Exploring BeagleBone: LKMs (by Derek Molloy)

V1.0

Generated by Doxygen 1.8.8

Sat Apr 25 2015 15:26:35

Contents

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

Chapter 2

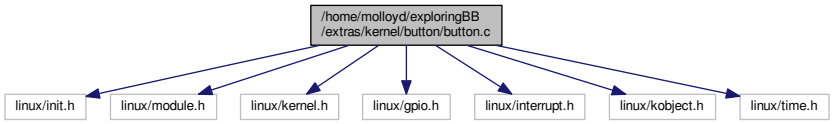
File Documentation

2.1 /home/molloyd/exploringBB/extras/kernel/button/button.c File Reference

A kernel module for controlling a button (or any signal) that is connected to a GPIO. It has full support for interrupts and for sysfs entries so that an interface can be created to the button or the button can be configured from Linux userspace. The sysfs entry appears at /sys/ebb/gpio115.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kobject.h>
#include <linux/time.h>
```

Include dependency graph for button.c:



Macros

- #define DEBOUNCE_TIME 200

  The default bounce time – 200ms.

Functions

- MODULE_LICENSE ("GPL")
- MODULE_AUTHOR ("Derek Molloy")
- MODULE_DESCRIPTION ("A simple Linux GPIO Button LKM for the BBB")
- MODULE_VERSION ("0.1")
- module_param (isRising, bool, S_IRUGO)

  Param desc. S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (isRising," Rising edge = 1 (default), Falling edge = 0")

  parameter description

- module_param (gpioButton, uint, S_IRUGO)

  Param desc. S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (gpioButton," GPIO Button number (default=115)")

  parameter description
- module_param (gpioLED, uint, S_IRUGO)

  Param desc. S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (gpioLED," GPIO LED number (default=49)")

  parameter description
- static irq_handler_t ebbgpio_irq_handler (unsigned int irq, void *dev_id, struct pt_regs *regs)

  Function prototype for the custom IRQ handler function – see below for the implementation.
- static ssize_t numberPresses_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  A callback function to output the numberPresses variable.
- static ssize_t numberPresses_store (struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)

  A callback function to read in the numberPresses variable.
- static ssize_t ledOn_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  Displays if the LED is on or off.
- static ssize_t lastTime_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  Displays the last time the button was pressed – manually output the date (no localization)
- static ssize_t diffTime_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  Display the time difference in the form secs.nanosecs to 9 places.
- static ssize_t isDebounce_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  Displays if button debouncing is on or off.
- static ssize_t isDebounce_store (struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)

  Stores and sets the debounce state.
- static int __init ebbButton_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.
- static void __exit ebbButton_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.
- module_init (ebbButton_init)
- module_exit (ebbButton_exit)

Variables

- static bool isRising = 1

  Rising edge is the default IRQ property.
- static unsigned int gpioButton = 115

  Default GPIO is 115.
- static unsigned int gpioLED = 49

  Default GPIO is 49.
- static char gpioName [8] = "gpioXXX"

  Null terminated default string – just in case.
- static int irqNumber

  Used to share the IRQ number within this file.
- static int numberPresses = 0

  For information, store the number of button presses.
- static bool ledOn = 0

  Is the LED on or off? Used to invert its state (off by default)

- static bool isDebounce = 1

  Use to store the debounce state (on by default)
- static struct timespec ts_last
  ts_current ts_diff

  timespecs from linux/time.h (has nano precision)
- static struct kobj_attribute count_attr = __ATTR(numberPresses, 0666, numberPresses_show, numberPresses_store)
- static struct kobj_attribute debounce_attr = __ATTR(isDebounce, 0666, isDebounce_show, isDebounce_store)
- static struct kobj_attribute ledon_attr = __ATTR_RO(ledOn)

  the ledon kobject attr
- static struct kobj_attribute time_attr = __ATTR_RO(lastTime)

  the last time pressed kobject attr
- static struct kobj_attribute diff_attr = __ATTR_RO(diffTime)

  the difference in time attr
- static struct attribute * ebb_attrs []
- static struct attribute_group attr_group
- static struct kobject * ebb_kobj

2.1.1 Detailed Description

A kernel module for controlling a button (or any signal) that is connected to a GPIO. It has full support for interrupts and for sysfs entries so that an interface can be created to the button or the button can be configured from Linux userspace. The sysfs entry appears at /sys/ebb/gpio115.

Author

Derek Molloy

Date

19 April 2015

See also

http://www.derekmolloy.ie/

2.1.2 Macro Definition Documentation

2.1.2.1 #define DEBOUNCE_TIME 200

The default bounce time – 200ms.

2.1.3 Function Documentation

2.1.3.1 static ssize_t diffTime_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

Display the time difference in the form secs.nanosecs to 9 places.

```
83                                                                          {
84     return sprintf(buf, "%lu.%.9lu\n", ts_diff.tv_sec, ts_diff.tv_nsec);
85 }
```

2.1.3.2 static void __exit ebbButton_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

```
212                                     {
213    printk(KERN_INFO "EBB Button: The button was pressed %d times\n",
       numberPresses);
214    kobject_put(ebb_kobj);                   // clean up -- remove the kobject sysfs entry
215    gpio_set_value(gpioLED, 0);              // Turn the LED off, makes it clear the device was
       unloaded
216    gpio_unexport(gpioLED);                  // Unexport the LED GPIO
217    free_irq(irqNumber, NULL);               // Free the IRQ number, no *dev_id required in this
       case
218    gpio_unexport(gpioButton);               // Unexport the Button GPIO
219    gpio_free(gpioLED);                      // Free the LED GPIO
220    gpio_free(gpioButton);                   // Free the Button GPIO
221    printk(KERN_INFO "EBB Button: Goodbye from the EBB Button LKM!\n");
222 }
```

2.1.3.3 static int __init ebbButton_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.

Returns

returns 0 if successful

GPIO numbers and IRQ numbers are not the same! This function performs the mapping for us

```
153                                     {
154    int result = 0;
155    unsigned long IRQflags = IRQF_TRIGGER_RISING;      // The default is a rising-edge interrupt
156
157    printk(KERN_INFO "EBB Button: Initializing the EBB Button LKM\n");
158    sprintf(gpioName, "gpio%d", gpioButton);           // Create the gpio115 name for
       /sys/ebb/gpio115
159
160    // create the kobject sysfs entry at /sys/ebb -- probably not an ideal location!
161    ebb_kobj = kobject_create_and_add("ebb", kernel_kobj->parent); // kernel_kobj points to
       /sys/kernel
162    if(!ebb_kobj){
163       printk(KERN_ALERT "EBB Button: failed to create kobject mapping\n");
164       return -ENOMEM;
165    }
166    // add the attributes to /sys/ebb/ -- for example, /sys/ebb/gpio115/numberPresses
167    result = sysfs_create_group(ebb_kobj, &attr_group);
168    if(result) {
169       printk(KERN_ALERT "EBB Button: failed to create sysfs group\n");
170       kobject_put(ebb_kobj);                          // clean up -- remove the kobject sysfs entry
171       return result;
172    }
173    getnstimeofday(&ts_last);                          // set the last time to be the current time
174    ts_diff = timespec_sub(ts_last, ts_last);          // set the initial time difference to be 0
175
176    // Going to set up the LED. It is a GPIO in output mode and will be on by default
177    ledOn = true;
178    gpio_request(gpioLED, "sysfs");          // gpioLED is hardcoded to 49, request it
179    gpio_direction_output(gpioLED, ledOn);   // Set the gpio to be in output mode and on
180 // gpio_set_value(gpioLED, ledOn);          // Not required as set by line above (here for reference)
181    gpio_export(gpioLED, false);             // Causes gpio49 to appear in /sys/class/gpio
182                                             // the bool argument prevents the direction from being changed
183    gpio_request(gpioButton, "sysfs");       // Set up the gpioButton
184    gpio_direction_input(gpioButton);        // Set the button GPIO to be an input
185    gpio_set_debounce(gpioButton, DEBOUNCE_TIME); // Debounce the button with a delay
       of 200ms
186    gpio_export(gpioButton, false);          // Causes gpio115 to appear in /sys/class/gpio
187                                             // the bool argument prevents the direction from being changed
188
189    // Perform a quick test to see that the button is working as expected on LKM load
190    printk(KERN_INFO "EBB Button: The button state is currently: %d\n", gpio_get_value(
       gpioButton));
191
193    irqNumber = gpio_to_irq(gpioButton);
```

