Contents

1 File Index .............................. 1
  1.1 File List ...................................... 1

2 File Documentation ..................... 3
  2.1 /home/molloyd/exploringBB/extras/kernel/button/button.c File Reference ........................................ 3
    2.1.1 Detailed Description ............................ 5
    2.1.2 Macro Definition Documentation ..................... 5
      2.1.2.1 DEBOUNCE_TIME .............................. 5
    2.1.3 Function Documentation ............................ 5
      2.1.3.1 diffTime_show ................................ 5
      2.1.3.2 ebbButton_exit ................................ 6
      2.1.3.3 ebbButton_init ................................ 6
      2.1.3.4 ebbgpio_irq_handler ............................ 7
      2.1.3.5 isDebounce_show ................................ 7
      2.1.3.6 isDebounce_store ................................ 7
      2.1.3.7 lastTime_show ................................ 8
      2.1.3.8 ledOn_show ..................................... 8
      2.1.3.9 MODULE_AUTHOR ................................. 8
      2.1.3.10 MODULE_DESCRIPTION ............................ 8
      2.1.3.11 module_exit .................................... 8
      2.1.3.12 module_init .................................... 8
      2.1.3.13 MODULE_LICENSE ............................... 8
      2.1.3.14 module_param .................................. 8
      2.1.3.15 module_param .................................. 8
      2.1.3.16 module_param .................................. 8
      2.1.3.17 MODULE_PARM_DESC ............................. 8
      2.1.3.18 MODULE_PARM_DESC ............................. 8
      2.1.3.19 MODULE_PARM_DESC ............................. 9
      2.1.3.20 MODULE_VERSION ............................... 9
      2.1.3.21 numberPresses_show ............................ 9
      2.1.3.22 numberPresses_store ............................ 9
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4.4 majorNumber</td>
<td>16</td>
</tr>
<tr>
<td>2.2.4.5 message</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4.6 numberOpens</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4.7 size_of_message</td>
<td>17</td>
</tr>
<tr>
<td>2.3 /home/molloyd/exploringBB/extras/kernel/ebbchar/testebbchar.c</td>
<td></td>
</tr>
<tr>
<td>File Reference</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Detailed Description</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2 Macro Definition Documentation</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2.1 BUFFER_LENGTH</td>
<td>18</td>
</tr>
<tr>
<td>2.3.3 Function Documentation</td>
<td>18</td>
</tr>
<tr>
<td>2.3.3.1 main</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4 Variable Documentation</td>
<td>18</td>
</tr>
<tr>
<td>2.3.4.1 receive</td>
<td>18</td>
</tr>
<tr>
<td>2.4 /home/molloyd/exploringBB/extras/kernel/ebbcharmutex/ebbcharmutex.c</td>
<td></td>
</tr>
<tr>
<td>File Reference</td>
<td>19</td>
</tr>
<tr>
<td>2.4.1 Detailed Description</td>
<td>20</td>
</tr>
<tr>
<td>2.4.2 Macro Definition Documentation</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2.1 CLASS_NAME</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2.2 DEVICE_NAME</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3 Function Documentation</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3.1 DEFINE_MUTEX</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3.2 dev_open</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3.3 dev_read</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3.4 dev_release</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3.5 dev_write</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3.6 ebbchar_exit</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3.7 ebbchar_init</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3.8 MODULE_AUTHOR</td>
<td>23</td>
</tr>
<tr>
<td>2.4.3.9 MODULE_DESCRIPTION</td>
<td>23</td>
</tr>
<tr>
<td>2.4.3.10 module_exit</td>
<td>23</td>
</tr>
<tr>
<td>2.4.3.11 module_init</td>
<td>23</td>
</tr>
<tr>
<td>2.4.3.12 MODULE_LICENSE</td>
<td>23</td>
</tr>
<tr>
<td>2.4.3.13 MODULE_VERSION</td>
<td>23</td>
</tr>
<tr>
<td>2.4.4 Variable Documentation</td>
<td>23</td>
</tr>
<tr>
<td>2.4.4.1 ebbcharClass</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.2 ebbcharDevice</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.3 fops</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.4 majorNumber</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.5 message</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.6 numberOpens</td>
<td>24</td>
</tr>
<tr>
<td>2.4.4.7 size_of_message</td>
<td>24</td>
</tr>
<tr>
<td>2.5 /home/molloyd/exploringBB/extras/kernel/ebbcharmutex/testebbcharmutex.c</td>
<td></td>
</tr>
<tr>
<td>File Reference</td>
<td>24</td>
</tr>
</tbody>
</table>
Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

