

Nicholas LaJoie, ECE 331, HW X

```
// Author: Nicholas LaJoie
// ECE 331 - HW8
// March 29, 2017
```

Makefile:

```
-----
obj-m += morse.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
```

Kernel Code:

```
-----
// A. Sheaff 3/14/2017
// Morse code kernel driver
// GPIO4 is active low enable
// GPIO17 is active high BPSK encoded morse data

// Note: Additional functionality written by Nicholas LaJoie for ECE 331
// Current Stage: Can successfully open, write to, and close the /dev/morse file
```

```
#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 <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 "morse.h"
```

```
// Function Declarations
```

```
static int morse_open(struct inode *inode, struct file *filp);
static ssize_t morse_write(struct file *filp, const char __user *ubuf, size_t s, loff_t *o);
static int morse_release(struct inode *inode, struct file *filp);
```

```
struct morse_moddat_t *morse_dat=NULL;           // Data to be passed around the calls
```

```
// Data to be "passed" around to various functions
```

```
struct morse_moddat_t {
    int major;                // Device major number
    struct class *morse_class; // Class for auto /dev population
    struct device *morse_dev;  // Device for auto /dev population
    struct gpio_desc *enable;  // gpiod Enable pin
    struct gpio_desc *bm;      // gpiod BPSK Morse
    struct gpio_desc *shdn;    // Shutdown pin
};
```

```

    int irq;                                // Shutdown IRQ
};

// File operations for the morse device
static const struct file_operations morse_fops = {
    .owner = THIS_MODULE,                    // Prevents module from being unloaded while in use
    .open = morse_open,                      // Open file
    .write = morse_write,                    // Write to file
    .release = morse_release,                // Release (close) file
};

/***** File Operations Functions *****/
// Open morse device file
static int morse_open(struct inode *inode, struct file *filp)
{
    // Only accept write-only files
    if (!(filp->f_flags & O_WRONLY)) {
        return -EINVAL;
    }
    printk(KERN_INFO "File opened successfully!\n");
    return 0;
}

// Write to morse device file
static ssize_t morse_write(struct file *filp, const char __user *ubuf, size_t s, loff_t *o)
{
    int err;                                // Error checking value
    char *kbuf;                              // Kernel buffer

    // Allocate kernel buffer memory
    kbuf = (char *)kmalloc(s+1, GFP_KERNEL);
    if (kbuf == NULL) {
        printk(KERN_INFO "Memory allocation failed!\n");
        return -ENOMEM;
    }

    // Clear buffer
    memset(kbuf, 0, s+1);
    if (kbuf == NULL) {
        printk(KERN_INFO "Memset failed!\n");
        return -ENOMEM;
    }

    // Copy userspace data
    err = copy_from_user(kbuf, ubuf, s);
    if (err) {
        printk(KERN_INFO "Copying from userspace failed!\n");
        kfree(kbuf);
        kbuf = NULL;
        return -EFAULT;
    }

    // Print copied data
    printk(KERN_INFO "%s\n", kbuf);

    // Clean up
    kfree(kbuf);
    kbuf = NULL;
    return s; // Return size_t variable
}

// Close morse device file
static int morse_release(struct inode *inode, struct file *filp)