```
194    printk(KERN_INFO "EBB Button: The button is mapped to IRQ: %d\n", irqNumber);
195
196    if(!isRising){                          // If the kernel parameter isRising=0 is supplied
197       IRQflags = IRQF_TRIGGER_FALLING;     // Set the interrupt to be on the falling edge
198    }
199    // This next call requests an interrupt line
200    result = request_irq(irqNumber,             // The interrupt number requested
201                         (irq_handler_t) ebbgpio_irq_handler, // The pointer to the
       handler function below
202                         IRQflags,              // Use the custom kernel param to set interrupt type
203                         "ebb_button_handler",  // Used in /proc/interrupts to identify the owner
204                         NULL);                 // The *dev_id for shared interrupt lines, NULL is okay
205    return result;
206 }
```

2.1.3.4 static irq_handler_t ebbgpio_irq_handler (unsigned int *irq*, void * *dev_id*, struct pt_regs * *regs*) [static]

Function prototype for the custom IRQ handler function – see below for the implementation.

The GPIO IRQ Handler function This function is a custom interrupt handler that is attached to the GPIO above. The same interrupt handler cannot be invoked concurrently as the interrupt line is masked out until the function is complete. This function is static as it should not be invoked directly from outside of this file.

Parameters

irq	the IRQ number that is associated with the GPIO – useful for logging.
dev_id	the *dev_id that is provided – can be used to identify which device caused the interrupt Not used in this example as NULL is passed.
regs	h/w specific register values – only really ever used for debugging. return returns IRQ_HAN↩ DLED if successful – should return IRQ_NONE otherwise.

```
234                                                                                    {
235    ledOn = !ledOn;                          // Invert the LED state on each button press
236    gpio_set_value(gpioLED, ledOn);          // Set the physical LED accordingly
237    getnstimeofday(&ts_current);             // Get the current time as ts_current
238    ts_diff = timespec_sub(ts_current, ts_last);   // Determine the time difference between last 2
       presses
239    ts_last = ts_current;                    // Store the current time as the last time ts_last
240    printk(KERN_INFO "EBB Button: The button state is currently: %d\n", gpio_get_value(
       gpioButton));
241    numberPresses++;                         // Global counter, will be outputted when the module
       is unloaded
242    return (irq_handler_t) IRQ_HANDLED;  // Announce that the IRQ has been handled correctly
243 }
```

2.1.3.5 static ssize_t isDebounce_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

Displays if button debouncing is on or off.

```
88                                                                                     {
89    return sprintf(buf, "%d\n", isDebounce);
90 }
```

2.1.3.6 static ssize_t isDebounce_store (struct kobject * *kobj*, struct kobj_attribute * *attr*, const char * *buf*, size_t *count*) [static]

Stores and sets the debounce state.

```
93
          {
94    unsigned int temp;
95    sscanf(buf, "%du", &temp);               // use a temp varable for correct int->bool
96    gpio_set_debounce(gpioButton,0);
97    isDebounce = temp;
98    if(isDebounce) { gpio_set_debounce(gpioButton,
      DEBOUNCE_TIME);
```

```
99        printk(KERN_INFO "EBB Button: Debounce on\n");
100     }
101     else { gpio_set_debounce(gpioButton, 0);  // set the debounce time to 0
102        printk(KERN_INFO "EBB Button: Debounce off\n");
103     }
104     return count;
105 }
```

2.1.3.7 static ssize_t lastTime_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

Displays the last time the button was pressed – manually output the date (no localization)

```
77                                                                                  {
78     return sprintf(buf, "%.2lu:%.2lu:%.2lu:%.9lu \n", (ts_last.tv_sec/3600)%24,
79            (ts_last.tv_sec/60) % 60, ts_last.tv_sec % 60, ts_last.tv_nsec );
80 }
```

2.1.3.8 static ssize_t ledOn_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

Displays if the LED is on or off.

```
72                                                                          {
73     return sprintf(buf, "%d\n", ledOn);
74 }
```

2.1.3.9 MODULE_AUTHOR ("Derek Molloy")

2.1.3.10 MODULE_DESCRIPTION ("A simple Linux GPIO Button LKM for the BBB")

2.1.3.11 module_exit (ebbButton_exit)

2.1.3.12 module_init (ebbButton_init)

2.1.3.13 MODULE_LICENSE ("GPL")

2.1.3.14 module_param (isRising , bool , S_IRUGO)

Param desc. S_IRUGO can be read/not changed.

2.1.3.15 module_param (gpioButton , uint , S_IRUGO)

Param desc. S_IRUGO can be read/not changed.

2.1.3.16 module_param (gpioLED , uint , S_IRUGO)

Param desc. S_IRUGO can be read/not changed.

2.1.3.17 MODULE_PARM_DESC (isRising , " Rising *edge* = 1 (default))

parameter description

2.1.3.18 MODULE_PARM_DESC (gpioButton , " GPIO Button number (default=115)")

parameter description

2.1.3.19 MODULE_PARM_DESC (gpioLED , " GPIO LED number (default=49)")

parameter description

2.1.3.20 MODULE_VERSION ("0.1")

2.1.3.21 static ssize_t numberPresses_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

A callback function to output the numberPresses variable.

Parameters

kobj	represents a kernel object device that appears in the sysfs filesystem
attr	the pointer to the kobj_attribute struct
buf	the buffer to which to write the number of presses

Returns

return the total number of characters written to the buffer (excluding null)

```
54                                                                                  {
55     return sprintf(buf, "%d\n", numberPresses);
56 }
```

2.1.3.22 static ssize_t numberPresses_store (struct kobject * *kobj*, struct kobj_attribute * *attr*, const char * *buf*, size_t *count*) [static]

A callback function to read in the numberPresses variable.

Parameters

kobj	represents a kernel object device that appears in the sysfs filesystem
attr	the pointer to the kobj_attribute struct
buf	the buffer from which to read the number of presses (e.g., reset to 0).
count	the number characters in the buffer

Returns

return should return the total number of characters used from the buffer

```
66                                                                  {
67     sscanf(buf, "%du", &numberPresses);
68     return count;
69 }
```

2.1.4 Variable Documentation

2.1.4.1 struct attribute_group attr_group [static]

Initial value:

```
= {
        .name  = gpioName,
        .attrs = ebb_attrs,
}
```

The attribute group uses the attribute array and a name, which is exposed on sysfs – in this case it is gpio115, which is automatically defined in the ebbButton_init() function below using the custom kernel parameter that can be passed when the module is loaded.

2.1.4.2 struct kobj_attribute count_attr = __ATTR(numberPresses, 0666, numberPresses_show, numberPresses_store) [static]

Use these helper macros to define the name and access levels of the kobj_attributes The kobj_attribute has an attribute attr (name and mode), show and store function pointers The count variable is associated with the number↩ Presses variable and it is to be exposed with mode 0666 using the numberPresses_show and numberPresses_store functions above

2.1.4.3 struct kobj_attribute debounce_attr = __ATTR(isDebounce, 0666, isDebounce_show, isDebounce_store) [static]

2.1.4.4 struct kobj_attribute diff_attr = __ATTR_RO(diffTime) [static]

the difference in time attr

2.1.4.5 struct attribute* ebb_attrs[] [static]

Initial value:

```
= {
      &count_attr.attr,
      &ledon_attr.attr,
      &time_attr.attr,
      &diff_attr.attr,
      &debounce_attr.attr,
      NULL,
}
```

The ebb_attrs[] is an array of attributes that is used to create the attribute group below. The attr property of the kobj_attribute is used to extract the attribute struct

2.1.4.6 struct kobject* ebb_kobj [static]

2.1.4.7 unsigned int gpioButton = 115 [static]

Default GPIO is 115.

2.1.4.8 unsigned int gpioLED = 49 [static]

Default GPIO is 49.

2.1.4.9 char gpioName[8] = "gpioXXX" [static]

Null terminated default string – just in case.

2.1.4.10 int irqNumber [static]

Used to share the IRQ number within this file.

2.1.4.11 bool isDebounce = 1 [static]

Use to store the debounce state (on by default)

2.1.4.12 bool isRising = 1 [static]

Rising edge is the default IRQ property.

2.1.4.13 bool ledOn = 0 [static]

Is the LED on or off? Used to invert its state (off by default)

2.1.4.14 struct kobj_attribute ledon_attr = __ATTR_RO(ledOn) [static]

the ledon kobject attr

The __ATTR_RO macro defines a read-only attribute. There is no need to identify that the function is called _show, but it must be present. __ATTR_WO can be used for a write-only attribute but only in Linux 3.11.x on.

2.1.4.15 int numberPresses = 0 [static]

For information, store the number of button presses.

2.1.4.16 struct kobj_attribute time_attr = __ATTR_RO(lastTime) [static]

the last time pressed kobject attr

2.1.4.17 struct timespec ts_last ts_current ts_diff [static]

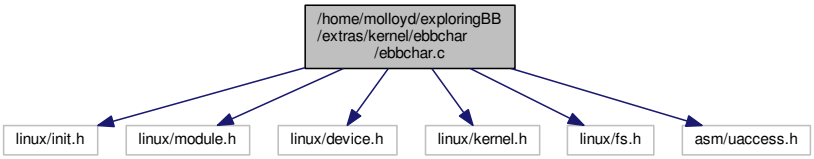
timespecs from linux/time.h (has nano precision)

2.2 /home/molloyd/exploringBB/extras/kernel/ebbchar/ebbchar.c File Reference

An introductory character driver to support the second article of my series on Linux loadable kernel module (LKM) development. This module maps to /dev/ebbchar and comes with a helper C program that can be run in Linux user space to communicate with this the LKM.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
```

Include dependency graph for ebbchar.c:



Macros

- #define DEVICE_NAME "ebbchar"

  The device will appear at /dev/ebbchar using this value.
- #define CLASS_NAME "ebb"

  The device class – this is a character device driver.

Functions

- MODULE_LICENSE ("GPL")

  The license type – this affects available functionality.
- MODULE_AUTHOR ("Derek Molloy")

  The author – visible when you use modinfo.
- MODULE_DESCRIPTION ("A simple Linux char driver for the BBB")

  The description – see modinfo.
- MODULE_VERSION ("0.1")

  A version number to inform users.
- static int dev_open (struct inode ∗inodep, struct file ∗filep)

  The device open function that is called each time the device is opened This will only increment the numberOpens counter in this case.
- static int dev_release (struct inode ∗inodep, struct file ∗filep)

  The device release function that is called whenever the device is closed/released by the userspace program.
- static ssize_t dev_read (struct file ∗filep, char ∗buffer, size_t len, loff_t ∗offset)

  This function is called whenever device is being read from user space i.e. data is being sent from the device to the user. In this case is uses the copy_to_user() function to send the buffer string to the user and captures any errors.
- static ssize_t dev_write (struct file ∗filep, const char ∗buffer, size_t len, loff_t ∗offset)

  This function is called whenever the device is being written to from user space i.e. data is sent to the device from the user. The data is copied to the message[] array in this LKM using the sprintf() function along with the length of the string.
- static int __init ebbchar_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.
- static void __exit ebbchar_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.
- module_init (ebbchar_init)

  A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)
- module_exit (ebbchar_exit)

Variables

- static int majorNumber

  Stores the device number – determined automatically.
- static char message [256] = {0}

  Memory for the string that is passed from userspace.
- static short size_of_message

  Used to remember the size of the string stored.
- static int numberOpens = 0

  Counts the number of times the device is opened.
- static struct class ∗ ebbcharClass = NULL

The device-driver class struct pointer.

- static struct device * ebbcharDevice = NULL

  The device-driver device struct pointer.

- static struct file_operations fops

  Devices are represented as file structure in the kernel. The file_operations structure from /linux/fs.h lists the callback functions that you wish to associated with your file operations using a C99 syntax structure. char devices usually implement open, read, write and release calls.

2.2.1 Detailed Description

An introductory character driver to support the second article of my series on Linux loadable kernel module (LKM) development. This module maps to /dev/ebbchar and comes with a helper C program that can be run in Linux user space to communicate with this the LKM.

Author

Derek Molloy

Date

7 April 2015

Version

0.1

See also

http://www.derekmolloy.ie/ for a full description and follow-up descriptions.

2.2.2 Macro Definition Documentation

2.2.2.1 #define CLASS_NAME "ebb"

The device class – this is a character device driver.

2.2.2.2 #define DEVICE_NAME "ebbchar"

The device will appear at /dev/ebbchar using this value.

2.2.3 Function Documentation

2.2.3.1 static int dev_open (struct inode * *inodep*, struct file * *filep*) [static]

The device open function that is called each time the device is opened This will only increment the numberOpens counter in this case.

Parameters

inodep	A pointer to an inode object (defined in linux/fs.h)

filep	A pointer to a file object (defined in linux/fs.h)

```
107                                                         {
108    numberOpens++;
109    printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n", numberOpens);
110    return 0;
111 }
```

2.2.3.2 static ssize_t dev_read (struct file * *filep*, char * *buffer*, size_t *len*, loff_t * *offset*) [static]

This function is called whenever device is being read from user space i.e. data is being sent from the device to the user. In this case is uses the copy_to_user() function to send the buffer string to the user and captures any errors.

Parameters

filep	A pointer to a file object (defined in linux/fs.h)
buffer	The pointer to the buffer to which this function writes the data
len	The length of the b
offset	The offset if required

```
121                                                                                 {
122    int error_count = 0;
123    // copy_to_user has the format ( * to, *from, size) and returns 0 on success
124    error_count = copy_to_user(buffer, message, size_of_message);
125
126    if (error_count==0){            // if true then have success
127       printk(KERN_INFO "EBBChar: Sent %d characters to the user\n",
       size_of_message);
128       return (size_of_message=0);  // clear the position to the start and return 0
129    }
130    else {
131       printk(KERN_INFO "EBBChar: Failed to send %d characters to the user\n", error_count);
132       return -EFAULT;              // Failed -- return a bad address message (i.e. -14)
133    }
134 }
```

2.2.3.3 static int dev_release (struct inode * *inodep*, struct file * *filep*) [static]

The device release function that is called whenever the device is closed/released by the userspace program.

Parameters

inodep	A pointer to an inode object (defined in linux/fs.h)
filep	A pointer to a file object (defined in linux/fs.h)

```
156                                                         {
157    printk(KERN_INFO "EBBChar: Device successfully closed\n");
158    return 0;
159 }
```

2.2.3.4 static ssize_t dev_write (struct file * *filep*, const char * *buffer*, size_t *len*, loff_t * *offset*) [static]

This function is called whenever the device is being written to from user space i.e. data is sent to the device from the user. The data is copied to the message[] array in this LKM using the sprintf() function along with the length of the string.

Parameters

filep	A pointer to a file object
buffer	The buffer to that contains the string to write to the device
len	The length of the array of data that is being passed in the const char buffer
offset	The offset if required

```
144                                                                                 {
145    sprintf(message, "%s(%d letters)", buffer, len);   // appending received string with its length
146    size_of_message = strlen(message);                 // store the length of the
       stored message
147    printk(KERN_INFO "EBBChar: Received %d characters from the user\n", len);
148    return len;
149 }
```

2.2.3.5 static void __exit ebbchar_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

```
94                                      {
95    device_destroy(ebbcharClass, MKDEV(majorNumber, 0));     // remove the device
96    class_unregister(ebbcharClass);                          // unregister the device class
97    class_destroy(ebbcharClass);                             // remove the device class
98    unregister_chrdev(majorNumber, DEVICE_NAME);             // unregister the major
       number
99    printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
100 }
```

2.2.3.6 static int __init ebbchar_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.

Returns

returns 0 if successful

```
58                                      {
59    printk(KERN_INFO "EBBChar: Initializing the EBBChar LKM\n");
60
61    // Try to dynamically allocate a major number for the device -- more difficult but worth it
62    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
63    if (majorNumber<0){
64       printk(KERN_ALERT "EBBChar failed to register a major number\n");
65       return majorNumber;
66    }
67    printk(KERN_INFO "EBBChar: registered correctly with major number %d\n",
      majorNumber);
68
69    // Register the device class
70    ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);
71    if (IS_ERR(ebbcharClass)){                // Check for error and clean up if there is
72       unregister_chrdev(majorNumber, DEVICE_NAME);
73       printk(KERN_ALERT "Failed to register device class\n");
74       return PTR_ERR(ebbcharClass);          // Correct way to return an error on a pointer
75    }
76    printk(KERN_INFO "EBBChar: device class registered correctly\n");
77
78    // Register the device driver
79    ebbcharDevice = device_create(ebbcharClass, NULL, MKDEV(
      majorNumber, 0), NULL, DEVICE_NAME);
80    if (IS_ERR(ebbcharDevice)){               // Clean up if there is an error
81       class_destroy(ebbcharClass);           // Repeated code but the alternative is goto
       statements
82       unregister_chrdev(majorNumber, DEVICE_NAME);
83       printk(KERN_ALERT "Failed to create the device\n");
84       return PTR_ERR(ebbcharDevice);
85    }
86    printk(KERN_INFO "EBBChar: device class created correctly\n"); // Made it! device was initialized
87    return 0;
88 }
```

2.2.3.7 MODULE_AUTHOR ("Derek Molloy")

The author – visible when you use modinfo.