/home/molloyd/exploringBB/extras/kernel/button/button.c
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 3

/home/molloyd/exploringBB/extras/kernel/ebbchar/ebbchar.c
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 . . . . . . . . 11

/home/molloyd/exploringBB/extras/kernel/ebbchar/testebbchar.c
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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

/home/molloyd/exploringBB/extras/kernel/ebbcharmutex/ebbcharmutex.c
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 . . . . . . . . . . . . . . . . . . . . . 19

/home/molloyd/exploringBB/extras/kernel/ebbcharmutex/testebbcharmutex.c
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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

/home/molloyd/exploringBB/extras/kernel/gpio_test/gpio_test.c
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 . 26

/home/molloyd/exploringBB/extras/kernel/hello/hello.c
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 . . . . . . . . 30

/home/molloyd/exploringBB/extras/kernel/led/led.c
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 . . . . . 33
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.

```c
#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:

![Include dependency graph](image)

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 )

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

```c
{ 
  return sprintf(buf, "%lu.%lu%n", ts_diff.tv_sec, ts_diff.tv_nsec); 
}
```
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.

```c
212 {  
213 printk(KERN_INFO "EBB Button: The button was pressed %d times\n",  
214 numberPresses);  
215 kobject_put(ebb_kobj); // clean up -- remove the kobject sysfs entry  
216 gpio_set_value(gpioLED, 0); // Turn the LED off, makes it clear the device was  
217 // unloaded  
218 gpio_unexport(gpioLED); // Unexport the LED GPIO  
219 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

```c
153 {  
154 int result = 0;  
155 unsigned long IRQflags = IRQF_TRIGGER_RISING; // The default is a rising-edge interrupt  
156 printk(KERN_INFO "EBB Button: Initializing the EBB Button LKM\n");  
157 sprintf(gpioName, "gpio%d", gpioButton); // Create the gpio115 name for  
/sys/ebb/gpio115  
159 // create the kobject sysfs entry at /sys/ebb -- probably not an ideal location!  
160 ebb_kobj = kobject_create_and_add("ebb", kernel_kobj->parent); // kernel_kobj points to  
/sys/kernel  
161 if(!ebb_kobj){  
162 printk(KERN_ALERT "EBB Button: failed to create kobject mapping\n");  
163 return -ENOMEM;  
164 }  
165 // add the attributes to /sys/ebb -- for example, /sys/ebb/gpio115/numberPresses  
166 result = sysfs_create_group(ebb_kobj, &attr_group);  
167 if(result) {  
168 printk(KERN_ALERT "EBB Button: failed to create sysfs group\n");  
169 return result;  
170 }  
171 getnstimeofday(&ts_last); // set the last time to be the current time  
172 ts_diff = timespec_sub(ts_last, ts_last); // set the initial time difference to be 0  
```

// Going to set up the LED. It is a GPIO in output mode and will be on by default

```c
173 ledOn = true;  
174 gpio_request(gpioLED, "sysfs"); // gpioLED is hardcoded to 49, request it  
175 gpio_direction_output(gpioLED, ledOn); // Set the gpio to be in output mode and on  
176 gpio_set_value(gpioLED, ledOn); // Not required as set by line above (here for reference)  
177 gpio_export(gpioLED, false); // Causes gpio49 to appear in /sys/class/gpio  
178 // the bool argument prevents the direction from being changed  
179 gpio_request(gpioButton, "sysfs");  
180 gpio_direction_input(gpioButton); // Set the button GPIO to be an input  
181 gpio_set_debounce(gpioButton, DEBOUNCE_TIME); // Debounce the button with a delay  
of 100ms  
182 gpio_export(gpioButton, false); // Causes gpio115 to appear in /sys/class/gpio  
183 // the bool argument prevents the direction from being changed  
184 // Perform a quick test to see that the button is working as expected on LKM load  
185 printk(KERN_INFO "EBB Button: The button state is currently: %d\n", gpio_get_value(  
186 gpioButton));  
187 irqNumber = gpio_to_irq(gpioButton);  
```
printk(KERN_INFO "EBB Button: The button is mapped to IRQ: %d\n", irqNumber);

if(!isRising) { // If the kernel parameter isRising=0 is supplied
    IRQflags = IRQF_TRIGGER_FALLING; // Set the interrupt to be on the falling edge
}

