Jordan Millett ECE 331 afsk.c

```
// Jordan Millett ECE 331 Project 1
// Adapted from;
// A. Sheaff 3/7/2016
// AFSK kernel driver framework - RPi
// A file operations structure must be defined for this
// module to work correctly
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/err.h>
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/fs.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/stat.h>

#include <linux/mutex.h>
#include <linux/delay.h>
#include <linux/string.h>
#include <asm/uaccess.h>
#include "afsk.h"
//#include <linux/init.h>
//#include <linux/fcntl.h>
#include <linux/sched.h>

#define AFSK_NOSTUFF 0
#define AFSK_STUFF 1
#define AX25_DELIM 0x7E

// Forward declaration
struct afsk_data_t;

// Data to be "passed" around to various functions
struct afsk_data_t {
    int gpio_enable;    // Enable pin
    int gpio_m_sb;      // Mark-Space bar pin
    int gpio_ptt;       // Push to talk pin
    int gpio_shdn;      // Shutdown input
    int major;          // Device major number
    struct class *afsk_class;   // Class for auto /dev population
    struct device *afsk_dev;    // Device for auto /dev population
    struct gpio_desc *enable;   // gpiod Enable pin
    struct gpio_desc *m_sb;     // gpiod Mark/Space bar pin
    struct gpio_desc *ptt;      // gpiod Push to talk pin
    struct gpio_desc *shdn;     // Shutdown pin
    u32 delim_cnt;              // Delimiter count
    u8 *delim_buf;              // Delimtter buffer - changes size in ioctl
    struct mutex *lock;
};


// Declare functions before file_operations
static int afsk_open(struct inode *inode, struct file *filp);
static int afsk_release(struct inode *inode, struct file *filp);
static int afsk_write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
static long afsk_ioctl(struct file *filp, uint cmd, unsigned long arg);
```

Jordan Millett ECE 331 afsk.c

```
// AFSK data structure access between functions
static struct afsk_data_t *afsk_data_fops;

static const struct file_operations afsk_fops = {
    .owner          = THIS_MODULE,
    .write          = afsk_write,
    .open           = afsk_open,
    .release        = afsk_release,
    .unlocked_ioctl     = afsk_ioctl
};
// Sets device node permission on the /dev device special file
static char *afsk_devnode(struct device *dev, umode_t *mode)
{
    if (mode) *mode = S_IRUGO|S_IWUGO;
    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
static int afsk_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;    // Device associcated with platform

    struct afsk_data_t *afsk_dat;       // Data to be passed around the calls

    struct device_node *dn;         // Start of my device tree
    struct device_node *dn_enable;  // Enable pin properties
    struct device_node *dn_m_sb;    // Mark-Space bar pin properties
    struct device_node *dn_ptt;     // PTT properties
    struct device_node *dn_shdn;    // Shutdown properties

    int ret;    // Return value

    const char *pin_name;   // Pin name

    // Allocate device driver data and save
    afsk_dat=kmalloc(sizeof(struct afsk_data_t),GFP_ATOMIC);
    if (afsk_dat==NULL) {
        printk(KERN_INFO "Memory allocation failed\n");
        return -ENOMEM;
    }

    memset(afsk_dat,0,sizeof(struct afsk_data_t));

    dev_set_drvdata(dev,afsk_dat);
    // Find my device node
    dn=of_find_node_by_name(NULL,"afsk");
    if (dn==NULL) {
        printk(KERN_INFO "Cannot find device\n");
        ret=-ENODEV;
        goto fail;
    }
    // Find the enable node
    dn_enable=of_get_child_by_name(dn,"Enable");
    if (dn_enable==NULL) {
        printk(KERN_INFO "No enable child\n");
        ret=-ENODEV;
        goto fail;
    }
    // Find the Mark/Space bar node
```