2.2.3.8 MODULE_DESCRIPTION ("A simple Linux char driver for the BBB")

The description – see modinfo.

2.2.3.9 module_exit (ebbchar_exit)

2.2.3.10 module_init (ebbchar_init)

A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)

2.2.3.11 MODULE_LICENSE ("GPL")

The license type – this affects available functionality.

2.2.3.12 MODULE_VERSION ("0.1")

A version number to inform users.

2.2.4 Variable Documentation

2.2.4.1 struct class* ebbcharClass = NULL [static]

The device-driver class struct pointer.

2.2.4.2 struct device* ebbcharDevice = NULL [static]

The device-driver device struct pointer.

2.2.4.3 struct file_operations fops [static]

Initial value:

```
=
{
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .release = dev_release,
}
```

Devices are represented as file structure in the kernel. The file_operations structure from /linux/fs.h lists the callback functions that you wish to associated with your file operations using a C99 syntax structure. char devices usually implement open, read, write and release calls.

2.2.4.4 int majorNumber [static]

Stores the device number – determined automatically.

2.2.4.5 char message[256] = {0} [static]

Memory for the string that is passed from userspace.

2.2.4.6 int numberOpens = 0 [static]

Counts the number of times the device is opened.

2.2.4.7 short size_of_message [static]

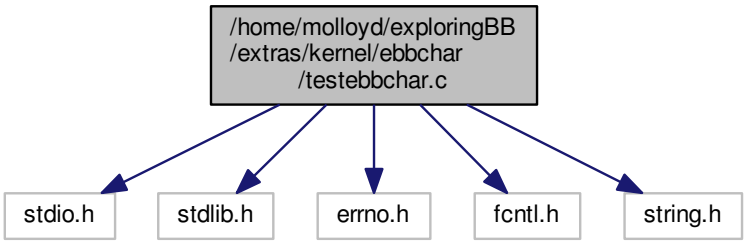
Used to remember the size of the string stored.

2.3 /home/molloyd/exploringBB/extras/kernel/ebbchar/testebbchar.c File Reference

A Linux user space program that communicates with the ebbchar.c LKM. It passes a string to the LKM and reads the response from the LKM. For this example to work the device must be called /dev/ebbchar.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
```

Include dependency graph for testebbchar.c:



Macros

- #define BUFFER_LENGTH 256

  The buffer length (crude but fine)

Functions

- int main ()

Variables

- static char receive [BUFFER_LENGTH]

  The receive buffer from the LKM.

2.3.1 Detailed Description

A Linux user space program that communicates with the ebbchar.c LKM. It passes a string to the LKM and reads the response from the LKM. For this example to work the device must be called /dev/ebbchar.

Author

Derek Molloy

Date

7 April 2015

Version

0.1

See also

http://www.derekmolloy.ie/ for a full description and follow-up descriptions.

2.3.2 Macro Definition Documentation

2.3.2.1 #define BUFFER_LENGTH 256

The buffer length (crude but fine)

2.3.3 Function Documentation

2.3.3.1 int main ( )

```
20          {
21    int ret, fd;
22    char stringToSend[BUFFER_LENGTH];
23    printf("Starting device test code example...\n");
24    fd = open("/dev/ebbchar", O_RDWR);             // Open the device with read/write access
25    if (fd < 0){
26       perror("Failed to open the device...");
27       return errno;
28    }
29    printf("Type in a short string to send to the kernel module:\n");
30    scanf("%[^\n]%*c", stringToSend);                // Read in a string (with spaces)
31    printf("Writing message to the device [%s].\n", stringToSend);
32    ret = write(fd, stringToSend, strlen(stringToSend)); // Send the string to the LKM
33    if (ret < 0){
34       perror("Failed to write the message to the device.");
35       return errno;
36    }
37
38    printf("Press ENTER to read back from the device...\n");
39    getchar();
40
41    printf("Reading from the device...\n");
42    ret = read(fd, receive, BUFFER_LENGTH);        // Read the response from the LKM
43    if (ret < 0){
44       perror("Failed to read the message from the device.");
45       return errno;
46    }
47    printf("The received message is: [%s]\n", receive);
48    printf("End of the program\n");
49    return 0;
50 }
```

2.3.4 Variable Documentation

2.3.4.1 char receive[BUFFER_LENGTH] [static]

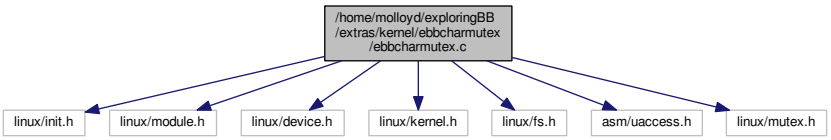
The receive buffer from the LKM.

2.4 /home/molloyd/exploringBB/extras/kernel/ebbcharmutex/ebbcharmutex.c File Reference

An introductory character driver to support the second article of my series on Linux loadable kernel module (LKM) development. This module maps to /dev/ebbchar and comes with a helper C program that can be run in Linux user space to communicate with this the LKM. This version has mutex locks to deal with synchronization problems.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/mutex.h>
```

Include dependency graph for ebbcharmutex.c:



Macros

- #define DEVICE_NAME "ebbchar"

  The device will appear at /dev/ebbchar using this value.
- #define CLASS_NAME "ebb"

  The device class – this is a character device driver.

Functions

- MODULE_LICENSE ("GPL")

  The license type – this affects available functionality.
- MODULE_AUTHOR ("Derek Molloy")

  The author – visible when you use modinfo.
- MODULE_DESCRIPTION ("A simple Linux char driver for the BBB")

  The description – see modinfo.
- MODULE_VERSION ("0.1")

  A version number to inform users.
- static DEFINE_MUTEX (ebbchar_mutex)

  Macro to declare a new mutex.
- static int dev_open (struct inode ∗, struct file ∗)

  The prototype functions for the character driver – must come before the struct definition.
- static int dev_release (struct inode ∗inodep, struct file ∗filep)

  The device release function that is called whenever the device is closed/released by the userspace program.
- static ssize_t dev_read (struct file ∗filep, char ∗buffer, size_t len, loff_t ∗offset)

  This function is called whenever device is being read from user space i.e. data is being sent from the device to the user. In this case is uses the copy_to_user() function to send the buffer string to the user and captures any errors.
- static ssize_t dev_write (struct file ∗filep, const char ∗buffer, size_t len, loff_t ∗offset)

This function is called whenever the device is being written to from user space i.e. data is sent to the device from the user. The data is copied to the message[] array in this LKM using message[x] = buffer[x].

- static int __init ebbchar_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.

- static void __exit ebbchar_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

- module_init (ebbchar_init)

  A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)

- module_exit (ebbchar_exit)

Variables

- static int majorNumber

  Store the device number – determined automatically.

- static char message [256] = {0}

  Memory for the string that is passed from userspace.

- static short size_of_message

  Used to remember the size of the string stored.

- static int numberOpens = 0

  Counts the number of times the device is opened.

- static struct class * ebbcharClass = NULL

  The device-driver class struct pointer.

- static struct device * ebbcharDevice = NULL

  The device-driver device struct pointer.

- static struct file_operations fops

2.4.1 Detailed Description

An introductory character driver to support the second article of my series on Linux loadable kernel module (LKM) development. This module maps to /dev/ebbchar and comes with a helper C program that can be run in Linux user space to communicate with this the LKM. This version has mutex locks to deal with synchronization problems.

Author

Derek Molloy

Date

7 April 2015

Version

0.1

See also

http://www.derekmolloy.ie/ for a full description and follow-up descriptions.

2.4.2 Macro Definition Documentation

2.4.2.1 #define CLASS_NAME "ebb"

The device class – this is a character device driver.

2.4.2.2 #define DEVICE_NAME "ebbchar"

The device will appear at /dev/ebbchar using this value.