// This next call requests an interrupt line
result = request_irq(irqNumber, // The interrupt number requested
    (irq_handler_t) ebbgpio_irq_handler, // The pointer to the handler function below
    IRQflags, // Use the custom kernel param to set interrupt type
    "ebb_button_handler", // Used in /proc/interrupts to identify the owner
    NULL); // The *dev_id for shared interrupt lines, NULL is okay
return result;
}

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

<table>
<thead>
<tr>
<th>irq</th>
<th>the IRQ number that is associated with the GPIO – useful for logging.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev_id</td>
<td>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.</td>
</tr>
<tr>
<td>regs</td>
<td>h/w specific register values – only really ever used for debugging. return returns IRQ_HANDLED if successful – should return IRQ_NONE otherwise.</td>
</tr>
</tbody>
</table>

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.

return sprintf(buf, "%d\n", isDebounce);

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.

unsigned int temp;
sscanf(buf, "%d", &temp); // use a temp variable for correct int->bool
gpio_set_debounce(gpioButton, 0);
if(isDebounce) { gpio_set_debounce(gpioButton, DEBOUNCE_TIME);
```c
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 )

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

```
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

\begin{verbatim}
  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
\end{verbatim}

Returns

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

\begin{verbatim}
54  return sprintf(buf, "%d\n", numberPresses);
55 }  
\end{verbatim}

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

\begin{verbatim}
  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
\end{verbatim}

Returns

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

\begin{verbatim}
66  {  
67   sscanf(buf, "%d", &numberPresses);
68   return count;
69 }  
\end{verbatim}

2.1.4  Variable Documentation

2.1.4.1  struct attribute_group attr_group [static]

Initial value:

\begin{verbatim}
  = {  
     .name = gpioName,
     .attrs = ebb_attr,
  }
\end{verbatim}

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 numberPresses 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/release 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 it 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)

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filep</td>
<td>A pointer to a file object (defined in linux/fs.h)</td>
</tr>
<tr>
<td>buffer</td>
<td>The pointer to the buffer to which this function writes the data</td>
</tr>
<tr>
<td>len</td>
<td>The length of the buffer</td>
</tr>
<tr>
<td>offset</td>
<td>The offset if required</td>
</tr>
</tbody>
</table>

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

### 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inodep</td>
<td>A pointer to an inode object (defined in linux/fs.h)</td>
</tr>
<tr>
<td>filep</td>
<td>A pointer to a file object (defined in linux/fs.h)</td>
</tr>
</tbody>
</table>

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

### 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filep</td>
<td>A pointer to a file object (defined in linux/fs.h)</td>
</tr>
<tr>
<td>buffer</td>
<td>The pointer to the buffer to which this function writes the data</td>
</tr>
<tr>
<td>len</td>
<td>The length of the buffer</td>
</tr>
<tr>
<td>offset</td>
<td>The offset if required</td>
</tr>
</tbody>
</table>

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

Generated on Sat Apr 25 2015 15:26:35 for Exploring BeagleBone: LKMs (by Derek Molloy) by Doxygen
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:

```c
    = [ 
        .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:

```plaintext
/include <stdio.h>  
/include <stdlib.h>  
/include <errno.h>  
/include <fcntl.h>  
/include <string.h>
```

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.\n");  
35 return errno;  
36 }  
37 printf("Press ENTER to read back from the device...\n");  
38 getchar();  
39 printf("Reading from the device...\n");  
40 ret = read(fd, receive, BUFFER_LENGTH); // Read the response from the LKM  
41 if (ret < 0){  
42 perror("Failed to read the message from the device.\n");  
43 return errno;  
44 }  
45 printf("The received message is: [\%s]\n", receive);  
46 printf("End of the program\n");  
47 return 0;  
48 }

2.3.4 Variable Documentation

2.3.4.1 char receive[BUFFER_LENGTH] [static]

The receive buffer from the LKM.
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.

```c
#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:

```
/home/molloyd/exploringBB/extras/kernel/ebbcharmutex/ebbcharmutex.c
linux/init.h
linux/module.h
linux/device.h
linux/kernel.h
linux/fs.h
asm/uaccess.h
linux/mutex.h
```

### 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 *ebbccharClass = NULL

  The device-driver class struct pointer.