Jordan Millett ECE 331 afsk.c

```
    dn_m_sb=of_get_child_by_name(dn,"Mark_SpaceBar");
    if (dn_m_sb==NULL) {
        printk(KERN_INFO "No M_Sb child\n");
        ret=-ENODEV;
        goto fail;
    }
    // Find the PTT node
    dn_ptt=of_get_child_by_name(dn,"PTT");
    if (dn_ptt==NULL) {
        printk(KERN_INFO "No PTT child\n");
        ret=-ENODEV;
        goto fail;
    }
    // Find the Shutdown node
    dn_shdn=of_get_child_by_name(dn,"Shutdown");
    if (dn_shdn==NULL) {
        printk(KERN_INFO "No Shutdown child\n");
        ret=-ENODEV;
        goto fail;
    }
    // Get the enable pin number
    afsk_dat->gpio_enable=of_get_named_gpio(dn_enable,"gpios",0);
    if (afsk_dat->gpio_enable<0) {
        printk(KERN_INFO "no enable GPIOs\n");
        ret=-ENODEV;
        goto fail;
    }
    printk(KERN_INFO "Found enable pin %d\n",afsk_dat->gpio_enable);
    if (!gpio_is_valid(afsk_dat->gpio_enable)) {
        ret=-EINVAL;
        goto fail;
    }
    // Get the Mark-Space bar pin number
    afsk_dat->gpio_m_sb=of_get_named_gpio(dn_m_sb,"gpios",0);
    if (afsk_dat->gpio_m_sb<0) {
        printk(KERN_INFO "no mark/space GPIOs\n");
        ret=-ENODEV;
        goto fail;
    }
    printk(KERN_INFO "Found Mark-Space bar pin %d\n",afsk_dat->gpio_m_sb);
    if (!gpio_is_valid(afsk_dat->gpio_m_sb)) {
        ret=-EINVAL;
        goto fail;
    }
    // Get the PTT pin number
    afsk_dat->gpio_ptt=of_get_named_gpio(dn_ptt,"gpios",0);
    if (afsk_dat->gpio_ptt<0) {
        printk(KERN_INFO "no ptt GPIOs\n");
        ret=-ENODEV;
        goto fail;
    }
    printk(KERN_INFO "Found PTT pin %d\n",afsk_dat->gpio_ptt);
    if (!gpio_is_valid(afsk_dat->gpio_ptt)) {
        ret=-EINVAL;
        goto fail;
    }
    // Get the Shutdown pin number
    afsk_dat->gpio_shdn=of_get_named_gpio(dn_shdn,"gpios",0);
    if (afsk_dat->gpio_shdn<0) {
        printk(KERN_INFO "no Shutdown GPIOs\n");
        ret=-ENODEV;
        goto fail;
    }
```

Jordan Millett ECE 331 afsk.c

```
    printk(KERN_INFO "Found Shutdown pin %d\n",afsk_dat->gpio_shdn);
    if (!gpio_is_valid(afsk_dat->gpio_shdn)) {
        ret=-EINVAL;
        goto fail;
    }


    // Also need to pull labels from DT to pass to devm_gpio_request_one
    // Request and allocate the Enable pin
    ret=of_property_read_string(dn_enable,"name",&pin_name);
    if (ret<0) {
        printk(KERN_INFO "no Enable name\n");
        ret=-EINVAL;
        goto fail;
    }
    ret=devm_gpio_request_one(dev,afsk_dat->gpio_enable,GPIOF_OUT_INIT_LOW,pin_name);
    if (ret<0) {
        dev_err(dev,"Cannot get enable gpio pin\n");
        ret=-ENODEV;
        goto fail;
    }

    ret=of_property_read_string(dn_m_sb,"name",&pin_name);
    if (ret<0) {
        printk(KERN_INFO "no Enable name\n");
        ret=-EINVAL;
        goto fail;
    }
    // Request and allocate the Mark-Space bar pin
    ret=devm_gpio_request_one(dev,afsk_dat->gpio_m_sb,GPIOF_OUT_INIT_LOW,pin_name);
    if (ret<0) {
        dev_err(dev,"Cannot get mark/space gpio pin\n");
        ret=-ENODEV;
        goto fail;
    }

    ret=of_property_read_string(dn_ptt,"name",&pin_name);
    if (ret<0) {
        printk(KERN_INFO "no Enable name\n");
        ret=-EINVAL;
        goto fail;
    }
    // Request and allocate the PTT pin
    ret=devm_gpio_request_one(dev,afsk_dat->gpio_ptt,GPIOF_OUT_INIT_LOW,pin_name);
    if (ret<0) {
        dev_err(dev,"Cannot get ptt gpio pin\n");
        ret=-ENODEV;
        goto fail;
    }

    ret=of_property_read_string(dn_shdn,"name",&pin_name);
    if (ret<0) {
        printk(KERN_INFO "no Enable name\n");
        ret=-EINVAL;
        goto fail;
    }
    // Request and allocate the Shutdown pin - input
    ret=devm_gpio_request_one(dev,afsk_dat->gpio_shdn,GPIOF_IN,pin_name);
    if (ret<0) {
        dev_err(dev,"Cannot get shutdown gpio pin\n");
        ret=-ENODEV;
        goto fail;
    }
```