2.4.3 Function Documentation

2.4.3.1 static DEFINE_MUTEX (ebbchar_mutex) [static]

Macro to declare a new mutex.

2.4.3.2 static int dev_open (struct inode * *inodep*, struct file * *filep*) [static]

The prototype functions for the character driver – must come before the struct definition.

The device open function that is called each time the device is opened This will only increment the numberOpens counter in this case.

Parameters

inodep	A pointer to an inode object (defined in linux/fs.h)
filep	A pointer to a file object (defined in linux/fs.h)

```
113                                                     {
114
115     if(!mutex_trylock(&ebbchar_mutex)){                 // Try to acquire the mutex (returns 0 on fail)
116          printk(KERN_ALERT "EBBChar: Device in use by another process");
117          return -EBUSY;
118     }
119     numberOpens++;
120     printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n", numberOpens);
121     return 0;
122 }
```

2.4.3.3 static ssize_t dev_read (struct file * *filep*, char * *buffer*, size_t *len*, loff_t * *offset*) [static]

This function is called whenever device is being read from user space i.e. data is being sent from the device to the user. In this case is uses the copy_to_user() function to send the buffer string to the user and captures any errors.

Parameters

filep	A pointer to a file object (defined in linux/fs.h)
buffer	The pointer to the buffer to which this function writes the data
len	The length of the b
offset	The offset if required

```
132                                                                                 {
133     int error_count = 0;
134     // copy_to_user has the format ( * to, *from, size) and returns 0 on success
135     error_count = copy_to_user(buffer, message, size_of_message);
136
137     if (error_count==0){            // success!
138        printk(KERN_INFO "EBBChar: Sent %d characters to the user\n",
    size_of_message);
139        return (size_of_message=0); // clear the position to the start and return 0
140     }
141     else {
```

```
142        printk(KERN_INFO "EBBChar: Failed to send %d characters to the user\n", error_count);
143        return -EFAULT;      // Failed -- return a bad address message (i.e. -14)
144    }
145 }
```

2.4.3.4 static int dev_release (struct inode * *inodep*, struct file * *filep*) [static]

The device release function that is called whenever the device is closed/released by the userspace program.

Parameters

inodep	A pointer to an inode object (defined in linux/fs.h)
filep	A pointer to a file object (defined in linux/fs.h)

```
168                                                         {
169    mutex_unlock(&ebbchar_mutex);                      // release the mutex (i.e., lock goes up)
170    printk(KERN_INFO "EBBChar: Device successfully closed\n");
171    return 0;
172 }
```

2.4.3.5 static ssize_t dev_write (struct file * *filep*, const char * *buffer*, size_t *len*, loff_t * *offset*) [static]

This function is called whenever the device is being written to from user space i.e. data is sent to the device from the user. The data is copied to the message[] array in this LKM using message[x] = buffer[x].

Parameters

filep	A pointer to a file object
buffer	The buffer to that contains the string to write to the device
len	The length of the array of data that is being passed in the const char buffer
offset	The offset if required

```
155                                                                                 {
156
157    sprintf(message, "%s(%d letters)", buffer, len);   // appending received string with its length
158    size_of_message = strlen(message);                 // store the length of the
       stored message
159    printk(KERN_INFO "EBBChar: Received %d characters from the user\n", len);
160    return len;
161 }
```

2.4.3.6 static void __exit ebbchar_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

```
99                                      {
100    mutex_destroy(&ebbchar_mutex);                     // destroy the dynamically-allocated mutex
101    device_destroy(ebbcharClass, MKDEV(majorNumber, 0)); // remove the device
102    class_unregister(ebbcharClass);                    // unregister the device class
103    class_destroy(ebbcharClass);                       // remove the device class
104    unregister_chrdev(majorNumber, DEVICE_NAME);       // unregister the major
       number
105    printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
106 }
```

2.4.3.7 static int __init ebbchar_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.

Returns

returns 0 if successful

```
62                                  {
63    printk(KERN_INFO "EBBChar: Initializing the EBBChar LKM\n");
64
65    // Try to dynamically allocate a major number for the device -- more difficult but worth it
66    majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
67    if (majorNumber<0){
68       printk(KERN_ALERT "EBBChar failed to register a major number\n");
69       return majorNumber;
70    }
71    printk(KERN_INFO "EBBChar: registered correctly with major number %d\n",
      majorNumber);
72
73    // Register the device class
74    ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);
75    if (IS_ERR(ebbcharClass)){           // Check for error and clean up if there is
76       unregister_chrdev(majorNumber, DEVICE_NAME);
77       printk(KERN_ALERT "Failed to register device class\n");
78       return PTR_ERR(ebbcharClass);     // Correct way to return an error on a pointer
79    }
80    printk(KERN_INFO "EBBChar: device class registered correctly\n");
81
82    // Register the device driver
83    ebbcharDevice = device_create(ebbcharClass, NULL, MKDEV(
      majorNumber, 0), NULL, DEVICE_NAME);
84    if (IS_ERR(ebbcharDevice)){          // Clean up if there is an error
85       class_destroy(ebbcharClass);      // Repeated code but the alternative is goto statements
86       unregister_chrdev(majorNumber, DEVICE_NAME);
87       printk(KERN_ALERT "Failed to create the device\n");
88       return PTR_ERR(ebbcharDevice);
89    }
90    printk(KERN_INFO "EBBChar: device class created correctly\n"); // Made it! device was initialized
91    mutex_init(&ebbchar_mutex);          // Initialize the mutex dynamically
92    return 0;
93 }
```

2.4.3.8 MODULE_AUTHOR ("Derek Molloy")

The author – visible when you use modinfo.

2.4.3.9 MODULE_DESCRIPTION ("A simple Linux char driver for the BBB")

The description – see modinfo.

2.4.3.10 module_exit (ebbchar_exit)

2.4.3.11 module_init (ebbchar_init)

A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)

2.4.3.12 MODULE_LICENSE ("GPL")

The license type – this affects available functionality.

2.4.3.13 MODULE_VERSION ("0.1")

A version number to inform users.

2.4.4 Variable Documentation

2.4.4.1 struct class* ebbcharClass = NULL [static]

The device-driver class struct pointer.

2.4.4.2 struct device* ebbcharDevice = NULL [static]

The device-driver device struct pointer.

2.4.4.3 struct file_operations fops [static]

Initial value:

```
=
{
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .release = dev_release,
}
```

Devices are represented as file structure in the kernel. The file_operations structure from /linux/fs.h lists the callback functions that you wish to associated with your file operations using a C99 syntax structure. char devices usually implement open, read, write and release calls

2.4.4.4 int majorNumber [static]

Store the device number – determined automatically.

2.4.4.5 char message[256] = {0} [static]

Memory for the string that is passed from userspace.

2.4.4.6 int numberOpens = 0 [static]

Counts the number of times the device is opened.

2.4.4.7 short size_of_message [static]

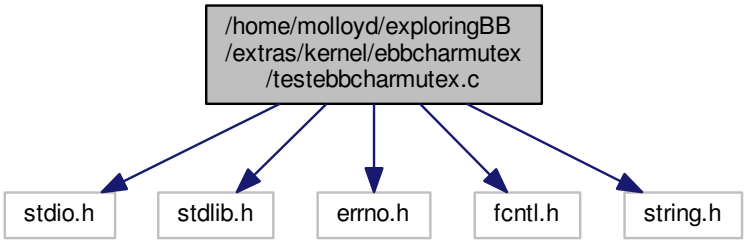
Used to remember the size of the string stored.

