```
// A. Sheaff 3/19/2019 - four FSK driver
// File operations framework
//
// Modified 4/17/18 by Josh Andrews
// Included the write, open, and release functions and introduced locking
//
// Blocking/nonblocking is still a concern (how to check pins in use)
// Partly tested, seems to encode/toggle correctly
// Also need to test multiple users

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/err.h>
#include <linux/fs.h>
#include <linux/spinlock.h>
#include <linux/delay.h>
#include <linux/list.h>
#include <linux/io.h>
#include <linux/ioctl.h>
#include <asm/uaccess.h>
#include <linux/irq.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
//#include <mach/gpio.h>
#include <linux/gpio.h>
#include <linux/of_gpio.h>
#include <linux/platform_device.h>
//#include <mach/platform.h>
#include <linux/pinctrl/consumer.h>
#include <linux/gpio/consumer.h>
#include <linux/types.h>
#include <linux/mutex.h>

// Time delays for toggling leds
#define ENABLE_TIME 1000    // 1ms for usleep
#define CLOCK_TIME  16000   // 16 ms for usleep
#define WAIT_TIME   2000    // 2 seconds for msleep

// Function declarations, needed to get things working correctly
static ssize_t four_fsk_write(struct file *filp, const char __user * buf, size_t count, lof
f_t * offp);
static long four_fsk_ioctl(struct file * filp, unsigned int cmd, unsigned long arg);
static int four_fsk_open(struct inode *inode, struct file *filp);
static int four_fsk_release(struct inode *inode, struct file *filp);
static int encode(char *buf, int length);


// declare and initialize data to null so we can have a global ptr to the data
struct four_fsk_data_t *fsk_dat = NULL;

// Data to be "passed" around to various functions
struct four_fsk_data_t {
    struct gpio_desc *gpio_enable;              // Enable pin
    struct gpio_desc *gpio_clock;               // Clock pin
    struct gpio_desc *gpio_bit1;                // Bit 0 pin
    struct gpio_desc *gpio_bit0;                // Bit 1 pin
    struct gpio_desc *gpio_shutdown;    // Shutdown input
    int major;                                  // Device major number
    struct class *four_fsk_class;       // Class for auto /dev population
    struct device *four_fsk_dev;        // Device for auto /dev population
    struct mutex lock;              // mutex locking
};

// File operations for the four_fsk device
static const struct file_operations four_fsk_fops = {
        .owner = THIS_MODULE,       // Us
        .open = four_fsk_open,          // Open
```

```
        .release = four_fsk_release,// Close
        .write = four_fsk_write,        // Write
        .unlocked_ioctl=four_fsk_ioctl, // ioctl
};

// May be used to control the device
static long four_fsk_ioctl(struct file * filp, unsigned int cmd, unsigned long arg)
{
        return -EINVAL;
}

// You will need to choose the type of locking yourself.  It may be atmonic variables, spin
locks, mutex, or semaphore.

// Write system call
// If another process is using the pins and the device was opened O_NONBLOCK
//   then return with the appropriate error
// Otherwise
//   If another process is using the pins
//     then block/wait for the pin to be free.  Clean up and return an error if a signal is
 received.
//   Otherwise
//     Copy the user space data using copy_from_user() to a local kernel space buffer
//     Encode to the copied data using 4b5b (your 4b5 code) to another kernel buffer
//     Free the first buffer
//     Toggle pins as in homework 06.  Go to sleep while delaying. *** SEE TIMERS-HOWTO.TXT
 IN THE KERNEL DOCUMENTATION ***
//  CLEAN UP AND RETURN APPROPRAITE VALUE
static ssize_t four_fsk_write(struct file *filp, const char __user * buf, size_t count, lof
f_t * offp)
{
    int err = 0;
    char *kbuf;

    // If device is busy and was opened with O_NONBLOCK, return error
    if (mutex_is_locked(&(fsk_dat->lock)) && (filp->f_flags & O_NONBLOCK)) {
        printk(KERN_INFO "Non-blocking specified and device in use");
        return -EAGAIN;
    }

    // Set up locking, interruptible so process will wait to write if busy
    err = mutex_lock_interruptible(&(fsk_dat->lock));
    if (err) {
        printk(KERN_INFO "Could not set locking!\n");
        return -EACCES;
    }

    // Allocate space for kernel buffer, same size as user buffer
    kbuf = (char *)kmalloc(count,GFP_ATOMIC);
    if (kbuf == NULL) {
        printk(KERN_INFO "Could not allocate memory!\n");
        return -ENOMEM;
    }

    //Copy user buffer to kernel buffer
    err = copy_from_user(kbuf, buf, count);
    if (err != 0) {
        printk(KERN_INFO "Could not copy userspace data!\n");
        kfree(kbuf);
        kbuf = NULL;
        mutex_unlock(&(fsk_dat->lock));
        return -EFAULT;
    }

    // Start of encoding
    // encode() sets bit1, bit0, clock, and toggles leds for those values
    // All pins should be setup and ready to go at this point so just set value
