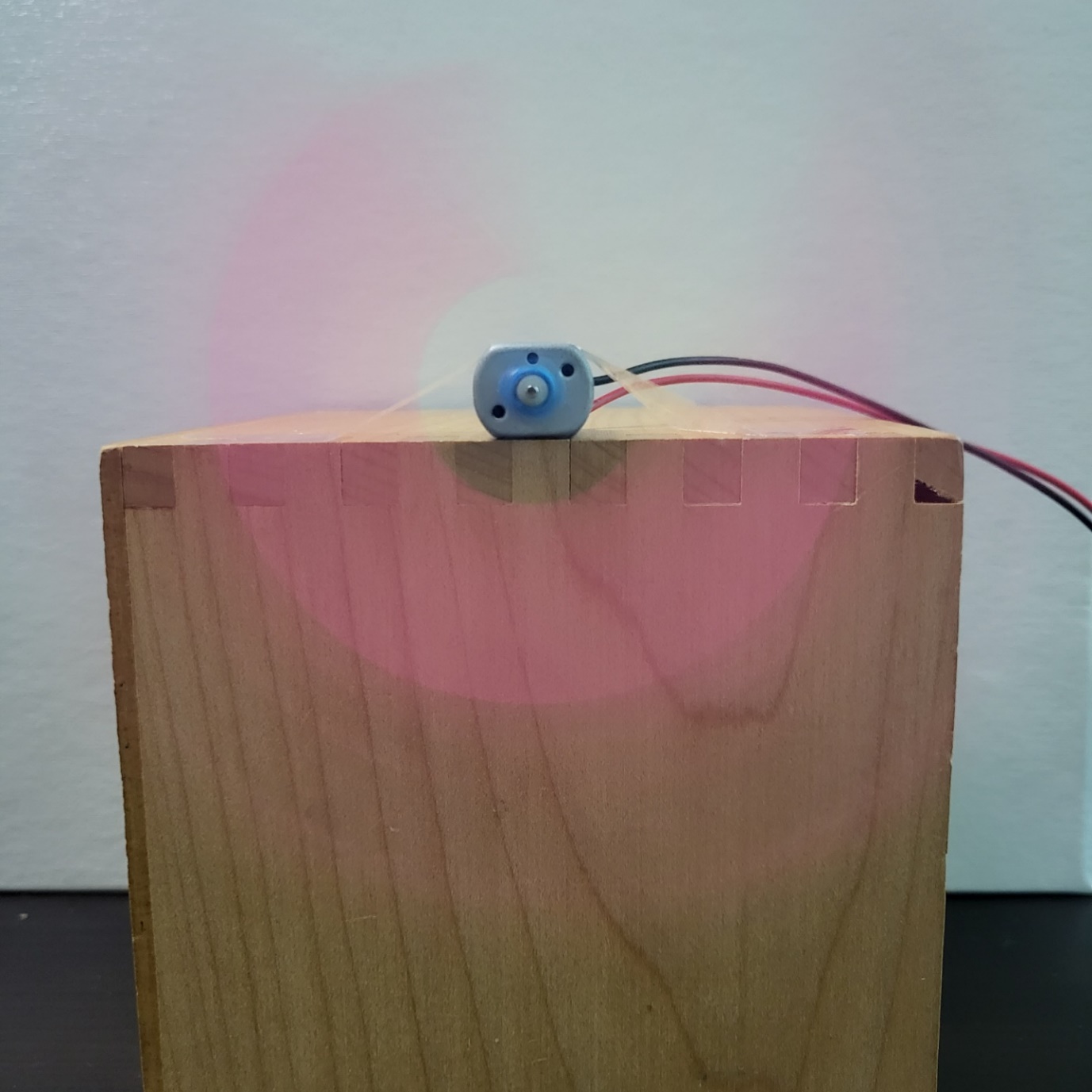


Power-on time = 300us ~ 1000us)