2.5 /home/molloyd/exploringBB/extras/kernel/ebbcharmutex/testebbcharmutex.c File Reference

A Linux user space program that communicates with the ebbchar.c LKM. It passes a string to the LKM and reads the response from the LKM. For this example to work the device must be called /dev/ebbchar.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
```

Include dependency graph for testebbcharmutex.c:



Macros

- #define BUFFER_LENGTH 256

  The buffer length (crude but fine)

Functions

- int main ()

Variables

- static char receive [BUFFER_LENGTH]

  The receive buffer from the LKM.

2.5.1 Detailed Description

A Linux user space program that communicates with the ebbchar.c LKM. It passes a string to the LKM and reads the response from the LKM. For this example to work the device must be called /dev/ebbchar.

Author

Derek Molloy

Date

7 April 2015

Version

0.1

See also

http://www.derekmolloy.ie/ for a full description and follow-up descriptions.

2.5.2 Macro Definition Documentation

2.5.2.1 #define BUFFER_LENGTH 256

The buffer length (crude but fine)

2.5.3 Function Documentation

2.5.3.1 int main ( )

```
20          {
21    int ret, fd;
22    char stringToSend[BUFFER_LENGTH];
23    printf("Starting device test code example...\n");
24    fd = open("/dev/ebbchar", O_RDWR);             // Open the device with read/write access
25    if (fd < 0){
26       perror("Failed to open the device...");
27       return errno;
28    }
29    printf("Type in a short string to send to the kernel module:\n");
30    scanf("%[^\n]%*c", stringToSend);              // Read in a string (with spaces)
31    printf("Writing message to the device [%s].\n", stringToSend);
32    ret = write(fd, stringToSend, strlen(stringToSend)); // Send the string to the LKM
33    if (ret < 0){
34       perror("Failed to write the message to the device.");
35       return errno;
36    }
37
38    printf("Press ENTER to read back from the device...");
39    getchar();
40
41    printf("Reading from the device...\n");
42    ret = read(fd, receive, BUFFER_LENGTH);        // Read the response from the LKM
43    if (ret < 0){
44       perror("Failed to read the message from the device.");
45       return errno;
46    }
47    printf("The received message is: [%s]\n", receive);
48    printf("End of the program\n");
49    return 0;
50 }
```

2.5.4 Variable Documentation

2.5.4.1 char receive[BUFFER_LENGTH] [static]

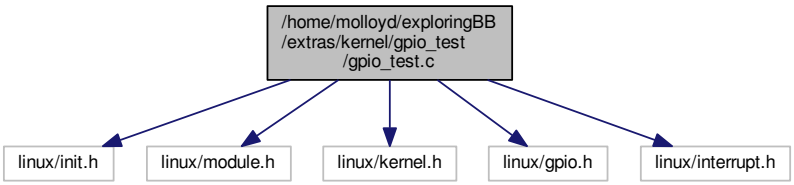
The receive buffer from the LKM.

2.6 /home/molloyd/exploringBB/extras/kernel/gpio_test/gpio_test.c File Reference

A kernel module for controlling a GPIO LED/button pair. The device mounts devices via sysfs /sys/class/gpio/gpio115 and gpio49. Therefore, this test LKM circuit assumes that an LED is attached to GPIO 49 which is on P9_23 and the button is attached to GPIO 115 on P9_27. There is no requirement for a custom overlay, as the pins are in their default mux mode states.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
```

Include dependency graph for gpio_test.c:



Functions

- MODULE_LICENSE ("GPL")
- MODULE_AUTHOR ("Derek Molloy")
- MODULE_DESCRIPTION ("A Button/LED test driver for the BBB")
- MODULE_VERSION ("0.1")
- static irq_handler_t ebbgpio_irq_handler (unsigned int irq, void *dev_id, struct pt_regs *regs)

  Function prototype for the custom IRQ handler function – see below for the implementation.
- static int __init ebbgpio_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.
- static void __exit ebbgpio_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required. Used to release the GPIOs and display cleanup messages.
- module_init (ebbgpio_init)
- module_exit (ebbgpio_exit)

Variables

- static unsigned int gpioLED = 49

  hard coding the LED gpio for this example to P9_23 (GPIO49)
- static unsigned int gpioButton = 115

  hard coding the button gpio for this example to P9_27 (GPIO115)
- static unsigned int irqNumber

  Used to share the IRQ number within this file.
- static unsigned int numberPresses = 0

  For information, store the number of button presses.
- static bool ledOn = 0

  Is the LED on or off? Used to invert its state (off by default)

2.6.1 Detailed Description

A kernel module for controlling a GPIO LED/button pair. The device mounts devices via sysfs /sys/class/gpio/gpio115 and gpio49. Therefore, this test LKM circuit assumes that an LED is attached to GPIO 49 which is on P9_23 and the button is attached to GPIO 115 on P9_27. There is no requirement for a custom overlay, as the pins are in their default mux mode states.

Author

Derek Molloy

Date

19 April 2015

See also

http://www.derekmolloy.ie/

2.6.2 Function Documentation

2.6.2.1 static void __exit ebbgpio_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required. Used to release the GPIOs and display cleanup messages.

```
82                                      {
83    printk(KERN_INFO "GPIO_TEST: The button state is currently: %d\n", gpio_get_value(
      gpioButton));
84    printk(KERN_INFO "GPIO_TEST: The button was pressed %d times\n",
      numberPresses);
85    gpio_set_value(gpioLED, 0);              // Turn the LED off, makes it clear the device was
       unloaded
86    gpio_unexport(gpioLED);                  // Unexport the LED GPIO
87    free_irq(irqNumber, NULL);               // Free the IRQ number, no *dev_id required in this
       case
88    gpio_unexport(gpioButton);               // Unexport the Button GPIO
89    gpio_free(gpioLED);                      // Free the LED GPIO
90    gpio_free(gpioButton);                   // Free the Button GPIO
91    printk(KERN_INFO "GPIO_TEST: Goodbye from the LKM!\n");
92 }
```

2.6.2.2 static int __init ebbgpio_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.

Returns

returns 0 if successful

```
39                                      {
40    int result = 0;
41    printk(KERN_INFO "GPIO_TEST: Initializing the GPIO_TEST LKM\n");
42    // Is the GPIO a valid GPIO number (e.g., the BBB has 4x32 but not all available)
43    if (!gpio_is_valid(gpioLED)){
44       printk(KERN_INFO "GPIO_TEST: invalid LED GPIO\n");
45       return -ENODEV;
46    }
47    // Going to set up the LED. It is a GPIO in output mode and will be on by default
48    ledOn = true;
49    gpio_request(gpioLED, "sysfs");          // gpioLED is hardcoded to 49, request it
50    gpio_direction_output(gpioLED, ledOn);   // Set the gpio to be in output mode and on
51 // gpio_set_value(gpioLED, ledOn);          // Not required as set by line above (here for reference)
52    gpio_export(gpioLED, false);             // Causes gpio49 to appear in /sys/class/gpio
53                                             // the bool argument prevents the direction from being changed
54    gpio_request(gpioButton, "sysfs");       // Set up the gpioButton
55    gpio_direction_input(gpioButton);        // Set the button GPIO to be an input
56    gpio_set_debounce(gpioButton, 200);      // Debounce the button with a delay of 200ms
57    gpio_export(gpioButton, false);          // Causes gpio115 to appear in /sys/class/gpio
58                                             // the bool argument prevents the direction from being changed
59    // Perform a quick test to see that the button is working as expected on LKM load
60    printk(KERN_INFO "GPIO_TEST: The button state is currently: %d\n", gpio_get_value(
      gpioButton));
```

```
61
62     // GPIO numbers and IRQ numbers are not the same! This function performs the mapping for us
63     irqNumber = gpio_to_irq(gpioButton);
64     printk(KERN_INFO "GPIO_TEST: The button is mapped to IRQ: %d\n", irqNumber);
65
66     // This next call requests an interrupt line
67     result = request_irq(irqNumber,             // The interrupt number requested
68                          (irq_handler_t) ebbgpio_irq_handler, // The pointer to the
       handler function below
69                          IRQF_TRIGGER_RISING,   // Interrupt on rising edge (button press, not release)
70                          "ebb_gpio_handler",    // Used in /proc/interrupts to identify the owner
71                          NULL);                 // The *dev_id for shared interrupt lines, NULL is okay
72
73     printk(KERN_INFO "GPIO_TEST: The interrupt request result is: %d\n", result);
74     return result;
75 }
```

2.6.2.3 static irq_handler_t ebbgpio_irq_handler (unsigned int *irq*, void * *dev_id*, struct pt_regs * *regs*) [static]

Function prototype for the custom IRQ handler function – see below for the implementation.

The GPIO IRQ Handler function This function is a custom interrupt handler that is attached to the GPIO above. The same interrupt handler cannot be invoked concurrently as the interrupt line is masked out until the function is complete. This function is static as it should not be invoked directly from outside of this file.

