```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>


#define NSEC_OVERFLOW_BORDER 999999999
#define SEC_IN_NSEC 1000000000


void *led_blinkenrasp();
int mygpio_write(int value);
int mygpio_read();


int nssleep(long nsec);

int main()
{
    int trigger_temp = 1;
    pthread_t led_blinkenrasp_thread;
    int led_blinkenrasp_running = 0;


    FILE *mygpio = NULL;
    mygpio_write(1);


    while(1) {
        char read_text[10];
        size_t read_count;
        int fclose_ret = 0;

        mygpio = fopen("/dev/mygpio", "r");
        if (mygpio == NULL) {
            printf("failed to open gpio");
            return -1;
        }
        read_count = fread(&read_text, sizeof(char), 2, mygpio);
        if(read_count > 0 ) {
            int trigger_state = atoi(read_text);
            if (trigger_temp == 0 && trigger_state == 1)  {
                if (!led_blinkenrasp_running) {
                    if(pthread_create(&led_blinkenrasp_thread, NULL, led_blinkenrasp
, NULL) != 0) {
                        printf("creation of blinkenrasp thread failed");
                    }
                    led_blinkenrasp_running = 1;
                } else {
                    pthread_cancel(led_blinkenrasp_thread);
                    pthread_join(led_blinkenrasp_thread, NULL);
                    led_blinkenrasp_running = 0;
                    mygpio_write(1);
                    printf("stoped blinking\n");

                }

            }
            trigger_temp = atoi(read_text);
        }
        fclose_ret = fclose(mygpio);
        if (fclose_ret == EOF) {
            return -1;
        }

    }
    return 0;
}
```

```c
void *led_blinkenrasp() {


    printf("for reference switching led to on\n");
    // on
    mygpio_write(0);
    sleep(2);



    printf("Starting to blink with 250HZ\n");
    while(1) {
        // on
        mygpio_write(0);
        nssleep(4000000);

        // off
        mygpio_write(1);
        nssleep(4000000);

    }
    mygpio_write(1);
    return 0;
}

int mygpio_write(int value){
    int write_count;
    int fclose_ret;
    char data[2];
    FILE *mygpio = fopen("/dev/mygpio", "w");
    if (mygpio == NULL) {
        printf("failed to open gpio");
        return -1;
    }
    int size = sprintf(data,"%d",value);

    write_count = fwrite(&data, sizeof(char), size, mygpio);
    if (write_count > 0 ) {
        ;
    }

    fclose_ret = fclose(mygpio);
    if (fclose_ret == EOF) {
        return -1;
    }

    return 1;

}
int mygpio_read(){
    int read_count;
    int fclose_ret;
    int read_int;
    FILE *mygpio = fopen("/dev/mygpio", "r");
    if (mygpio == NULL) {
        printf("failed to open gpio");
        return -1;
    }

    read_count = fread(&read_int, sizeof(int), 1, mygpio);
    if (read_count > 0 ) {
        ;
    }

    fclose_ret = fclose(mygpio);
    if (fclose_ret == EOF) {
    }

    return 1;
}
```

```c
int nssleep(long nsec) {
    struct timespec sleeptime;

    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = nsec;


    while (clock_nanosleep(CLOCK_MONOTONIC, 0, &sleeptime, NULL) == EINTR );

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>


#define GPIO_17 "17"
#define GPIO_18 "18"
#define GPIO_IN "in"
#define GPIO_OUT "out"
#define NSEC_OVERFLOW_BORDER 999999999
#define SEC_IN_NSEC 1000000000

void sigint_handler(int);

void *led_blinkenrasp(void *);

int gpio_export_port(char *port);
int gpio_unexport_port(char *port);
int gpio_set_direction(char *port, char *direction);
int gpio_set_value(char *port, int value);

int nssleep(long nsec);

int main()
{
    printf("activating sighandler\n");
    struct sigaction  sa;

    int trigger_temp = 1;
    pthread_t led_blinkenrasp_thread;
    int led_blinkenrasp_running = 0;


    FILE *gpio17 = NULL;

    sigemptyset(&sa.sa_mask);