300us ~ 1000us 사진으로 구별 불가능

Power-on time = 2000us ~ 5000us)



2000us ~ 5000us 사진으로 구별 불가능

1. 고찰

커널 모드 프로그램을 작성하면서 제일 골치 아팠던 부분은, 실수로 루프문을 종료할 방법을 만들어 두지 않았을 때 종료할 방법이 없는 것이었습니다. 커널 모드에서 동작하기 때문에 ctrl + c를 입력하여 종료하는 방법도 사용할 수 없었습니다. 시스템이 다운되지는 않아 쉘을 추가로 실행하여 시스템 종료 명령을 입력하였으나, 루프문에서 매우 빠른 속도로 명령어를 실행시키자 시스템 종료 명령이 뒤로 밀려 작동하지 않았습니다. 시스템을 멈추기 위해, 전원선을 분리하는 방법 밖에 없었습니다.

저런 실수를 몇 번 겪으면서, 되도록이면 무한 루프를 사용하지 않고 만약 사용하게 된다면 반드시 종료하는 방법을 만들어 둬야 되겠다 생각하였습니다. 커널 쓰레드를 이용하는 방법으로도 해결할 수 있을 것 같습니다.

문자열로 입력 받은 숫자를 숫자 자료형으로 변환하기 위해, “stdlib.h”에 포함된 atoi 함수를 사용할 수 없으므로 변환 함수를 직접 구현하고자 하였습니다. 하지만 검색을 통해 확인해본 결과, 이미 커널 모드에서 동작하는 문자 → 숫자 변환 함수가 존재하였고 아직 과제를 해결하는 수준에서 사용하는 기본적인 함수들은 커널 모드에서 이미 구현되어 있음을 확인하였습니다.

지금까지 사용했던 기본적인 라이브러리 외에 아주 많은 종류의 유저 모드 라이브러리가 있다는 것을 알게 되었고, 유저 모드에서 함수 호출 시, 같은 동작을 하는 커널 모드 함수가 호출되는 방식으로 프로그램이 실행되는 것이 아닌가 짐작해 보았습니다.

1. 프로그램 소스 코드

motor\_driver.c

#include <linux/cdev.h>

#include <linux/delay.h>

#include <linux/fs.h>

#include <linux/gpio.h>

#include <linux/init.h>

#include <linux/ioctl.h>

#include <linux/kthread.h>

#include <linux/module.h>

#include <linux/uaccess.h>

#include "cmd.h"

/\* Meta Information \*/

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Minhee Park");

MODULE\_DESCRIPTION("A simple motor driver for controlling the motor speed using PWM");

/\* Buffer for data \*/

static char buffer[255];

/\* Variables for controlling the motor \*/

static int usec\_on\_min = 500;

static int usec\_on\_max = 1000;

static int usec\_off\_min = 500;

static int usec\_off\_max = 1000;

static int state = 0;

/\* Variables for kthread \*/

static struct task\_struct\* task;

#define KTHREAD\_NAME "motor\_thread"

/\* Variables for device and device class \*/

static dev\_t motor\_device\_no;

static struct class\* motor\_class;

static struct cdev motor\_device;

#define DRIVER\_NAME "motor\_driver"

#define DRIVER\_CLASS "MotorModuleClass"

#define GPIO\_23\_OUT 23

/\* This function is called, when trying to start the motor \*/

static int motor\_thread(void\* args)

{

while (1) {

gpio\_set\_value(GPIO\_23\_OUT, 1);

usleep\_range(usec\_on\_min, usec\_on\_max);

gpio\_set\_value(GPIO\_23\_OUT, 0);

usleep\_range(usec\_off\_min, usec\_off\_max);

if (kthread\_should\_stop()) {

break;

}

}

return 0;

}

/\*

\* This function is called, when user writes data to buffer

\*/

static ssize\_t motor\_write(struct file\* instance, const char\* user\_buffer, size\_t count, loff\_t\* offs)