Parameters

irq	the IRQ number that is associated with the GPIO – useful for logging.
dev_id	the *dev_id that is provided – can be used to identify which device caused the interrupt Not used in this example as NULL is passed.
regs	h/w specific register values – only really ever used for debugging. return returns IRQ_HANDLED if successful – should return IRQ_NONE otherwise.

```
104                                                                                                 {
105     ledOn = !ledOn;                          // Invert the LED state on each button press
106     gpio_set_value(gpioLED, ledOn);          // Set the physical LED accordingly
107     printk(KERN_INFO "GPIO_TEST: Interrupt! (button state is %d)\n", gpio_get_value(
        gpioButton));
108     numberPresses++;                         // Global counter, will be outputted when the
        module is unloaded
109     return (irq_handler_t) IRQ_HANDLED;      // Announce that the IRQ has been handled correctly
110 }
```

2.6.2.4 MODULE_AUTHOR ("Derek Molloy")

2.6.2.5 MODULE_DESCRIPTION ("A Button/LED test driver for the BBB")

2.6.2.6 module_exit (ebbgpio_exit)

2.6.2.7 module_init (ebbgpio_init)

This next calls are mandatory – they identify the initialization function and the cleanup function (as above).

2.6.2.8 MODULE_LICENSE ("GPL")

2.6.2.9 MODULE_VERSION ("0.1")

2.6.3 Variable Documentation

2.6.3.1 unsigned int gpioButton = 115 [static]

hard coding the button gpio for this example to P9_27 (GPIO115)

2.6.3.2 unsigned int gpioLED = 49 [static]

hard coding the LED gpio for this example to P9_23 (GPIO49)

2.6.3.3 unsigned int irqNumber [static]

Used to share the IRQ number within this file.

2.6.3.4 bool ledOn = 0 [static]

Is the LED on or off? Used to invert its state (off by default)

2.6.3.5 unsigned int numberPresses = 0 [static]

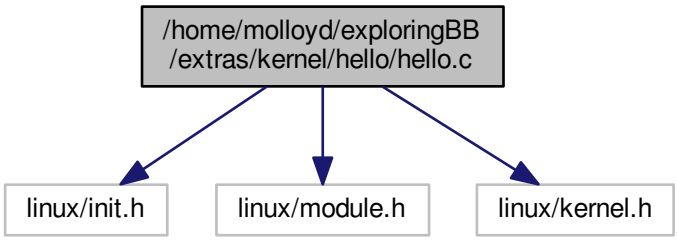
For information, store the number of button presses.

2.7 /home/molloyd/exploringBB/extras/kernel/hello/hello.c File Reference

An introductory "Hello World!" loadable kernel module (LKM) that can display a message in the /var/log/kern.log file when the module is loaded and removed. The module can accept an argument when it is loaded – the name, which appears in the kernel log files.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
```

Include dependency graph for hello.c:



Functions

- MODULE_LICENSE ("GPL")

  The license type – this affects runtime behavior.
- MODULE_AUTHOR ("Derek Molloy")

  The author – visible when you use modinfo.
- MODULE_DESCRIPTION ("A simple Linux driver for the BBB.")

  The description – see modinfo.
- MODULE_VERSION ("0.1")

  The version of the module.

- module_param (name, charp, S_IRUGO)

  Param desc. charp = char ptr, S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (name,"The name to display in /var/log/kern.log")

  parameter description
- static int __init helloBBB_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.
- static void __exit helloBBB_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.
- module_init (helloBBB_init)

  A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)
- module_exit (helloBBB_exit)

Variables

- static char * name = "world"

  An example LKM argument – default value is "world".

2.7.1 Detailed Description

An introductory "Hello World!" loadable kernel module (LKM) that can display a message in the /var/log/kern.log file when the module is loaded and removed. The module can accept an argument when it is loaded – the name, which appears in the kernel log files.

Author

Derek Molloy

Date

4 April 2015

Version

0.1

See also

http://www.derekmolloy.ie/ for a full description and follow-up descriptions.

2.7.2 Function Documentation

2.7.2.1 static void __exit helloBBB_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

```
40                                      {
41    printk(KERN_INFO "EBB: Goodbye %s from the BBB LKM!\n", name);
42 }
```

2.7.2.2 static int __init helloBBB_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point.

Returns

returns 0 if successful

```
31                                      {
32    printk(KERN_INFO "EBB: Hello %s from the BBB LKM!\n", name);
33    return 0;
34 }
```

2.7.2.3 MODULE_AUTHOR ("Derek Molloy")

The author – visible when you use modinfo.

2.7.2.4 MODULE_DESCRIPTION ("A simple Linux driver for the BBB.")

The description – see modinfo.

2.7.2.5 module_exit (helloBBB_exit)

2.7.2.6 module_init (helloBBB_init)

A module must use the module_init() module_exit() macros from linux/init.h, which identify the initialization function at insertion time and the cleanup function (as listed above)

2.7.2.7 MODULE_LICENSE ("GPL")

The license type – this affects runtime behavior.

2.7.2.8 module_param (name , charp , S_IRUGO)

Param desc. charp = char ptr, S_IRUGO can be read/not changed.

2.7.2.9 MODULE_PARM_DESC (name , "The name to display in /var/log/kern.log")

parameter description

2.7.2.10 MODULE_VERSION ("0.1")

The version of the module.

2.7.3 Variable Documentation

2.7.3.1 char* name = "world" [static]

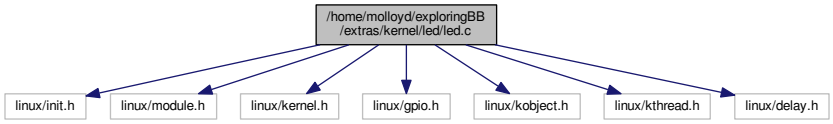
An example LKM argument – default value is "world".

2.8 /home/molloyd/exploringBB/extras/kernel/led/led.c File Reference

A kernel module for controlling a simple LED (or any signal) that is connected to a GPIO. It is threaded in order that it can flash the LED. The sysfs entry appears at /sys/ebb/led49.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/kobject.h>
#include <linux/kthread.h>
#include <linux/delay.h>
```

Include dependency graph for led.c:



Enumerations

- enum modes { OFF, ON, FLASH }

Functions

- MODULE_LICENSE ("GPL")
- MODULE_AUTHOR ("Derek Molloy")
- MODULE_DESCRIPTION ("A simple Linux LED driver LKM for the BBB")
- MODULE_VERSION ("0.1")
- module_param (gpioLED, uint, S_IRUGO)

  Param desc. S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (gpioLED," GPIO LED number (default=49)")

  parameter description
- module_param (blinkPeriod, uint, S_IRUGO)

  Param desc. S_IRUGO can be read/not changed.
- MODULE_PARM_DESC (blinkPeriod," LED blink period in ms (min=1, default=1000, max=10000)")
- static ssize_t mode_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  A callback function to display the LED mode.
- static ssize_t mode_store (struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)

  A callback function to store the LED mode using the enum above.
- static ssize_t period_show (struct kobject *kobj, struct kobj_attribute *attr, char *buf)

  A callback function to display the LED period.
- static ssize_t period_store (struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)

  A callback function to store the LED period value.
- static int flash (void *arg)

  The pointer to the thread task.
- static int __init ebbLED_init (void)

  The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.

- static void __exit ebbLED_exit (void)

  The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.
- module_init (ebbLED_init)
- module_exit (ebbLED_exit)

Variables

- static unsigned int gpioLED = 49

  Default GPIO for the LED is 49.
- static unsigned int blinkPeriod = 1000

  The blink period in ms.
- static char ledName [7] = "ledXXX"

  Null terminated default string – just in case.
- static bool ledOn = 0

  Is the LED on or off? Used for flashing.
- static enum modes mode = FLASH

  Default mode is flashing.
- static struct kobj_attribute period_attr = __ATTR(blinkPeriod, 0666, period_show, period_store)
- static struct kobj_attribute mode_attr = __ATTR(mode, 0666, mode_show, mode_store)
- static struct attribute * ebb_attrs []
- static struct attribute_group attr_group
- static struct kobject * ebb_kobj
- static struct task_struct * task

  The pointer to the kobject.

2.8.1 Detailed Description

A kernel module for controlling a simple LED (or any signal) that is connected to a GPIO. It is threaded in order that it can flash the LED. The sysfs entry appears at /sys/ebb/led49.