```

```
    // Set enable high
    gpiod_set_value(fsk_dat->gpio_enable,1);
    printk(KERN_INFO "Enable set High\n");

    // sleep 1 ms before toggling other pins
    usleep_range(ENABLE_TIME, ENABLE_TIME + 25);

    // now encode the buffer and set bits/clock (also lights LEDs)
    // encode function returns 0 on success, or err which is -ENOMEM
    err = encode(kbuf, count);
    if (err < 0) {
        printk(KERN_INFO "Encoding Failed\n");
        kfree(kbuf);
        kbuf = NULL;
        return -ENOMEM;
    }

    // All done so set enable low
    gpiod_set_value(fsk_dat->gpio_enable,0);
    printk(KERN_INFO "Enable set low\n");

    // wait 2 sec for good measure (per sheaff suggestion)
    // use msleep as recommended by timers-howto.txt referenced above
    msleep(ENABLE_TIME);

    // turn off locking
    mutex_unlock(&(fsk_dat->lock));

    // clean up memory, return size
    printk(KERN_INFO "Now cleaning up\n");
    kfree(kbuf);
    kbuf = NULL;
    return count;
}

// Function to 4b5 encode user buffer and toggle leds based on buffer bit values
static int encode(char *buf, int length)
{
    // 4b5 encoding map
    const char map4b5[16] = {0b11110, 0b01001, 0b10100, 0b10101, 0b01010,
                             0b01011, 0b01110, 0b01111, 0b10010, 0b10011, 0b10110,
                             0b10111, 0b11010, 0b11011, 0b11100, 0b11101};
    int i, j;
    int size;
    int bit_cnt = 0;    // counter varible to limit bits printed
    int byte_cur = 0;   // index of data buf where 4b5 bits go
    int bit_pos = 7;    // the bit position in the current byte
    char *data;
    int high_nib, low_nib;  // the high and low nibble of each byte

    // The number of bytes needed to hold the encoded buffer
    size = ((length % 4) == 0) ? (5*length / 4) : ((5*length / 4) + 1);

    // allocate memory for encoded buffer
    data = (char *)kmalloc(size,GFP_ATOMIC);
    if (data == NULL) {
        printk(KERN_INFO "Encode memory allocation failed!\n");
        return -ENOMEM;
    }
    // Initialize allocated space to 0 so we encode correctly
    data = memset(data, 0, size);

    // Start 4b5 encoding (taken from HW5)
    for (i = 0; i < length; i++) {

        //get the high 4 and low 4 bits of each byte
        high_nib = (int)buf[i] / 16;
        low_nib = (int)buf[i] % 16;
```

```
        // map high nibble to 4b5 value and place into data array
        // if there is enough room on the current byte, put bits there
        if (bit_pos > 4) {
            data[byte_cur] |= map4b5[high_nib] << (bit_pos - 4);
            bit_pos -= 5;
        }
        // if not enough room, split between current and next byte
        else {
            data[byte_cur] |= map4b5[high_nib] >> (4 - bit_pos);
            data[++byte_cur] |= map4b5[high_nib] << (bit_pos + 4);
            bit_pos += 3;
        }

        // map low nibble to 4b5 value and place into data array
        // same priciple as for the high nibble
        if (bit_pos > 4) {
            data[byte_cur] |= map4b5[low_nib] << (bit_pos - 4);
            bit_pos -= 5;
        }
        else {
            data[byte_cur] |= map4b5[low_nib] >> (4 - bit_pos);
            data[++byte_cur] |= map4b5[low_nib] << (bit_pos + 4);
            bit_pos += 3;
        }
    }

    // Now light up the leds
    printk(KERN_INFO "Starting LEDs\n");

    // Loop through encode buffer and light leds(bit1, bit0) based on bit pairs
    for (i = 0; i < size; i++) {
        for (j = 7; j > 0; j-=2) {

            // stop if end of encoded data is reached
            if (bit_cnt == length*10) {
                break;
            }

            // set clock high
            gpiod_set_value(fsk_dat->gpio_clock,1);

            //set bit values
            gpiod_set_value(fsk_dat->gpio_bit1,(data[i] >> j) & 0x01);
            gpiod_set_value(fsk_dat->gpio_bit0, (data[i] >> (j-1)) & 0x01);
            bit_cnt += 2;

            //wait 16ms
            usleep_range(CLOCK_TIME, CLOCK_TIME + 50);

            // Set clock low
            gpiod_set_value(fsk_dat->gpio_clock, 0);

            //wait 16ms then repeat
            usleep_range(CLOCK_TIME, CLOCK_TIME + 50);
        }
    }

    // Set data LEDs to 0 since were done
    gpiod_set_value(fsk_dat->gpio_bit1, 0);
    gpiod_set_value(fsk_dat->gpio_bit0, 0);


    // free up memory and return
    kfree(data);
    data = NULL;
    return 0;
}
```

```
// Open system call
// Open only if the file access flags (NOT permissions) are appropiate as discussed in clas
s
// Return an appropraite error otherwise
static int four_fsk_open(struct inode *inode, struct file *filp)
{
    // Only allow file to be opened write only
    if (!(filp->f_flags & O_WRONLY)) {
        return -EINVAL;
    }
    printk(KERN_INFO "Four FSK opened successfully\n");
    return 0;
}

// Close system call
// What is there to do?