Jordan Millett ECE 331 afsk.c

```
    // "release" devicetree nodes
    if (dn_shdn) of_node_put(dn_shdn);
    if (dn_ptt) of_node_put(dn_ptt);
    if (dn_m_sb) of_node_put(dn_m_sb);
    if (dn_enable) of_node_put(dn_enable);
    if (dn) of_node_put(dn);

#if 0
    // Create the device - automagically assign a major number
    afsk_dat->major=register_chrdev(0,"afsk",&afsk_fops);
    if (afsk_dat->major<0) {
        printk(KERN_INFO "Failed to register character device\n");
        ret=afsk_dat->major;
        goto fail;
    }
#endif
    afsk_dat->lock=kmalloc(sizeof(struct mutex),GFP_KERNEL);
    mutex_init(afsk_dat->lock);
    afsk_dat->major = register_chrdev(0,"afsk",&afsk_fops);
    // Create a class instance
    afsk_dat->afsk_class=class_create(THIS_MODULE, "afsk_class");
    if (IS_ERR(afsk_dat->afsk_class)) {
        printk(KERN_INFO "Failed to create class\n");
        ret=PTR_ERR(afsk_dat->afsk_class);
        goto fail;
    }

    // Setup the device so the device special file is created with 0666 perms
    afsk_dat->afsk_class->devnode=afsk_devnode;
    afsk_dat->afsk_dev=device_create(afsk_dat->afsk_class,NULL,MKDEV(afsk_dat->major,0),(void
*)afsk_dat,"afsk");
    if (IS_ERR(afsk_dat->afsk_dev)) {
        printk(KERN_INFO "Failed to create device file\n");
        ret=PTR_ERR(afsk_dat->afsk_dev);
        goto fail;
    }

    // Get the gpiod pin struct for the enable pin
    afsk_dat->enable=gpio_to_desc(afsk_dat->gpio_enable);
    if (afsk_dat->enable==NULL) {
        printk(KERN_INFO "Failed to acquire enable gpio\n");
        ret=-ENODEV;
        goto fail;
    }

    // Get the gpiod pin struct for the s_mb pin
    afsk_dat->m_sb=gpio_to_desc(afsk_dat->gpio_m_sb);
    if (afsk_dat->m_sb==NULL) {
        printk(KERN_INFO "Failed to acquire mark/space gpio\n");
        ret=-ENODEV;
        goto fail;
    }

    // Get the gpiod pin struct for the ptt pin
    afsk_dat->ptt=gpio_to_desc(afsk_dat->gpio_ptt);
    if (afsk_dat->ptt==NULL) {
        printk(KERN_INFO "Failed to acquire ptt gpio\n");
        ret=-ENODEV;
        goto fail;
    }

    // Get the gpiod pin struct for the shutdown pin
```

Jordan Millett ECE 331 afsk.c

```
    afsk_dat->shdn=gpio_to_desc(afsk_dat->gpio_shdn);
    if (afsk_dat->shdn==NULL) {
        printk(KERN_INFO "Failed to acquire shutdown gpio\n");
        ret=-ENODEV;
        goto fail;
    }

    // Set up our global pointer to our data
    afsk_data_fops=afsk_dat;

    // Initialize the output pins - should already be done above....
    gpiod_set_value(afsk_dat->enable,0);
    gpiod_set_value(afsk_dat->m_sb,0);
    gpiod_set_value(afsk_dat->ptt,0);

    // Set the delim count for the AX25 packet
    afsk_dat->delim_cnt=16;

    // Allocate memory for the delim
    afsk_dat->delim_buf=kmalloc(afsk_dat->delim_cnt,GFP_KERNEL);
    if (afsk_dat->delim_buf==NULL) {
        printk(KERN_INFO "Failed to allocate delim memory\n");
        ret=-ENOMEM;
        goto fail;
    }

    // Set the delim buffer values
    memset(afsk_dat->delim_buf,AX25_DELIM,afsk_dat->delim_cnt);



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

fail:
    if (afsk_dat->delim_buf) kfree(afsk_dat->delim_buf);
    if (afsk_dat->shdn) gpiod_put(afsk_dat->shdn);
    if (afsk_dat->ptt) gpiod_put(afsk_dat->ptt);
    if (afsk_dat->m_sb) gpiod_put(afsk_dat->m_sb);
    if (afsk_dat->enable) gpiod_put(afsk_dat->enable);
    if (!(IS_ERR(afsk_dat->afsk_dev))) device_destroy(afsk_dat->afsk_class,MKDEV(afsk_dat->maj
or,0));
    if (!(IS_ERR(afsk_dat->afsk_class))) class_destroy(afsk_dat->afsk_class);
    if (afsk_dat->major>0) unregister_chrdev(afsk_dat->major,"afsk");
    if (afsk_dat->gpio_shdn>0) devm_gpio_free(dev,afsk_dat->gpio_shdn);
    if (afsk_dat->gpio_ptt>0) devm_gpio_free(dev,afsk_dat->gpio_ptt);
    if (afsk_dat->gpio_m_sb>0) devm_gpio_free(dev,afsk_dat->gpio_m_sb);
    if (afsk_dat->gpio_enable>0) devm_gpio_free(dev,afsk_dat->gpio_enable);
    if (dn_shdn) of_node_put(dn_shdn);
    if (dn_ptt) of_node_put(dn_ptt);
    if (dn_m_sb) of_node_put(dn_m_sb);
    if (dn_enable) of_node_put(dn_enable);
    if (dn) of_node_put(dn);
    if (afsk_dat) kfree(afsk_dat);
    dev_set_drvdata(dev,NULL);
    printk(KERN_INFO "AFSK Failed\n");
    return ret;

}

// Called when the device is removed or the module is removed
static int afsk_remove(struct platform_device *pdev)
```

