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 requsting 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 rusult 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 successfull.
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 that 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",
       GPIO_PIN_NUMB);
11
12         return result;
13     } else {
```

```
        printk(KERN_INFO "SUCCESS :: GPIO PIN was successfully allocated\n");
    }

    // Initialise the timer
    init_timer(&my_timer);

    // Set expiration time in jiffies - convert period to milliseconds
    my_timer->expires = jiffies + HZ*period/1000;

    // Define the function to be executed upon timer expire
    my_timer->function = blink_timer;

    my_timer->data = 0;

    printk(KERN_INFO "STATUS :: The timer has been initialised\n");

    // Start the timer
    add_timer(&my_timer);

    ...
}
```

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 `syfs` 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. 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 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 structure with the `class_destroy` and print the error message, else we print a success message.

```
#define DEVICE_NAME "blink"
#define CLASS_NAME  "external_led"

static int __init start_blink(void) {

    ...

    // Create a struct class structure <Module that owns the class, String for the name
    of the class>
    external_leds = class_create(THIS_MODULE, CLASS_NAME);

    if (IS_ERR(external_leds)) {

        printk(KERN_ERR "ERROR :: Failed to create device class\n");

        return PTR_ERR(external_leds);
    } else {

        printk(KERN_INFO "SUCCESS :: EXTERNAL_LEDS :: Device class was successfully
    created\n");
    }

    // Create a device and register it with sysfs:
```

```
    // <Class it should be registered to, Parent struct device, dev_t to be added, String
     for device name>
    blink = device_create(external_leds, NULL, 0, NULL, DEVICE_NAME);

    if (IS_ERR(blink)) {

        // Destroy the struct class structure if the was an error
        class_destroy(external_leds);

        printk(KERN_ERR "ERROR :: Failed to create the device\n");

        return PTR_ERR(blink);
    } else {

        printk(KERN_INFO "SUCCESS :: Device class created successfully\n");
    }

    ...
}
```

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 descritor is a function, The device attribute descriptor is a file, that if an interger 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 the `kobj_attribute`. We then check if the device file was succesfully 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`.

```
// Device attribute descriptor: <attribute name, permission, function, function>
static struct kobj_attribute period_attr = __ATTR(period, 0644, NULL, store_period);

static int __init start_blink(void) {

    ...

    // Create sysfs attribute file for device: <device, device attribute descriptor>
    result = device_create_file(blink, &period_attr);

    if (result) {

        printk(KERN_ERR "ERROR :: Failed to create the sysfs file\n");
    } else {

        printk(KERN_INFO "SUCCESS :: sysfs file successfully created\n");
    }

    ...

}
```

Code Snippet 3: Creating a `sysfs` attribute file

By unloading the module, the extention 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 3, 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.

```
static void __exit stop_blink(void) {

    // Turn off LED
    gpio_set_value(GPIO_PIN_NUMB, LED_OFF);

    printk(KERN_INFO "BLINK :: LED turned OFF\n");

    // Free the GPIO PIN
    gpio_free(GPIO_PIN_NUMB);

    printk(KERN_INFO "BLINK :: GPIO freed\n");

    // Deactivate the timer
    del_timer(&my_timer);

    printk(KERN_INFO "BLINK :: Timer deactivated\n");

    // Remove the device created by device_create
    device_destroy(external_leds, 0);

    // Unregister the previously registered class
    class_unregister(external_leds);

    // Destroy the struct class structure
    class_destroy(external_leds);

    // Remove sysfs attribute file from its own method
    device_remove_file(blink, &period_attr);

    printk(KERN_INFO "BLINK :: The module has been unloaded\n");
}
```

Code Snippet 4: Unloading of the Module

2 Button-Controlled LED Blink Module