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    gpio_export_port(GPIO_17);
    gpio_export_port(GPIO_18);
    printf("setting gpio 17 to in\n");
    gpio_set_direction(GPIO_17, GPIO_IN);
    printf("setting gpio 18 to out\n");
    gpio_set_direction(GPIO_18, GPIO_OUT);
    gpio_set_value(GPIO_18, 1);


    while(1) {
        char read_text[10];
        size_t read_count;
        int fclose_ret = 0;

        gpio17 = fopen("/sys/class/gpio/gpio17/value", "r");
        if (gpio17 == NULL) {
            printf("failed to open gpio");
            return -1;
        }
        read_count = fread(&read_text, sizeof(char), 2, gpio17);
        if(read_count > 0 ) {
            int trigger_state = atoi(read_text);
            if (trigger_temp == 0 && trigger_state == 1)  {
                if (!led_blinkenrasp_running) {
                    if(pthread_create(&led_blinkenrasp_thread, NULL, led_blinkenrasp
, NULL) != 0) {
                        printf("creation of blinkenrasp thread failed");
                    }
```

```
                    led_blinkenrasp_running = 1;
                    printf("started blinking \n");
                } else {
                    pthread_cancel(led_blinkenrasp_thread);
                    pthread_join(led_blinkenrasp_thread, NULL);
                    gpio_set_value(GPIO_18, 1);
                    led_blinkenrasp_running = 0;
                    printf("stoped blinking\n");

                }

            }
            trigger_temp = atoi(read_text);
        }
        fclose_ret = fclose(gpio17);
        if (fclose_ret == EOF) {
            return -1;
        }

    }
    return 0;
}



void *led_blinkenrasp(void * data) {
    while(1) {
        // on
        gpio_set_value(GPIO_18, 0);
        nssleep(100000000);

        // off
        gpio_set_value(GPIO_18, 1);
        nssleep(100000000);

    }
}

void sigint_handler(int signum) {
    printf("Programm interrupted with signum: %d\n", signum);
    gpio_set_value(GPIO_18, 1);
    gpio_unexport_port(GPIO_17);
    gpio_unexport_port(GPIO_18);
    exit(0);
}


int gpio_set_value(char *port, int value){
    FILE *gpio_port = NULL;
    size_t write_count;
    int fclose_ret = 0;
    char value_string_path[100];
    char int_string_value[4];
    sprintf(int_string_value, "%d", value);
    sprintf(value_string_path, "/sys/class/gpio/gpio%s/value", port);
    gpio_port = fopen(value_string_path, "w");
    if (gpio_port == NULL) {
        printf("failed get write permission to set port %s's value\n", port);
        return -1;
    }
    write_count = fwrite(&int_string_value, sizeof(char), 1, gpio_port);
    if (write_count > 0 ) {
        ;
    }

    fclose_ret = fclose(gpio_port);
    if (fclose_ret == EOF) {
        printf("failed to close value file\n");
        return -1;
    }

    return 1;
}
```

```c
int gpio_set_direction(char *port, char *direction) {
    FILE *gpio_port = NULL;
    size_t write_count;
    int fclose_ret = 0;
    char direction_string_path[100];
    sprintf(direction_string_path, "/sys/class/gpio/gpio%s/direction", port);
    gpio_port = fopen(direction_string_path, "w");
    if (gpio_port == NULL) {
        printf("failed get write permission to set port %s's direction", port);
        return -1;
    }
    write_count = fwrite(direction, sizeof(char), strlen(direction) + 1, gpio_port);
    if (write_count > 0 ) {
        ;
    }

    fclose_ret = fclose(gpio_port);
    if (fclose_ret == EOF) {
        return -1;
    }

    return 1;

}
int gpio_export_port(char *port) {
    FILE *export = NULL;
    size_t write_count;
    int fclose_ret = 0;
    export = fopen("/sys/class/gpio/export", "w");
    if (export == NULL) {
        printf("failed get write permission to export port %s", port);
        return -1;
    }
    write_count = fwrite(port, sizeof(char), strlen(port) + 1, export);