```

```
{
    printk(KERN_INFO "File closed successfully!\n");
    return 0;
}

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

static struct gpio_desc *morse_dt_obtain_pin(struct device *dev, struct device_node *parent, char *name, int init_val)
{
    struct device_node *child=NULL;
    struct gpio_desc *gpiod_pin=NULL;
    char *label=NULL;
    int pin=-1;
    int ret=-1;

    // Find the child node - release with of_node_put()
    child=of_get_child_by_name(parent,name);
    if (child==NULL) {
        printk(KERN_INFO "No device child\n");
        return NULL;
    }
    // Get the child pin number - Does not appear to need to be released
    pin=of_get_named_gpio(child,"gpios",0);
    if (pin<0) {
        printk(KERN_INFO "no GPIO pin\n");
        of_node_put(child);
        return NULL;
    }
    printk(KERN_INFO "Found %s pin %d\n",name,pin);
    // Verify the pin is OK
    if (!gpio_is_valid(pin)) {
        of_node_put(child);
        return NULL;
    }
    // Get the of string tied to pin - Does not appear to need to be released
    ret=of_property_read_string(child,"label",(const char **)&label);
    if (ret<0) {
        printk(KERN_INFO "Cannot find label\n");
        of_node_put(child);
        return NULL;
    }
    // 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_HIGH,label);
        if (ret<0) {
            dev_err(dev,"Cannot allocate gpio pin\n");
            of_node_put(child);
            return NULL;
        }
    } else {
        ret=devm_gpio_request_one(dev,pin,GPIOF_IN,label);
        if (ret<0) {
            dev_err(dev,"Cannot allocate gpio pin\n");
            of_node_put(child);
            return NULL;
        }
    }
}
```

```

    }

    // Release the device node
    of_node_put(child);

    // Get the gpio pin struct
    gpiod_pin=gpio_to_desc(pin);
    if (gpiod_pin==NULL) {
        of_node_put(child);
        devm_gpio_free(dev,pin);
        printk(KERN_INFO "Failed to acquire enable gpio\n");
        return NULL;
    }

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

    return gpiod_pin;
}

// 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 morse_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;          // Device associated with platform
    struct device_node *dn=NULL;              // Start of my device tree
    int ret;                                  // Return value

    // Allocate device driver data and save
    morse_dat=kmalloc(sizeof(struct morse_moddat_t),GFP_ATOMIC);
    if (morse_dat==NULL) {
        printk(KERN_INFO "Memory allocation failed\n");
        return -ENOMEM;
    }
    memset(morse_dat,0,sizeof(struct morse_moddat_t));

    // Tag in device data to the device
    dev_set_drvdata(dev,morse_dat);

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

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

    // Setup the device so the device special file is created with 0666 perms
    morse_dat->morse_class->devnode=morse_devnode;
    morse_dat->morse_dev=device_create(morse_dat->morse_class,NULL,MKDEV(morse_dat->major,
0),(void *)morse_dat,"morse");
    if (IS_ERR(morse_dat->morse_dev)) {

```

```

        printk(KERN_INFO "Failed to create device file\n");
        ret=PTR_ERR(morse_dat->morse_dev);
        goto fail;
    }

    // Find my device node
    dn=of_find_node_by_name(NULL,"morse");
    if (dn==NULL) {
        printk(KERN_INFO "Cannot find device\n");
        ret=-ENODEV;
        goto fail;
    }
    morse_dat->enable=morse_dt_obtain_pin(dev,dn,"Enable",1);
    if (morse_dat->enable==NULL) {
        goto fail;
    }
    morse_dat->bm=morse_dt_obtain_pin(dev,dn,"BPSK_Morse",0);
    if (morse_dat->bm==NULL) {
        goto fail;
    }
    morse_dat->shdn=morse_dt_obtain_pin(dev,dn,"Shutdown",-1);
    if (morse_dat->shdn==NULL) {
        goto fail;
    }

    // Release the device node
    if (dn) of_node_put(dn);

    // Initialize the output pins - should already be done above....
    gpiod_set_value(morse_dat->enable,1);
    gpiod_set_value(morse_dat->bm,0);

    // Get the IRQ # tagged with the input shutdown pin
    /*morse_dat->irq=gpiod_to_irq(morse_dat->shdn);
    if (morse_dat->irq<0) {
        printk(KERN_INFO "Failed to get shutdown IRQ #\n");
        ret=-ENODEV;
        goto fail;
    }
    printk(KERN_INFO "IRQ: %d\n",morse_dat->irq);
    // Actually request and register a handler
    ret=request_irq(morse_dat->irq,morse_irq,IRQF_TRIGGER_RISING,"morse#shutdown",(void *)
morse_dat);
    if (ret<0) {
        printk(KERN_INFO "Failed to register shutdown IRQ\n");
        ret=-ENODEV;
        goto fail;
    }
    printk(KERN_INFO "IRQ Registered\n");
*/
    printk(KERN_INFO "Registered\n");
    dev_info(dev, "Initialized");
    return 0;

fail:
    if (morse_dat->shdn) {
        devm_gpio_free(dev,desc_to_gpio(morse_dat->shdn));
        gpiod_put(morse_dat->shdn);
    }
    if (morse_dat->bm) {
        devm_gpio_free(dev,desc_to_gpio(morse_dat->bm));
        gpiod_put(morse_dat->bm);
    }

