

# Lab 5

## GPIO and Interrupt Management Using Kernel Modules

### Embedded Linux

Julia Desmazes  
Michael Nissen

November 11, 2017

## 1 Configurable LED Blink Module

### 1.1 Simple GPIO Blink Module

### 1.2 sysfs-based User Interface

The goal here is to create a `sysfs`-based user interface that allows the user to change the LED blinking frequency at runtime. When the module is loading, it is initially going to display a message and blink the LED using a default timer with a period of 1 second.

In the Code Snippet 1 below, we can see the `__init` function, that is a macro used to mark some functions or initialized data as an ‘initialisation’ function. The kernel can then take this as a hint that the function is used only during the initialisation phase, and free up used memory resources after. Essentially, the kernel build system looks for all of the functions flagged with `__init`, across all of the pieces of the kernel, and arranges them so that they will all be in the same block of memory. Then, when the kernel boots, it can free that one block of memory at once.

The code starts by printing a message to the terminal. It is then requesting a single GPIO pin with an initial setup using the `gpio_request_one`. As seen in the comments for the code, it takes the arguments of the GPIO pin number, a flag the specified the initial GPIO configuration, and a label with a literal description string of this GPIO. The result is stored in a global `result` variable, that we can use to check if the request and initialisation of the GPIO pin was successful or not. If `result` is not 0, it will print error message to the terminal saying that it could not initialise the GPIO pin, otherwise it will print a statement saying it was successful.

We then initialise our timer by `init_timer(&my_timer)`, set an expiration time (1 second), and define what function will be executed once the timer expires. We want to execute our `blink_timer` function, that handles the blinking. We then print out a status message saying that the timer has been initialised, and start the timer.

The timer LED should now be starting with an initial configuration of `HIGH`, and blinking on and off every 1 second.

```
1 static int __init start_blink(void) {
2
3     printk("Blink :: start module :\n");
4
5     // Request a single GPIO with initial configuration HIGH: <PIN, FLAGS, label>
6     result = gpio_request_one(GPIO_PIN_NUMB, GPIOF_OUT_INIT_HIGH, "sysfs");
7
8     if (result) {
9
10        printk(KERN_INFO "ERROR :: Failed to initialise the GPIO PIN: %d\n",
11        GPIO_PIN_NUMB);
12
13        return result;
14    } else {
```

```

14     printk(KERN_INFO "SUCCESS :: GPIO PIN was successfully allocated\n");
15 }
16
17 // Initialise the timer
18 init_timer(&my_timer);
19
20 // Set expiration time in jiffies - convert period to milliseconds
21 my_timer->expires = jiffies + HZ*period/1000;
22
23 // Define the function to be executed upon timer expire
24 my_timer->function = blink_timer;
25
26 my_timer->data = 0;
27
28 printk(KERN_INFO "STATUS :: The timer has been initialised\n");
29
30 // Start the timer
31 add_timer(&my_timer);
32
33 ...
34 }
35

```

Code Snippet 1: Initial Configuration of LED

Now that we have our initial configuration up and running, we want to register a device class. A class is a higher-level view of a device that abstracts out low-level implementation details and can be found in `sysfs` under the `/sys/class/` folder. In many cases, the class subsystem is the best way of exporting information to the user space. When a subsystem creates a class, it owns the class entirely, so there is no need to worry about which module owns the attributes found there.

A class is defined by an instance of `struct class` class. Each class needs a unique name, which is how the class appears under `/sys/class`.

If we take a look at Code Snippet 2, we start off by creating a `struct class` class structure using the `class_create` function. We pass in the module that owns the class, and a unique string for the name of the class - this is defined in the `#define CLASS_NAME` at the start of the code. We then check if there was an error creating the class - if there was, we print out an error message saying that the creation of the device class has failed and return the error, otherwise we print a success statement, saying it was created.

We then go on to register the device object with `sysfs` using the `device_create` which takes as arguments, the class it should be registered to, the parent struct device, the `dev_t` to be added, and a unique string for the device name. As name of the class has been defined as `external_led` and the name of the device `blink`, the device object can be found under `/sys/class/external_led/blink`.

As usual, we check if the device object was created successfully and if not we destroy the `struct class` class structure with the `class_destroy` and print the error message, else we print a success message.

```

1 #define DEVICE_NAME "blink"
2 #define CLASS_NAME "external_led"
3
4 static int __init start_blink(void) {
5
6     ...
7
8     // Create a struct class structure <Module that owns the class, String for the name
9     // of the class>
10    external_leds = class_create(THIS_MODULE, CLASS_NAME);
11
12    if (IS_ERR(external_leds)) {
13
14        printk(KERN_ERR "ERROR :: Failed to create device class\n");
15
16        return PTR_ERR(external_leds);
17    } else {
18
19        printk(KERN_INFO "SUCCESS :: EXTERNAL_LEDS :: Device class was successfully
20        created\n");
21    }
22
23    // Create a device and register it with sysfs:
24

```

```

22 // <Class it should be registered to, Parent struct device, dev_t to be added, String
    for device name>
23 blink = device_create(external_leds, NULL, 0, NULL, DEVICE_NAME);
24
25 if (IS_ERR(blink)) {
26
27     // Destroy the struct class structure if there was an error
28     class_destroy(external_leds);
29
30     printk(KERN_ERR "ERROR :: Failed to create the device\n");
31
32     return PTR_ERR(blink);
33 } else {
34
35     printk(KERN_INFO "SUCCESS :: Device class created successfully\n");
36 }
37
38 ...
39 }

```

Code Snippet 2: Registering Device Class and Object

We now go on to create a **sysfs** attribute file for the device using the `device_create_file()` which takes the device created with `device_create()` and the device attribute descriptor as arguments. The device attribute descriptor is a function. The device attribute descriptor is a file, that if an integer is written to it, it can later be read out. A simple attribute has no means by which it can be read or written to - it needs wrapper routines for reading and writing. For this purpose, **kobjects** defines a special structure `struct kobj_attribute` where the first argument represents the name of the attribute file, the second argument is the file permission (0644 which is read/write for the owning user and read for group/other), the third argument is the function to be called when the file is read, and the last argument is the function to be called when the file is written.