    if (write_count > 0 ) {
        ;
    }

    fclose_ret = fclose(export);
    if (fclose_ret == EOF) {
        return -1;
    }

    return 1;
}

int gpio_unexport_port(char *port) {
    FILE *unexport = NULL;
    size_t write_count;
    int fclose_ret = 0;
    unexport = fopen("/sys/class/gpio/unexport", "w");
    if (unexport == NULL) {
        printf("failed get write permission to unexport port %s", port);
        return -1;
    }
    write_count = fwrite(port, sizeof(char), strlen(port) + 1, unexport);
    if (write_count > 0 ) {
        ;
    }

    fclose_ret = fclose(unexport);
    if (fclose_ret == EOF) {
        return -1;
    }

    return 1;
}

int nssleep(long nsec) {
    struct timespec sleeptime;
```

```
    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = nsec;


    while (clock_nanosleep(CLOCK_MONOTONIC, 0, &sleeptime, NULL) == EINTR );

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <sched.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <math.h>


#define NSEC_OVERFLOW_BORDER 999999999
#define SEC_IN_NSEC 1000000000
#define HIGH "1"
#define LOW "0"

long long blinken_states[] = {
    100000000,  // 5 HZ
    50000000,   // 10 HZ
    25000000,   // 20 HZ
    10000000,   // 50 HZ
    5000000,    // 100 HZ
    2000000,    // 250 HZ
};

enum { HZ_5, HZ_10, HZ_20, HZ_50, HZ_100, HZ_250, OFF};



void *led_blinkenrasp(void *);
int mygpio_write(char* value);
int mygpio_read();
void blink(long long sleeptime);
int frequencyToHZ(long long frq);

void sigint_handler(int signum);


int nssleep(long nsec);

int main()
{
    int trigger_temp = 1;
    pthread_t led_blinkenrasp_thread;
    int selected_state = 0;

    FILE *mygpio = NULL;

    mygpio_write(HIGH);


    struct sigaction  sa;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    int blinkenrasp_running = 0;




    while(1) {
        char read_text[10];
        size_t read_count;
        int fclose_ret = 0;

        mygpio = fopen("/dev/mygpio", "r");
        if (mygpio == NULL) {
```

```
            printf("failed to open gpio");
            return -1;
        }
        read_count = fread(&read_text, sizeof(char), 2, mygpio);
        if(read_count > 0 ) {
            int trigger_state = atoi(read_text);
            if (trigger_temp == 0 && trigger_state == 1)  {
                if(selected_state != OFF) {
                    if (blinkenrasp_running){
                        pthread_cancel(led_blinkenrasp_thread);
                        pthread_join(led_blinkenrasp_thread, NULL);
                    }

                    long long data =  blinken_states[selected_state];
                    if(pthread_create(&led_blinkenrasp_thread, NULL, led_blinkenrasp
, (void *) &data) != 0) {
                        printf("creation of blinkenrasp thread failed with frequency
 %lld", data);
                    }
                    blinkenrasp_running=1;
                    selected_state++;

                } else {
                    pthread_cancel(led_blinkenrasp_thread);
                    pthread_join(led_blinkenrasp_thread, NULL);
                    mygpio_write(HIGH);
                    printf("stoped blinking\n");
                    selected_state = 0;
                    blinkenrasp_running=0;

                }
            }
            trigger_temp = atoi(read_text);
        }
        fclose_ret = fclose(mygpio);
        if (fclose_ret == EOF) {
            return -1;
        }

    }
    return 0;
}



void *led_blinkenrasp(void * data) {

    long long frequency = *(long long  *)data;

    printf("starting of blinkenrasp thread with %dHZ\n", frequencyToHZ(frequency*2))
;

    while(1) {
        blink(frequency);
    }

    return 0;
}

int frequencyToHZ(long long frq) {

    return (1/(frq * pow(10, -9))) + 1;

}

void sigint_handler(int signum) {