Author

Derek Molloy

Date

19 April 2015

See also

http://www.derekmolloy.ie/

2.8.2 Enumeration Type Documentation

2.8.2.1 enum modes

Enumerator

OFF

ON

FLASH

```
34 { OFF, ON, FLASH };
```

2.8.3 Function Documentation

2.8.3.1 static void __exit ebbLED_exit (void) [static]

The LKM cleanup function Similar to the initialization function, it is static. The __exit macro notifies that if this code is used for a built-in driver (not a LKM) that this function is not required.

```
168                                     {
169    kthread_stop(task);                      // Stop the LED flashing thread
170    kobject_put(ebb_kobj);                   // clean up -- remove the kobject sysfs entry
171    gpio_set_value(gpioLED, 0);              // Turn the LED off, indicates device was unloaded
172    gpio_unexport(gpioLED);                  // Unexport the Button GPIO
173    gpio_free(gpioLED);                      // Free the LED GPIO
174    printk(KERN_INFO "EBB LED: Goodbye from the EBB LED LKM!\n");
175 }
```

2.8.3.2 static int __init ebbLED_init (void) [static]

The LKM initialization function The static keyword restricts the visibility of the function to within this C file. The __init macro means that for a built-in driver (not a LKM) the function is only used at initialization time and that it can be discarded and its memory freed up after that point. In this example this function sets up the GPIOs and the IRQ.

Returns

returns 0 if successful

```
132                                     {
133    int result = 0;
134
135    printk(KERN_INFO "EBB LED: Initializing the EBB LED LKM\n");
136    sprintf(ledName, "led%d", gpioLED);      // Create the gpio115 name for /sys/ebb/led49
137
138    ebb_kobj = kobject_create_and_add("ebb", kernel_kobj->parent); // kernel_kobj points to
       /sys/kernel
139    if(!ebb_kobj){
140       printk(KERN_ALERT "EBB LED: failed to create kobject\n");
141       return -ENOMEM;
142    }
143    // add the attributes to /sys/ebb/ -- for example, /sys/ebb/led49/ledOn
144    result = sysfs_create_group(ebb_kobj, &attr_group);
145    if(result) {
146       printk(KERN_ALERT "EBB LED: failed to create sysfs group\n");
147       kobject_put(ebb_kobj);                // clean up -- remove the kobject sysfs entry
148       return result;
149    }
150    ledOn = true;
151    gpio_request(gpioLED, "sysfs");          // gpioLED is 49 by default, request it
152    gpio_direction_output(gpioLED, ledOn);   // Set the gpio to be in output mode and turn on
153    gpio_export(gpioLED, false);  // causes gpio49 to appear in /sys/class/gpio
154                                  // the second argument prevents the direction from being changed
155
156    task = kthread_run(flash, NULL, "LED_flash_thread");  // Start the LED flashing thread
157    if(IS_ERR(task)){                                     // Kthread name is LED_flash_thread
158       printk(KERN_ALERT "EBB LED: failed to create the task\n");
159       return PTR_ERR(task);
160    }
161    return result;
162 }
```

2.8.3.3 static int flash (void * *arg*) [static]

The pointer to the thread task.

The LED Flasher main kthread loop

Parameters

arg	A void pointer used in order to pass data to the thread

Returns

returns 0 if successful

```
110                             {
111    printk(KERN_INFO "EBB LED: Thread has started running \n");
112    while(!kthread_should_stop()){           // Returns true when kthread_stop() is called
113       set_current_state(TASK_RUNNING);
114       if (mode==FLASH) ledOn = !ledOn;      // Invert the LED state
115       else if (mode==ON) ledOn = true;
116       else ledOn = false;
117       gpio_set_value(gpioLED, ledOn);       // Use the LED state to light/turn off the LED
118       set_current_state(TASK_INTERRUPTIBLE);
119       msleep(blinkPeriod/2);                // millisecond sleep for half of the period
120    }
121    printk(KERN_INFO "EBB LED: Thread has run to completion \n");
122    return 0;
123 }
```

2.8.3.4 static ssize_t mode_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

A callback function to display the LED mode.

Parameters

kobj	represents a kernel object device that appears in the sysfs filesystem
attr	the pointer to the kobj_attribute struct
buf	the buffer to which to write the number of presses

Returns

return the number of characters of the mode string successfully displayed

```
43                                                                                  {
44    switch(mode){
45       case OFF:   return sprintf(buf, "off\n");       // Display the state -- simplistic approach
46       case ON:    return sprintf(buf, "on\n");
47       case FLASH: return sprintf(buf, "flash\n");
48       default:    return sprintf(buf, "LKM Error\n"); // Cannot get here
49    }
50 }
```

2.8.3.5 static ssize_t mode_store (struct kobject * *kobj*, struct kobj_attribute * *attr*, const char * *buf*, size_t *count*) [static]

A callback function to store the LED mode using the enum above.

```
53
      {
54    // the count-1 is important as otherwise the \n is used in the comparison
55    if (strncmp(buf,"on",count-1)==0) { mode = ON; }   // strncmp() compare with fixed number chars
56    else if (strncmp(buf,"off",count-1)==0) { mode = OFF; }
57    else if (strncmp(buf,"flash",count-1)==0) { mode = FLASH; }
58    return count;
59 }
```

2.8.3.6 MODULE_AUTHOR ("Derek Molloy")

2.8.3.7 MODULE_DESCRIPTION ("A simple Linux LED driver LKM for the BBB")

2.8.3.8 module_exit (ebbLED_exit)

2.8.3.9 module_init (ebbLED_init)

This next calls are mandatory – they identify the initialization function and the cleanup function (as above).

2.8.3.10 MODULE_LICENSE ("GPL")

2.8.3.11 module_param (gpioLED , uint , S_IRUGO)

Param desc. S_IRUGO can be read/not changed.

2.8.3.12 module_param (blinkPeriod , uint , S_IRUGO)

Param desc. S_IRUGO can be read/not changed.

2.8.3.13 MODULE_PARM_DESC (gpioLED , " GPIO LED number (default=49)")

parameter description

2.8.3.14 MODULE_PARM_DESC (blinkPeriod , " LED blink period in ms (min=1, default=1000, max=10000)")

2.8.3.15 MODULE_VERSION ("0.1")

2.8.3.16 static ssize_t period_show (struct kobject * *kobj*, struct kobj_attribute * *attr*, char * *buf*) [static]

A callback function to display the LED period.

```
62                                                                                  {
63     return sprintf(buf, "%d\n", blinkPeriod);
64 }
```

2.8.3.17 static ssize_t period_store (struct kobject * *kobj*, struct kobj_attribute * *attr*, const char * *buf*, size_t *count*) [static]

A callback function to store the LED period value.

```
67
        {
68     unsigned int period;                    // Using a variable to validate the data sent
69     sscanf(buf, "%du", &period);            // Read in the period as an unsigned int
70     if ((period>1)&&(period<=10000)){       // Must be 2ms or greater, 10secs or less
71        blinkPeriod = period;                // Within range, assign to blinkPeriod variable
72     }
73     return period;
74 }
```

2.8.4 Variable Documentation

2.8.4.1 struct attribute_group attr_group [static]

Initial value:

```
= {
   .name  = ledName,
   .attrs = ebb_attrs,
}
```

The attribute group uses the attribute array and a name, which is exposed on sysfs – in this case it is gpio49, which is automatically defined in the ebbLED_init() function below using the custom kernel parameter that can be passed when the module is loaded.

2.8.4.2 unsigned int blinkPeriod = 1000 [static]

The blink period in ms.

2.8.4.3 struct attribute* ebb_attrs[] [static]

Initial value:

```
= {
   &period_attr.attr,
   &mode_attr.attr,
   NULL,
}
```

The ebb_attrs[] is an array of attributes that is used to create the attribute group below. The attr property of the kobj_attribute is used to extract the attribute struct

2.8.4.4 struct kobject* ebb_kobj [static]

2.8.4.5 unsigned int gpioLED = 49 [static]

Default GPIO for the LED is 49.

2.8.4.6 char ledName[7] = "ledXXX" [static]

Null terminated default string – just in case.

2.8.4.7 bool ledOn = 0 [static]

Is the LED on or off? Used for flashing.

2.8.4.8 enum modes mode = FLASH [static]

Default mode is flashing.

2.8.4.9 struct kobj_attribute mode_attr = __ATTR(mode, 0666, mode_show, mode_store) [static]

2.8.4.10 struct kobj_attribute period_attr = __ATTR(blinkPeriod, 0666, period_show, period_store) [static]

Use these helper macros to define the name and access levels of the kobj_attributes The kobj_attribute has an attribute attr (name and mode), show and store function pointers The period variable is associated with the blink↩ Period variable and it is to be exposed with mode 0666 using the period_show and period_store functions above

2.8.4.11 struct task_struct* task [static]

The pointer to the kobject.

Index