We define the name, permission, and functions to be executed when using the attribute file, and then we create the device file passing in the device and the pointer to the address of the `kobj_attribute`. We then check if the device file was successfully created or not, and print out a relevant message depending on the result.

As we gave the file the name `period`, it can be written to by echoing an integer value into the file placed in `/sys/class/external_led/blink/period`.

This can be seen in Code Snippet 3, below.

```

1 // Device attribute descriptor: <attribute name, permission, function, function>
2 static struct kobj_attribute period_attr = __ATTR(period, 0644, NULL, store_period);
3
4 static int __init start_blink(void) {
5
6     ...
7
8     // Create sysfs attribute file for device: <device, device attribute descriptor>
9     result = device_create_file(blink, &period_attr);
10
11     if (result) {
12
13         printk(KERN_ERR "ERROR :: Failed to create the sysfs file\n");
14     } else {
15
16         printk(KERN_INFO "SUCCESS :: sysfs file successfully created\n");
17     }
18
19     ...
20
21 }

```

Code Snippet 3: Creating a **sysfs** Attribute File

By unloading the module, the extension shall delete the **sysfs**-based interface, display a termination message, delete the timer, stop blinking the LED, and release the **GPIO** request. All of this is done using the `__exit` macro. The `__exit` macro is used to deallocate any and all resources it has acquired, as well as any hardware it may have activated. The exit routine is the last chance that the module has to perform these operations, or it may leave the system in an unstable state.

Taking a look at the code in Code Snippet 4, we start by setting a value of low on the GPIO pin to turn off the LED, we free up the GPIO pin, deactivate the timer, remove and unregister the device class and object, destroying the class structure and removing the `sysfs` attribute file from its own method.

```

1 static void __exit stop_blink(void) {
2
3     // Turn off LED
4     gpio_set_value(GPIO_PIN_NUMB, LED_OFF);
5
6     printk(KERN_INFO "BLINK :: LED turned OFF\n");
7
8     // Free the GPIO PIN
9     gpio_free(GPIO_PIN_NUMB);
10
11    printk(KERN_INFO "BLINK :: GPIO freed\n");
12
13    // Deactivate the timer
14    del_timer(&my_timer);
15
16    printk(KERN_INFO "BLINK :: Timer deactivated\n");
17
18    // Remove the device created by device_create
19    device_destroy(external_leds, 0);
20
21    // Unregister the previously registered class
22    class_unregister(external_leds);
23
24    // Destroy the struct class structure
25    class_destroy(external_leds);
26
27    // Remove sysfs attribute file from its own method
28    device_remove_file(blink, &period_attr);
29
30    printk(KERN_INFO "BLINK :: The module has been unloaded\n");
31 }

```

Code Snippet 4: Unloading of the Module

Last but not least, we have the function that is to be executed when the timer expires. This function will perform a simple investigation of the initial value of the `jiffies`. `jiffies` is a global variable which only usage is to store the number of ticks occurred since system start-up. On kernel boot-up, `jiffies` is initialised to a special initial value, and it is incremented by one for each timer interrupt. Since there are HZ ticks occurring in one second, there are HZ `jiffies` in a second.

The function will then investigate the status of the LED using the global variable `led_status` - is it on or is it off, and determine the correct behavior based on the status. If the LED is on, we set the value of the GPIO pin to 0 with `gpio_set_value()`, set the `led_status` to 0 and print a status message saying that the LED was turned off. If the LED is off, the opposite is happening.

The expiration time of the timer is then set to the value specified in the period `sysfs` attribute file, and the the timer is started again. This can be seen in Code Snippet 5, below.

```

1 unsigned int led_status = LED_ON;
2
3 static void blink_timer() {
4
5     // Investigate initial value of jiffies
6     printk(KERN_INFO "Jiffies %lu :", jiffies);
7
8     // If the LED is on - turn it off / else turn it on
9     if (led_status) {
10
11         gpio_set_value(GPIO_PIN_NUMB, LED_OFF);
12         led_status = 0;
13         printk(KERN_INFO "SET :: LED OFF\n");
14     } else {
15
16         gpio_set_value(GPIO_PIN_NUMB, LED_ON);
17         led_status = 1;
18         printk(KERN_INFO "SET :: LED ON\n");
19     }
20 }

```

```
21 // Set expiration time in jiffies - convert period to milliseconds
22 my_timer->expires = jiffies + HZ * period / 1000;
23
24 // Start the timer
25 add_timer(&my_timer);
26 }
```

Code Snippet 5: The Blink Function to be Executed on Timer End

## 2 Button-Controlled LED Blink Module

### References

- [1] Torvalds: Linux Kernel Source Tree,  
<https://github.com/torvalds/linux>
- [2] J. Corbet, A. Rubini, G. Kroah-Hartman: Linux Device Drivers,  
<https://lwn.net/Kernel/LDD3/>
- [3] The Kernel Development Community: The Linux Kernel,  
<https://01.org/linuxgraphics/gfx-docs/drm/index.html>
- [4] 0xAX: Linux Inside,  
<https://www.gitbook.com/book/0xax/linux-insides/details>