    printf("Programm interrupted with signum: %d\n", signum);
    mygpio_write(HIGH);
    exit(0);
}

void blink(long long sleeptime) {
    // on
```

```
    mygpio_write(LOW);
    nssleep(sleeptime);

    // off
    mygpio_write(HIGH);
    nssleep(sleeptime);
}


int mygpio_write(char *value){
    int led_file = open("/dev/mygpio", O_NONBLOCK | O_RDWR);

    write(led_file, value, 2);

    if (close(led_file) != 0) {
    }

    return 1;

}

int mygpio_read(){
    int read_count;
    int fclose_ret;
    int read_int;
    int mygpio = open("/dev/mygpio", O_NONBLOCK | O_RDWR);
    if (mygpio == -1) {
        printf("failed to open gpio");
        return -1;
    }

    read_count = read(mygpio, &read_int, sizeof(int));
    if (read_count > 0 ) {
        ;
    }

    fclose_ret = close(mygpio);
    if (fclose_ret == 0) {
    }

    return 1;
}


int nssleep(long nsec) {
    struct timespec sleeptime;

    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = nsec;


    while (clock_nanosleep(CLOCK_MONOTONIC, 0, &sleeptime, NULL) == EINTR );

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <sched.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>


#define NSEC_OVERFLOW_BORDER 999999999
#define SEC_IN_NSEC 1000000000
#define LOOPS 100000


int mygpio_write(char *value);
int mygpio_read();
struct timespec diff(struct timespec, struct timespec);


int nssleep(long nsec);

int main()
{


    long long nsec_average = 0;
    mygpio_write("1");

    pid_t pod = getpid();
    struct sched_param param;

    sched_setscheduler(pod, SCHED_FIFO, &param);

    int i = 0;

    struct timespec res_start, res_stop, diff_stamp;

    for(i = 0; i < LOOPS; i++) {
        clock_gettime(CLOCK_REALTIME, &res_start);
        mygpio_write("0");
        mygpio_write("1");
        clock_gettime(CLOCK_REALTIME, &res_stop);


        diff_stamp = diff(res_start, res_stop);

        //printf("diff_time = %lld.%.9ld, average = %ld\n", (long long) diff_stamp.t
v_sec, diff_stamp.tv_nsec, nsec_average);

        if (nsec_average == 0 ) {
            nsec_average = diff_stamp.tv_nsec;
        } else {
            nsec_average = (nsec_average + diff_stamp.tv_nsec) / 2;
        }


    }

    double nseconds = (double)nsec_average * pow(10, -9);
    printf("Nsec Avarage = %lld 10^-9=%lf nseconds =%lf\n", nsec_average, pow(10,-9)
, nseconds);


    printf("kHZ = %lf\n", (1.0 / nseconds) / 1000.0);


    return 0;
```

```
}



int mygpio_write(char *value){
    int led_file = open("/dev/mygpio", O_NONBLOCK | O_RDWR);

    write(led_file, value, 2);

    if (close(led_file) != 0) {
    }

    return 1;

}
int mygpio_read(){
    int read_count;
    int fclose_ret;
    int read_int;
    FILE *mygpio = fopen("/dev/mygpio", "r");
    if (mygpio == NULL) {
        printf("failed to open gpio");
        return -1;
    }

    read_count = fread(&read_int, sizeof(int), 1, mygpio);
    if (read_count > 0 ) {
        ;
    }

    fclose_ret = fclose(mygpio);
    if (fclose_ret == EOF) {
    }

    return 1;
}



int nssleep(long nsec) {
    struct timespec sleeptime;

    sleeptime.tv_sec = 0;
    sleeptime.tv_nsec = nsec;


    while (clock_nanosleep(CLOCK_MONOTONIC, 0, &sleeptime, NULL) == EINTR );

    return 0;
}

struct timespec diff(struct timespec start, struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
        temp.tv_sec = end.tv_sec-start.tv_sec-1;
        temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec-start.tv_sec;
        temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return temp;
}
```

```
#!/bin/sh
RUN_FLAG=1
# pin freigeben durch signal handler
trap RUN_FLAG=0 EXIT SIGINT SIGHUP SIGTERM SIGQUIT
#pin besschlagnahmen.
echo "18" >  /sys/class/gpio/export
# ausgang setzten
echo "out" > /sys/class/gpio/gpio18/direction
echo 1 >   /sys/class/gpio/gpio18/value


while [ $RUN_FLAG -eq 1 ]
do