static int four_fsk_release(struct inode *inode, struct file *filp)
{
    // print so we know we got here (debugging)
    printk(KERN_INFO "Four FSK released successfully\n");
    return 0;
}
```

```
static struct gpio_desc *four_fsk_dt_obtain_pin(struct device *dev, struct device_node *par
ent, char *name, int init_val)
{
        struct device_node *dn_child=NULL;      // DT child
        struct gpio_desc *gpiod_pin=NULL;       // GPIO Descriptor for setting value
        int ret=-1;     // Return value
        int pin=-1;     // Pin number
        char *label=NULL;       // DT Pin label

        // Find the child - release with of_node_put()
        dn_child=of_get_child_by_name(parent,name);
        if (dn_child==NULL) {
                printk(KERN_INFO "No child %s\n",name);
                gpiod_pin=NULL;
                goto fail;
        }

        // Get the child pin number - does not appear to need to be released
        pin=of_get_named_gpio(dn_child,"gpios",0);
        if (pin<0) {
                printk(KERN_INFO "no %s GPIOs\n",name);
                gpiod_pin=NULL;
                goto fail;
        }
        // Verify pin is OK
        if (!gpio_is_valid(pin)) {
                gpiod_pin=NULL;
                goto fail;
        }
        printk(KERN_INFO "Found %s pin %d\n",name,pin);

        // Get the of string tied to pin - Does not appear to need to be released
        ret=of_property_read_string(dn_child,"label",(const char **)&label);
        if (ret<0) {
                printk(KERN_INFO "Cannot find label\n");
                gpiod_pin=NULL;
                goto fail;
        }
```

```
        // Request the pin - release with devm_gpio_free() by pin number
        if (init_val>=0) {
                ret=devm_gpio_request_one(dev,pin,GPIOF_OUT_INIT_LOW,label);
                if (ret<0) {
                        dev_err(dev,"Cannot get %s gpio pin\n",name);
                        gpiod_pin=NULL;
                        goto fail;
                }
        } else {
                ret=devm_gpio_request_one(dev,pin,GPIOF_IN,label);
                if (ret<0) {
                        dev_err(dev,"Cannot get %s gpio pin\n",name);
                        gpiod_pin=NULL;
                        goto fail;
                }
        }

        // Get the gpiod pin struct
        gpiod_pin=gpio_to_desc(pin);
        if (gpiod_pin==NULL) {
                printk(KERN_INFO "Failed to acquire enable gpio\n");
                gpiod_pin=NULL;
                goto fail;
        }

        // Make sure the pin is set correctly
        if (init_val>=0) gpiod_set_value(gpiod_pin,init_val);


        // Release the device node
    of_node_put(dn_child);

        return gpiod_pin;

fail:
        if (pin>=0) devm_gpio_free(dev,pin);
    if (dn_child) of_node_put(dn_child);

        return gpiod_pin;
}


// Sets device node permission on the /dev device special file
static char *four_fsk_devnode(struct device *dev, umode_t *mode)
{
        if (mode) *mode = 0666;
        return NULL;
}

// My data is going to go in either platform_data or driver_data
//  within &pdev->dev. (dev_set/get_drvdata)
// Called when the device is "found" - for us
// This is called on module load based on ".of_match_table" member
// Added the mutex initialization here (Josh 4/15/18)
static int four_fsk_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;        // Device associcated with platform

        struct four_fsk_data_t *four_fsk_dat;           // Data to be passed around the cal
ls
        struct device_node *dn=NULL;

        int ret=-1;     // Return value


        // Allocate device driver data and save
        four_fsk_dat=kmalloc(sizeof(struct four_fsk_data_t),GFP_ATOMIC);
        if (four_fsk_dat==NULL) {
```

```
                printk(KERN_INFO "Memory allocation failed\n");
                return -ENOMEM;
        }

        memset(four_fsk_dat,0,sizeof(struct four_fsk_data_t));

        dev_set_drvdata(dev,four_fsk_dat);