{

int to\_copy, not\_copied, delta;

/\* Get amount of data to copy \*/

to\_copy = min(count, sizeof(buffer));

/\* Copy data from user \*/

not\_copied = copy\_from\_user(buffer, user\_buffer, to\_copy);

/\* Set the power-on time in microseconds \*/

usec\_on\_min = (int)simple\_strtol(buffer, NULL, 10);

usec\_on\_max = usec\_on\_min \* 2;

pr\_info("motor\_write - Update the power-on time to %dms!\n", usec\_on\_min);

/\* Calculate data \*/

delta = to\_copy - not\_copied;

return delta;

}

/\*

\* This function is called, when the device file is opened

\*/

static int motor\_open(struct inode\* device\_file, struct file\* instance)

{

pr\_info("%s opened...!!!\n", DRIVER\_NAME);

return 0;

}

/\*

\* This function is called, when the device file is closed

\*/

static int motor\_close(struct inode\* device\_file, struct file\* instance)

{

pr\_info("%s closed...!!!\n", DRIVER\_NAME);

return 0;

}

/\*

\* This function is called, when the user calls ioctl in user mode

\*/

static long int motor\_ioctl(struct file\* instance, unsigned cmd, unsigned long arg)

{

switch (cmd) {

case USEC\_ON:

copy\_from\_user(&usec\_on\_min, (int)arg, sizeof(usec\_on\_min));

usec\_on\_max = usec\_on\_min \* 2;

pr\_info("motor\_ioctl - Update the power-on time to %dms!\n", usec\_on\_min);

break;

case USEC\_OFF:

copy\_from\_user(&usec\_off\_min, (int)arg, sizeof(usec\_off\_min));

usec\_off\_max = usec\_off\_min \* 2;

pr\_info("motor\_ioctl - Update the power-off time to %dms!\n", usec\_on\_min);

break;

case STATE:

copy\_from\_user(&state, (int)arg, sizeof(state));

if (state == 1) {

task = kthread\_run(motor\_thread, NULL, KTHREAD\_NAME);

pr\_info("motor\_ioctl - Motor thread started!\n");

}

else {

kthread\_stop(task);

pr\_info("motor\_ioctl - Motor thread stopped!\n");

}

}

return 0;

}

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = motor\_open,

.release = motor\_close,

.write = motor\_write,

.unlocked\_ioctl = motor\_ioctl

};

/\*

\* This function is called, when the module is loaded into the kernel

\*/

static int \_\_init motor\_init(void)

{

/\* Allocate device number \*/

if (alloc\_chrdev\_region(&motor\_device\_no, 0, 1, DRIVER\_NAME) < 0) {

pr\_err("Unable to allocate device number!\n");

return -1;

}

pr\_info("%s - Device No. Major: %d, Minor: %d was registered!\n", DRIVER\_NAME, MAJOR(motor\_device\_no), MINOR(motor\_device\_no));

/\* Create device class \*/

if ((motor\_class = class\_create(THIS\_MODULE, DRIVER\_CLASS)) == NULL) {

pr\_err("Unable to create device class!\n");

goto ClassError;

}

/\* Create device file \*/

if (device\_create(motor\_class, NULL, motor\_device\_no, NULL, DRIVER\_NAME) == NULL) {

pr\_err("Unable to create device file!\n");

goto FileError;

}

/\* Initialize device \*/

cdev\_init(&motor\_device, &fops);

/\* Add device to kernel \*/

if (cdev\_add(&motor\_device, motor\_device\_no, 1) < 0) {

pr\_err("Unable to add device to kernel!\n");

goto AddError;

}

/\* Request GPIO 23 \*/

if (gpio\_request(GPIO\_23\_OUT, "GPIO\_23\_OUT") < 0) {

pr\_err("Unable to request GPIO %d!\n", GPIO\_23\_OUT);

goto RequestError;