Jordan Millett ECE 331 afsk.c

```
{
    struct device *dev = &pdev->dev;
    struct afsk_data_t *afsk_dat;   // Data to be passed around the calls

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

    kfree(afsk_dat->delim_buf);

    gpiod_put(afsk_dat->shdn);
    gpiod_put(afsk_dat->ptt);
    gpiod_put(afsk_dat->m_sb);
    gpiod_put(afsk_dat->enable);

    // Release the device
    device_destroy(afsk_dat->afsk_class,MKDEV(afsk_dat->major,0));

    // Release the class
    class_destroy(afsk_dat->afsk_class);

    // Release the character device
    unregister_chrdev(afsk_dat->major,"afsk");

    // Free the gpio pins
    devm_gpio_free(dev,afsk_dat->gpio_shdn);
    devm_gpio_free(dev,afsk_dat->gpio_ptt);
    devm_gpio_free(dev,afsk_dat->gpio_m_sb);
    devm_gpio_free(dev,afsk_dat->gpio_enable);

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

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

    return 0;
}

// Jordan’s code Start
int encoder(char *data, int mode)
{
    int i;
    int j;
    int sm = 0;
    int counter = 0;
    int stuffed = 0;
    int MASK = 1;
    int *bits, *stuffbits;
    int size, numbits;

    size = strlen(data);
    numbits = size * 8;
    // Memory allocation
    bits = kmalloc(8 * size * sizeof(int), GFP_KERNEL);
    stuffbits = kmalloc((8 * size + (size * 8) / 5) * sizeof(int), GFP_KERNEL);
    // Error checking
    if (bits == NULL || stuffbits == NULL) {
        return -ENOMEM;
    }
    // Store binary values
    for (i = 0; i < size; i++) {
        for (j = 0; j < 8; j++) {
```

```
            bits[i * 8 + j] = (data[i] & (MASK << j));
        }
    }
    // Bit stuffing
    if (mode) {
        for (i = 0; i < numbits; i++) {
            stuffbits[i + stuffed] = bits[i];
            if(stuffbits[i + stuffed]) {
                counter++;
                if(counter == 5) {
                    stuffed++;
                    counter = 0;
                    numbits++;
                    stuffbits[i + stuffed] = 0;
                }
            } else {
                counter = 0;
            }
        }
    } else {
        for (i = 0; i < numbits; i++) {
            stuffbits[i] = bits[i];
        }
    }
    // NRZI
    gpiod_set_value(afsk_data_fops->m_sb,sm);
    for (i = 0; i < numbits; i++) {
        if(stuffbits[i]) {
            gpiod_set_value(afsk_data_fops->m_sb,sm);
        } else {
            sm = !sm;
            gpiod_set_value(afsk_data_fops->m_sb,sm);
        }
    }
    kfree(bits);
    kfree(stuffbits);
    return 0;
}
static int afsk_open(struct inode *inode, struct file *filp)
{
    if(filp->f_flags & O_WRONLY) return 0;
    return -EINVAL; // Error
    //ENOTSUP doesn't work for some reason
}
static int afsk_write(struct file *filp, const char __user *buff, size_t count, loff_t *offp)
{
    int ret;
    char *data;
    data = kmalloc(sizeof(char) * count, GFP_KERNEL);

    // Lock
    ret = mutex_lock_interruptible(afsk_data_fops->lock);
    if (!ret) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOLCK;
    }
    if (afsk_data_fops->delim_buf == NULL) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOMEM;
    }
```