// Find my device node
        dn=of_find_node_by_name(NULL,"four_fsk");
        if (dn==NULL) {
                printk(KERN_INFO "Cannot find device\n");
                ret=-ENODEV;
                goto fail;
        }
        four_fsk_dat->gpio_enable=four_fsk_dt_obtain_pin(dev,dn,"Enable",0);
        if (four_fsk_dat->gpio_enable==NULL) goto fail;
        four_fsk_dat->gpio_clock=four_fsk_dt_obtain_pin(dev,dn,"Clock",0);
        if (four_fsk_dat->gpio_clock==NULL) goto fail;
        four_fsk_dat->gpio_bit0=four_fsk_dt_obtain_pin(dev,dn,"Bit0",0);
        if (four_fsk_dat->gpio_bit0==NULL) goto fail;
        four_fsk_dat->gpio_bit1=four_fsk_dt_obtain_pin(dev,dn,"Bit1",0);
        if (four_fsk_dat->gpio_bit1==NULL) goto fail;
        four_fsk_dat->gpio_shutdown=four_fsk_dt_obtain_pin(dev,dn,"Shutdown",-1);
        if (four_fsk_dat->gpio_shutdown==NULL) goto fail;



        // Create the device - automagically assign a major number
        four_fsk_dat->major=register_chrdev(0,"four_fsk",&four_fsk_fops);
        if (four_fsk_dat->major<0) {
                printk(KERN_INFO "Failed to register character device\n");
                ret=four_fsk_dat->major;
                goto fail;
        }

        // Create a class instance
        four_fsk_dat->four_fsk_class=class_create(THIS_MODULE, "four_fsk_class");
        if (IS_ERR(four_fsk_dat->four_fsk_class)) {
                printk(KERN_INFO "Failed to create class\n");
                ret=PTR_ERR(four_fsk_dat->four_fsk_class);
                goto fail;
        }

        // Setup the device so the device special file is created with 0666 perms
        four_fsk_dat->four_fsk_class->devnode=four_fsk_devnode;
        four_fsk_dat->four_fsk_dev=device_create(four_fsk_dat->four_fsk_class,NULL,MKDEV(fo
ur_fsk_dat->major,0),(void *)four_fsk_dat,"four_fsk");
        if (IS_ERR(four_fsk_dat->four_fsk_dev)) {
                printk(KERN_INFO "Failed to create device file\n");
                ret=PTR_ERR(four_fsk_dat->four_fsk_dev);
                goto fail;
        }
    // Not sure if I actually need this now, might try to
    fsk_dat=four_fsk_dat;
    // Initialize mutex
    mutex_init(&(four_fsk_dat->lock));
    printk(KERN_INFO "Mutex initialized!\n");

        printk(KERN_INFO "Registered\n");
        dev_info(dev, "Initialized");
        return 0;

fail:
        // Device cleanup
        if (four_fsk_dat->four_fsk_dev) device_destroy(four_fsk_dat->four_fsk_class,MKDEV(f
our_fsk_dat->major,0));
        // Class cleanup
```

```
        if (four_fsk_dat->four_fsk_class) class_destroy(four_fsk_dat->four_fsk_class);
        // char dev clean up
        if (four_fsk_dat->major) unregister_chrdev(four_fsk_dat->major,"four_fsk");

        if (four_fsk_dat->gpio_shutdown) devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio
_shutdown));
        if (four_fsk_dat->gpio_bit1) devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_bit
1));
        if (four_fsk_dat->gpio_bit0) devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_bit
0));
        if (four_fsk_dat->gpio_clock) devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_cl
ock));
        if (four_fsk_dat->gpio_enable) devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_e
nable));


        dev_set_drvdata(dev,NULL);
        kfree(four_fsk_dat);
        printk(KERN_INFO "Four FSK Failed\n");
        return ret;
}

// Called when the device is removed or the module is removed
static int four_fsk_remove(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct four_fsk_data_t *four_fsk_dat;   // Data to be passed around the calls

        // Obtain the device driver data
        four_fsk_dat=dev_get_drvdata(dev);

        // Device cleanup
        device_destroy(four_fsk_dat->four_fsk_class,MKDEV(four_fsk_dat->major,0));
        // Class cleanup
        class_destroy(four_fsk_dat->four_fsk_class);
        // Remove char dev
        unregister_chrdev(four_fsk_dat->major,"four_fsk");