- static struct device *ebbccharDevice = 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

<table>
<thead>
<tr>
<th>inodep</th>
<th>A pointer to an inode object (defined in linux/fs.h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>filep</td>
<td>A pointer to a file object (defined in linux/fs.h)</td>
</tr>
</tbody>
</table>

113 {  
114     if(!mutex_trylock(&ebbchar_mutex)){ // Try to acquire the mutex (returns 0 on fail)  
115         printk(KERN_ALERT "EBBChar: Device in use by another process");  
116         return -EBUSY;  
117     }  
118     numberOpens++;  
119     printk(KERN_INFO "EBBChar: Device has been opened %d time(s)\n", numberOpens);  
119     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 it uses the copy_to_user() function to send the buffer string to the user and captures any errors.

Parameters

<table>
<thead>
<tr>
<th>filep</th>
<th>A pointer to a file object (defined in linux/fs.h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>The pointer to the buffer to which this function writes the data</td>
</tr>
<tr>
<td>len</td>
<td>The length of the buffer</td>
</tr>
<tr>
<td>offset</td>
<td>The offset if required</td>
</tr>
</tbody>
</table>

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 {  

### 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*inodep</td>
<td>A pointer to an inode object (defined in linux/fs.h)</td>
</tr>
<tr>
<td>*filep</td>
<td>A pointer to a file object (defined in linux/fs.h)</td>
</tr>
</tbody>
</table>

```c
mutex_unlock(&ebbchar_mutex); // release the mutex (i.e., lock goes up)
printk(KERN_INFO "EBBChar: Device successfully closed\n");
return 0;
```

### 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*filep</td>
<td>A pointer to a file object</td>
</tr>
<tr>
<td>buffer</td>
<td>The buffer to that contains the string to write to the device</td>
</tr>
<tr>
<td>len</td>
<td>The length of the array of data that is being passed in the const char buffer</td>
</tr>
<tr>
<td>offset</td>
<td>The offset if required</td>
</tr>
</tbody>
</table>

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

### 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.

```c
mutex_destroy(&ebbchar_mutex); // destroy the dynamically-allocated mutex
device_destroy(ebbcharClass, MKDEV(majorNumber, 0)); // remove the device
class_unregister(ebbcharClass); // unregister the device class
class_destroy(ebbcharClass); // remove the device class number
unregister_chrdev(majorNumber, DEVICE_NAME); // unregister the major number
printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
```

### 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.

```c
mutex_destroy(&ebbchar_mutex); // destroy the dynamically-allocated mutex
device_destroy(ebbcharClass, MKDEV(majorNumber, 0)); // remove the device
class_unregister(ebbcharClass); // unregister the device class
class_destroy(ebbcharClass); // remove the device class number
unregister_chrdev(majorNumber, DEVICE_NAME); // unregister the major number
printk(KERN_INFO "EBBChar: Goodbye from the LKM!\n");
```
Returns

returns 0 if successful

```c
{  
  printk(KERN_INFO "EBBChar: Initializing the EBBChar LKM\n\n");  
  // Try to dynamically allocate a major number for the device -- more difficult but worth it  
  majorNumber = register_chrdev(0, DEVICE_NAME, &fops);  
  if (majorNumber<0) {  
    printk(KERN_ALERT "EBBChar failed to register a major number\n\n");  
    return majorNumber;  
  }  
  printk(KERN_INFO "EBBChar: registered correctly with major number %d\n", majorNumber);  
  // Register the device class  
  ebbcharClass = class_create(THIS_MODULE, CLASS_NAME);  
  if (IS_ERR(ebbcharClass)) { // Check for error and clean up if there is  
    unregister_chrdev(majorNumber, DEVICE_NAME);  
    printk(KERN_ALERT "Failed to register device class\n\n");  
    return PTR_ERR(ebbcharClass); // Correct way to return an error on a pointer  
  }  
  printk(KERN_INFO "EBBChar: device class registered correctly\n\n");  
  // Register the device driver  
  ebbcharDevice = device_create(ebbcharClass, NULL, MKDEV(  
    majorNumber, 0), NULL, DEVICE_NAME);  
  if (IS_ERR(ebbcharDevice)) { // Clean up if there is an error  
    unregister_chrdev(majorNumber, DEVICE_NAME);  
    printk(KERN_ALERT "Failed to create the device\n\n");  
    return PTR_ERR(ebbcharDevice);  
  }  
  printk(KERN_INFO "EBBChar: device class created correctly\n\n"); // Made it! device was initialized  
  mutex_init(&ebbchar_mutex); // Initialize the mutex dynamically  
  return 0;  
}
```

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 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:

```c
  =
  [.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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
```
Include dependency graph for testebbcharmbox.c:

```
/home/molloyd/exploringBB/extras/kernel/ebbcharmbox/testebbcharmbox.c
```

```
stdio.h  stdlib.h  errno.h  fcntl.h  string.h
```

### 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**

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 ( )

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

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.

```c
#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:

```
/home/molloyd/exploringBB/extras/kernel/gpio_test/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.
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.