```

Nicholas LaJoie, ECE 331, HW X

```
    if (morse_dat->enable) {
        devm_gpio_free(dev, desc_to_gpio(morse_dat->enable));
        gpiod_put(morse_dat->enable);
    }
    if (morse_dat->morse_class) class_destroy(morse_dat->morse_class);
    if (morse_dat->major) unregister_chrdev(morse_dat->major, "morse");
    dev_set_drvdata(dev, NULL);
    kfree(morse_dat);
    printk(KERN_INFO "Morse Failed\n");
    return ret;
}

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

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

    if (morse_dat->irq > 0) free_irq(morse_dat->irq, (void *)morse_dat);

    if (morse_dat->shdn) {
        devm_gpio_free(dev, desc_to_gpio(morse_dat->shdn));
        gpiod_put(morse_dat->shdn);
    }
    if (morse_dat->bm) {
        devm_gpio_free(dev, desc_to_gpio(morse_dat->bm));
        gpiod_put(morse_dat->bm);
    }
    if (morse_dat->enable) {
        devm_gpio_free(dev, desc_to_gpio(morse_dat->enable));
        gpiod_put(morse_dat->enable);
    }

    // Release the device
    device_destroy(morse_dat->morse_class, MKDEV(morse_dat->major, 0));

    // Release the class
    class_destroy(morse_dat->morse_class);

    // Release the character device
    unregister_chrdev(morse_dat->major, "morse");

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

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

    return 0;
}

// From Caf on StackOverflow - "Shutdown (embedded) linux from kernel-space"
/*static char *poweroff_argv[]={
    "/sbin/poweroff", NULL,
};*/

/*static irqreturn_t morse_irq(int irq, void *data)
{

```

Nicholas LaJoie, ECE 331, HW X

```
    struct morse_moddat_t *morse_dat=(struct morse_moddat_t *)data;
//    int p;

//    call_usermodehelper(poweroff_argv[0],poweroff_argv,NULL,UMH_NO_WAIT);
    printk(KERN_INFO "In IRQ\n");
//    p=gpiod_get_value(morse_dat->shdn);
//    printk(KERN_INFO "GPIO: %d\n",p);
    return IRQ_HANDLED;
}*/

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

MODULE_DEVICE_TABLE(of, morse_of_match);

static struct platform_driver morse_driver = {
    .probe = morse_probe,
    .remove = morse_remove,
    .driver = {
        .name = "bcm2835-morse",
        .owner = THIS_MODULE,
        .of_match_table = morse_of_match,
    },
};

module_platform_driver(morse_driver);

MODULE_DESCRIPTION("Morse pin modulator");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Morse");
//MODULE_ALIAS("platform:morse-bcm2835");
```

Userspace Test Program:

```
-----
// Author: Nicholas LaJoie
// ECE 331 â\200\223 HW8
// March 28, 2017
// Description: Test program for interacting with the "morse" kernel module
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
```

```
int main(int argc, char * argv[])
{
    int fd, ret;
    char *test_msg;
    int msg_size;

    if (argc != 2) {
        perror("Usage: ./test \"<your string>\"\n");
        return 1;
    }

    // Test message is the command line argument
```

Nicholas LaJoie, ECE 331, HW X

```
test_msg = argv[1];
msg_size = strlen(test_msg);
fd = open("/dev/morse", O_WRONLY);

// Error checking
if (fd < 0) {
    perror("Failed to open morse device!\n");
    return 2;
}

// Write in kernel space
ret = write(fd, test_msg, msg_size);
if (ret < 0) {
    perror("Failed to write to morse device!\n");
    return 3;
}

// Clean up
close(fd);

return 0;
}
```