Jordan Millett ECE 331 afsk.c

```
    // Enable PTT
    gpiod_set_value(afsk_data_fops->ptt,1);
    // Wait
    mdelay(5);
    // Enable enable
    gpiod_set_value(afsk_data_fops->enable,1);
    // Get data from userspace
    ret = copy_from_user(data, buff, count);
    if (!ret) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOMEM;
    }

    /* Data             */
    // Delim -> NRZI -> MS
    ret = encoder(afsk_data_fops->delim_buf, AFSK_NOSTUFF);
    if (!ret) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOMEM;
    }
    // Write buffer -> bitstuffing -> NRZI -> MS
    ret = encoder(data, AFSK_STUFF);
    if (!ret) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOMEM;
    }
    // Delim ->NRZI -> MS
    ret = encoder(afsk_data_fops->delim_buf, AFSK_NOSTUFF);
    if (!ret) {
        // Unlock
        mutex_unlock(afsk_data_fops->lock);
        return -ENOMEM;
    }
    /* End Data         */

    // Disable enable
    gpiod_set_value(afsk_data_fops->enable,0);
    // Wait
    mdelay(5);
    // Disable PTT
    gpiod_set_value(afsk_data_fops->ptt,0);

    // Unlock
    mutex_unlock(afsk_data_fops->lock);
    return 0;
}
static int afsk_release(struct inode *inode, struct file *filp)
{
    return 0;
}

static long afsk_ioctl(struct file *filp, uint cmd, unsigned long arg)
{
    int ret;
    uint32_t memsize;
    ret = 1;
    switch (cmd) {
        case 6669:
            ret = mutex_lock_interruptible(afsk_data_fops->lock);
            if (ret != 0) {
```

Jordan Millett ECE 331 afsk.c

```
                // Unlock
                mutex_unlock(afsk_data_fops->lock);
                printk(KERN_INFO "Locking");
                return -ENOLCK;
            }
            // Gets size of allocated delim buffer
            memsize = afsk_data_fops->delim_cnt;
            // Sends size to user space
            ret = put_user(memsize, (uint8_t __user *) arg);
            if (ret != 0) {
                // Unlock
                mutex_unlock(afsk_data_fops->lock);
                printk(KERN_INFO "put_user");
                return -EFAULT;
            }
            // Unlock
            //mutex_unlock(afsk_data_fops->lock);
            printk(KERN_INFO "query %d",arg);
            return 0;
        case 6670:
            ret = mutex_lock_interruptible(afsk_data_fops->lock);
            if (ret != 0) {
                // Unlock
                mutex_unlock(afsk_data_fops->lock);
                printk(KERN_INFO "Locking");
                return -ENOLCK;
            }
            // Get value of arg from userspace
            ret = get_user(memsize, (uint32_t __user *) arg);
            if (ret != 0) {
                // Unlock
                mutex_unlock(afsk_data_fops->lock);
                printk(KERN_INFO "get_user");
                return ret;
            }
            // Free old buffer
            kfree(afsk_data_fops->delim_buf);
            // Allocates new buffer and saves the size
            afsk_data_fops->delim_cnt = memsize;
            afsk_data_fops->delim_buf = kmalloc(afsk_data_fops->delim_cnt, GFP_KERNEL);
            // Error checking
            if (afsk_data_fops->delim_buf == NULL) {
                printk(KERN_INFO "Failed to allocate delim memory\n");
                // Unlock
                mutex_unlock(afsk_data_fops->lock);
                return -ENOMEM;
            }
            // Store the delim in the buffer
            memset(afsk_data_fops->delim_buf, AX25_DELIM, afsk_data_fops->delim_cnt);
            // Unlock
        //  mutex_unlock(afsk_data_fops->lock);
            printk(KERN_INFO "query %d",arg);
            return 0;
        default:
            printk(KERN_INFO "Invalid cmd");
            return -EINVAL;
    }
}
// Jordan's code end

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

```
};

MODULE_DEVICE_TABLE(of, afsk_of_match);

static struct platform_driver afsk_driver = {
    .probe = afsk_probe,
    .remove = afsk_remove,
    .driver = {
           .name = "bcm2835-afsk",
           .owner = THIS_MODULE,
           .of_match_table = afsk_of_match,
           },
};

module_platform_driver(afsk_driver);

MODULE_DESCRIPTION("AFSK pin modulator");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("AFSK");
```