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

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

```c
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",
61 gpio_get_value(gpioButton));
```
// GPIO numbers and IRQ numbers are not the same! This function performs the mapping for us
irqNumber = gpio_to_irq(gpioButton);
printk(KERN_INFO "GPIO_TEST: The button is mapped to IRQ: %d\n", irqNumber);

// This next call requests an interrupt line
result = request_irq(irqNumber, // The interrupt number requested
    ebbgpio_irq_handler, // The pointer to the handler function below
    IRQF_TRIGGER_RISING, // Interrupt on rising edge (button press, not release)
    NULL); // The *dev_id for shared interrupt lines, NULL is okay

printk(KERN_INFO "GPIO_TEST: The interrupt request result is: %d\n", result);
return result;
}

#### 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**

<table>
<thead>
<tr>
<th>irq</th>
<th>the IRQ number that is associated with the GPIO – useful for logging.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev_id</td>
<td>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.</td>
</tr>
<tr>
<td>regs</td>
<td>h/w specific register values – only really ever used for debugging. return returns IRQ_HANDLED if successful – should return IRQ_NONE otherwise.</td>
</tr>
</tbody>
</table>

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

#### 2.6.4 MODULE_AUTHOR ( "Derek Molloy" )

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

#### 2.6.6 module_exit ( ebbgpio_exit )

#### 2.6.7 module_init ( ebbgpio_init )

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

#### 2.6.8 MODULE_LICENSE ( "GPL" )

#### 2.6.9 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.

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

Include dependency graph for hello.c:

```
/home/molloyd/exploringBB/extras/kernel/hello/hello.c
linux/init.h  linux/module.h  linux/kernel.h
```

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.
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.

```c
71 printk(KERN_INFO "EBB: Goodbye %s from the BBB LKM!\n", name);
```
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”.

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.

```c
#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

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

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

The pointer to the thread task.

The LED Flasher main kthread loop

Generated on Sat Apr 25 2015 15:26:35 for Exploring BeagleBone: LKMs (by Derek Molloy) by Doxygen
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg</code></td>
<td>A void pointer used in order to pass data to the thread</td>
</tr>
</tbody>
</table>

Returns

returns 0 if successful

```c
prnte(KERN_INFO "EBB LED: Thread has started running \n");
while(!kthread_should_stop()){ // Returns true when kthread_stop() is called
    set_current_state(TASK_RUNNING);
    if (mode==FLASH) ledOn = !ledOn; // Invert the LED state
    else if (mode==ON) ledOn = true;
    else ledOn = false;
    gpio_set_value(gpioLED, ledOn); // Use the LED state to light/tum off the LED
    set_current_state(TASK_INTERRUPTIBLE);
    msleep(blinkPeriod/2); // millisecond sleep for half of the period
}
prnte(KERN_INFO "EBB LED: Thread has run to completion \n");
return 0;
```

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kobj</code></td>
<td>represents a kernel object device that appears in the sysfs filesystem</td>
</tr>
<tr>
<td><code>attr</code></td>
<td>the pointer to the kobj_attribute struct</td>
</tr>
<tr>
<td><code>buf</code></td>
<td>the buffer to which to write the number of presses</td>
</tr>
</tbody>
</table>

Returns

return the number of characters of the mode string successfully displayed

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

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.

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

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)"

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, "%d", &period); // Read in the period as an unsigned int
70 if ((period>1)&(period<=10000)) { // Must be 2ms or greater, 10s 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 = ebbAttrs,
}

Generated on Sat Apr 25 2015 15:26:35 for Exploring BeagleBone: LKMs (by Derek Molloy) by Doxygen
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:

```c
= {
    &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_attribute` The `kobj_attribute` has an attribute `attr` (name and mode), show and store function pointers The period variable is associated with the `blink_period` 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`. 

Generated on Sat Apr 25 2015 15:26:35 for Exploring BeagleBone: LKMs (by Derek Molloy) by Doxygen
Index

FLASH
    led.c, 34

led.c
    FLASH, 34
    OFF, 34
    ON, 34

OFF
    led.c, 34

ON
    led.c, 34