}

/\* Set GPIO 23 direction \*/

if (gpio\_direction\_output(GPIO\_23\_OUT, 0) < 0) {

pr\_err("Unable to set GPIO %d to output!\n", GPIO\_23\_OUT);

goto DirectionError;

}

pr\_info("%s inserted...!!!\n", DRIVER\_NAME);

return 0;

DirectionError:

gpio\_free(GPIO\_23\_OUT);

RequestError:

cdev\_del(&motor\_device);

AddError:

device\_destroy(motor\_class, motor\_device\_no);

FileError:

class\_destroy(motor\_class);

ClassError:

unregister\_chrdev\_region(motor\_device\_no, 1);

return -1;

}

/\*

\* This function is called, when the module is removed from the kernel

\*/

static void \_\_exit motor\_exit(void)

{

gpio\_set\_value(GPIO\_23\_OUT, 0);

gpio\_free(GPIO\_23\_OUT);

cdev\_del(&motor\_device);

device\_destroy(motor\_class, motor\_device\_no);

class\_destroy(motor\_class);

unregister\_chrdev\_region(motor\_device\_no, 1);

pr\_info("%s removed...!!!\n", DRIVER\_NAME);

}

module\_init(motor\_init);

module\_exit(motor\_exit);

motor\_user.c

#include <sys/ioctl.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include "cmd.h"

int main(int argc, char\* argv[])

{

int fd = open("/dev/motor\_driver", O\_WRONLY);

int value;

if (fd < 0) {

printf("Unable to open device!\n");

return -1;

}

if (argc == 2) {

write(fd, argv[1], sizeof(argv[1]));

printf("Update the power-on time using function 'write'...!!!\n");

}

else {

value = atoi(argv[2]);

if (strcmp(argv[1], "on") == 0) {

ioctl(fd, USEC\_ON, &value);

printf("Update the power-on time using function 'ioctl'...!!!\n");

}

else if (strcmp(argv[1], "off") == 0) {

ioctl(fd, USEC\_OFF, &value);

printf("Update the power-off time using function 'ioctl'...!!!\n");

}

else if (strcmp(argv[1], "state") == 0) {

ioctl(fd, STATE, &value);

printf("Update the state using function 'ioctl'...!!!\n");

}

}

close(fd);

return 0;

}

cmd.h

#ifndef \_\_CMD\_H\_\_

#define \_\_CMD\_H\_\_

#define USEC\_ON \_IOW('a', 'a', int\*)

#define USEC\_OFF \_IOW('a', 'b', int\*)

#define STATE \_IOW('a', 'c', int\*)

#endif

1. 자료 출처

“Let's code a Linux Driver - 4: GPIO driver”, Johannes 4GNU\_Linux, 2020. 12. 24, <https://www.youtube.com/watch?v=P3Judr4i7QI>

“Let's code a Linux Driver - 6: PWM Driver”, Johannes 4GNU\_Linux, 2020. 12. 24, <https://www.youtube.com/watch?v=yLm4EDVNceo>

“Let's code a Linux Driver - 13: IOCtl in a Linux Kernel Module”, Johannes 4GNU\_Linux, 2021. 5. 27, <https://www.youtube.com/watch?v=j331b_xS4NY>

“Bit Banging PWM Driver using a Kernel Module on Raspberry Pi”, Arri, 2018. 9. 15, <https://www.manmade2.com/bit-banging-pwm-driver-using-a-kernel-module-on-raspberry-pi/>

“PWM Driver built on Threads in Linux Kernel Module on Raspberry Pi”, Arri, 2018. 9. 16, <https://www.manmade2.com/pwm-driver-built-on-threads-in-linux-kernel-module-on-raspberry-pi/>

“[Arduino] ULN2004(ULN2003)를 이용한 DC 모터(DC Motor) 제어”, 당근천국, 2014. 12. 13, <https://blog.danggun.net/2647>