        // Free the gpio pins with devm_gpio_free() & gpiod_put()
        devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_shutdown));
        devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_bit1));
        devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_bit0));
        devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_clock));
        devm_gpio_free(dev,desc_to_gpio(four_fsk_dat->gpio_enable));
#if 0
        // not clear if these are allocated and need to be freed
        gpiod_put(four_fsk_dat->gpio_shutdown);
        gpiod_put(four_fsk_dat->gpio_bit1);
        gpiod_put(four_fsk_dat->gpio_bit0);
        gpiod_put(four_fsk_dat->gpio_clock);
        gpiod_put(four_fsk_dat->gpio_enable);
#endif

        // Free the device driver data
        dev_set_drvdata(dev,NULL);
        kfree(four_fsk_dat);

        printk(KERN_INFO "Removed\n");
        dev_info(dev, "GPIO mem driver removed - OK");

        return 0;
}

static const struct of_device_id four_fsk_of_match[] = {
    {.compatible = "brcm,bcm2835-four_fsk",},
    { /* sentinel */ },
};
```

```
MODULE_DEVICE_TABLE(of, four_fsk_of_match);

static struct platform_driver four_fsk_driver = {
    .probe = four_fsk_probe,
    .remove = four_fsk_remove,
    .driver = {
           .name = "bcm2835-four_fsk",
           .owner = THIS_MODULE,
           .of_match_table = four_fsk_of_match,
           },
};

module_platform_driver(four_fsk_driver);

MODULE_DESCRIPTION("Four FSK pin modulator");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Four FSK");
```

```
# Makefile for Four-Fsk kernel driver

obj-m+= four-fsk.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
/* user.c
 *
 * Test program for the 4fsk kernel driver
 *
 * Tests correct flags on open call
 * Tests for correct encoding
 * Tests blocking using multiple processes
 *
 * Josh Andrews
 * ECE-331
 * 4/17/18
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int fd, length, err;
    char *msg;
    pid_t child;

    // Only accept one arguement
    if (argc != 2) {
        printf("Usage: %s <message>\n", argv[0]);
        return -1;
    }

    // copy message and get length
    msg = argv[1];
    length = strlen(msg);

    // Test RDONLY flag, should fail so say so and move on to more testing
    fd = open("dev/four_fsk", O_RDONLY);
    if (fd < 0) {
        printf("O_RDONLY did not work\n");
    }

    // Test RDWR flag, should fail so print it and continue testing
    fd = open("dev/four_fsk", O_RDWR);
    if (fd < 0) {
        printf("O_RDWR did not work\n");
    }

    // Use the NONBLOCK open to test correct operation
    // Use O_WRONLY open to test correct blocking

    fd = open("/dev/four_fsk", O_WRONLY | O_NONBLOCK);
    //fd = open("/dev/four_fsk", O_WRONLY);
    if (fd < 0) {
        printf("Could not open device!\n");
        return -2;
    }

    // Write the message to the device
    err = write(fd, msg, length);
    if (err < 0) {
        printf("Could not write to device!\n");
        close(fd);
        return -3;
    }

    // Create parent-child pair of processes to test blocking
    child = fork();
```

```
    // Write with parent process
    if (child > 0) {
        err = write(fd, msg, length);
        if (err < 0) {
            printf("Parent process could not write\n");
            close(fd);
            return -4;
        }
    }
    // Write with child process
    else if (child == 0) {
        err = write(fd, msg, length);
        if (err < 0) {
            printf("Child process could not write\n");
            close(fd);
            return -4;
        }
    }
    // fork failed
    else {
        printf("Could not generate processes\n");
        close(fd);
        return -5;
    }

    //close device and return
    close(fd);
    return 0;
}
```

```
# Makefile for user.c

CFLAGS=-g -Wall
OBJS=user.o
CC=gcc

.PHONY: all clean

all: user

${TARGET}: ${OBJS}
	${CC} -o ${TARGET} ${OBJS} ${LIBS}

clean:
	rm -f user ${OBJS}
```