  # led einschalten
  echo 0 >   /sys/class/gpio/gpio18/value
  sleep 1
  # led ausschalten.
  echo 1 >   /sys/class/gpio/gpio18/value
  sleep 1



done

echo "18" >/sys/class/gpio/unexport
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/timer.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()
#include <asm/segment.h>
#include <linux/buffer_head.h>
#include <linux/gpio.h>
#include <linux/io.h>
#include <asm/io.h>

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("mygpio device");
MODULE_SUPPORTED_DEVICE("none");

#define DEVNAME "mygpio"
#define MAJORNUM 150
#define NUMDEVICES 1

#define GPBASE(x) (0xF2200000 | x)

#define GPFSEL0 GPBASE(0x0000)
#define GPFSEL1 GPBASE(0x0004)
#define GPFSEL2 GPBASE(0x0008)
#define GPFSEL3 GPBASE(0x000C)
#define GPFSEL4 GPBASE(0x0010)
#define GPFSEL5 GPBASE(0x0014)
#define GPSET0 GPBASE(0x001C)
#define GPSET1 GPBASE(0x0020)
#define GPCLR0 GPBASE(0x0028)
#define GPCLR1 GPBASE(0x002C)
#define GPLEV0 GPBASE(0x0034)
#define GPLEV1 GPBASE(0x0038)


#define GPIO_PORT_18_SEL
#define GPIO_PORT_25_SEL
#define GPIO_PORT_18_SEL
#define GPIO_PORT_25_SET
#define GPIO_PORT_25_SET


static int read_bit(u32 n , int bit) {
    return (n & (1 << bit)) >> bit;
}

static u32 turn_on_bit(u32 n, int bitnum) {
        return n | (1 << bitnum);
}

static u32 turn_off_bit(u32 n, int bitnum) {
        return n & (~ (1 << bitnum));
}


static struct cdev *cdev = NULL;
static struct class *dev_class;




// function prototypes
static int __init mod_init(void);
static void __exit mod_exit(void);
static int driver_open(struct inode *inode, struct file *instance);
static ssize_t driver_write(struct file *instance, const char __user * userbuf, size
_t count, loff_t * off);
```

```
static int driver_close(struct inode *inode, struct file *instance);
static ssize_t driver_read(struct file *instance, char *user, size_t count, loff_t *
 offset);
static int read_bit(u32 n , int bit);


static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .read = driver_read,
    .write = driver_write,
    .release = driver_close
};


static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset)
{
    u32 *ptr_port_button = (u32*) GPLEV0;
    u32 old_value_button = readl(ptr_port_button);

    char data[5];

    int size = sprintf(data, "%d\n", read_bit(old_value_button, 25));

    int not_copied = copy_to_user(user, data , size);


    return  size - not_copied;
}

static ssize_t driver_write(struct file *instance, const char __user * userbuf, size
_t count, loff_t * off)
{
    int userval;

    int not_copied = kstrtoint_from_user(userbuf, count, 0, &userval);
    u32 *ptr_port_led = NULL;

    if (userval == 1) {
        ptr_port_led = (u32*)GPSET0;
    }
    if (userval == 0) {
        ptr_port_led = (u32*)GPCLR0;
    }
    if (ptr_port_led == NULL) {
        return count;
    }
    u32 old_value_led = (0x00000000) | (1 << 18);


    writel(old_value_led, ptr_port_led);


    return count - not_copied;
}


static ssize_t driver_close(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t driver_open(struct inode *inode, struct file *file)
{


    u32 *ptr_port_button = (u32*)GPFSEL2;
    u32 old_value_button = readl(ptr_port_button);
    rmb();
    // clear its fuer port 25 = FFFC7FFF
    u32 cleared_ports = old_value_button & 0xFFFC7FFF;
```

```
    // writing
    wmb();
    writel(cleared_ports, ptr_port_button);



    u32 *ptr_port_led = (u32*)GPFSEL1;
    u32 old_value_led = readl(ptr_port_led);
    rmb();
    // clear its fuer port 18 = F8FFFFFF
    u32 cleared_ports_led = old_value_led & 0xF8FFFFFF;
    u32 new_ports_led = cleared_ports_led | 0x01000000;
    wmb();
    writel(new_ports_led, ptr_port_led);



    return 0;
}


static int __init mod_init(void)
{

    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    printk(KERN_ALERT "mypgio: Hello, world!\n");

    if (register_chrdev_region(major_nummer, NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n", MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();

    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);



    return 0;

      free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
      free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

static void __exit mod_exit(void)
{
    printk(KERN_ALERT "mypgio: Goodbye, cruel world\n");
    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);
```

```
    if (cdev) {
        cdev_del(cdev);
    }

    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);

}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>



void sigint_handler(int);


int main()
{
    int trigger_count = 0;
    printf("activating sighandler\n");
    struct sigaction  sa;
    int fclose_ret = 0;
    int trigger_temp = 1;
    FILE *gpio17 = NULL;

    sigemptyset(&sa.sa_mask);
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    printf("starting trigger logger\n");
    perror("");

    perror("");



    while(1) {
        char read_text[10];
        size_t read_count;

        gpio17 = fopen("/sys/class/gpio/gpio17/value", "r");
        if (gpio17 == NULL) {
            printf("failed to open gpio");
            return -1;
        }
        read_count = fread(&read_text, sizeof(char), 2, gpio17);
        if(read_count > 0 ) {
            int trigger_state = atoi(read_text);
            if (trigger_temp == 0 && trigger_state == 1){
                trigger_count++;
                printf("trigger pushed %d times\n",trigger_count);
            }
            trigger_temp = atoi(read_text);
        }
        fclose_ret = fclose(gpio17);
        if (fclose_ret == EOF) {
            return -1;
        }

    }


    return 0;
}

void sigint_handler(int signum) {
    printf("Programm interrupted with signum: %d\n", signum);
    exit(0);
}
```