Implementación del driver básico de Blinkstick Strip

Juan Carlos Sáez Alcaide

Índice

1 Introducción **1**

2 Registro del driver USB **1**

3 Estructura de estado **3**

4 Gestión de ficheros especiales de dispositivo **5**

5 Modificando el estado de los LEDs de Blinkstick Strip **7**

1 Introducción

En este breve artículo se describe la implementación del driver `blinkdrv.c` , que se proporciona como parte de la práctica 3. La funcionalidad de este driver ha de ampliarse en la citada práctica, proporcionando una implementación alternativa de la función `blink_write()`, cuya versión básica se describe en la sección 5 de este documento. No obstante, este artículo describe toda la implementación del driver.

2 Registro del driver USB

Para que un módulo del kernel se comporte como un driver USB debe implementar la interfaz `usb_driver` declarada en `<linux/usb.h>`:

```
struct usb_driver {
    const char *name;
    int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id);
    void (*disconnect) (struct usb_interface *intf);
    int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
            void *buf);
    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    int (*resume) (struct usb_interface *intf);
    int (*reset_resume)(struct usb_interface *intf);

    int (*pre_reset)(struct usb_interface *intf);
    int (*post_reset)(struct usb_interface *intf);

    const struct usb_device_id *id_table;
    const struct attribute_group **dev_groups;

    struct usb_dynids dynids;
```

```
    struct usbdrv_wrap drvwrap;
    unsigned int no_dynamic_id:1;
    unsigned int supports_autosuspend:1;
    unsigned int disable_hub_initiated_lpm:1;
    unsigned int soft_unbind:1;
};
```

A pesar de que `usb_driver` está definida como una estructura, nos referiremos a ella como interfaz porque muchos de sus campos son punteros a función, como en otras estructuras del kernel como `proc_ops` o `file_operations`.

Al igual que sucede al gestionar entradas `/proc` no es necesario que el driver implemente todas las operaciones de la interfaz. Basta con implementar las operaciones mínimas para garantizar el correcto funcionamiento del dispositivo.

Los campos de `struct usb_driver` que la mayor parte de drivers USB en Linux suelen definir son los siguientes:

- `name`: cadena de caracteres arbitraria que representa el nombre del driver
- `id_table`: tabla donde se especifican qué dispositivos son compatibles con el driver. Cada dispositivo compatible se especifica mediante su *vendor ID* y su *product ID*
- `probe`: Función del driver que se ejecuta cuando el USB core detecta que se ha conectado al *host* un dispositivo compatible con nuestro driver USB. El descriptor de la interfaz USB del dispositivo (`struct usb_interface *`) se pasa como parámetro a la función.
- `disconnect`: Función del driver que se ejecuta cuando desconectamos del *host* uno de los dispositivos USB gestionados por el driver

El driver básico del dispositivo Blinkstick solo define los cuatro campos básicos de la interfaz `usb_driver`. Para más información sobre el resto de campos y operaciones se ha de consultar las fuentes del núcleo.

A continuación se muestra el fragmento de código del driver `blinkdrv.c` (módulo del kernel) donde se lleva a cabo la inicialización de la interfaz y el registro del driver USB:

```
static struct usb_driver blink_driver = {
    .name =     "blinkstick",
    .probe =    blink_probe,
    .disconnect =   blink_disconnect,
    .id_table = blink_table,
};

/* Module initialization */
int blinkdrv_module_init(void)
{
   return usb_register(&blink_driver);
}

/* Module cleanup function */
void blinkdrv_module_cleanup(void)
{
  usb_deregister(&blink_driver);
}

module_init(blinkdrv_module_init);
module_exit(blinkdrv_module_cleanup);
```

Como puede observarse, el registro del driver se produce en la función de inicialización del módulo (`blinkdrv_module_init()`) y el desregistro en la función de limpieza (`blinkdrv_module_cleanup()`). Para llevar a cabo el registro del driver se invoca la función `usb_register()`, que acepta como parámetro un puntero a la interfaz `usb_driver` que se instancia en la parte superior del fragmento de código (variable `blink_driver`). En el caso de que el registro del driver se produzca correctamente la función `usb_register()` devuelve 0; en caso de error se devuelve un número negativo que codifica el error. La descarga del driver es análoga; basta invocar la función `usb_deregister()` en la

función de limpieza del módulo, pasando como parámetro el puntero a la interfaz `usb_driver` correspondiente.

La instanciación de la interfaz `usb_driver` conlleva la definición de una variable global (variable `blink_driver`), para la cual solo se inicializan los cuatro campos mencionados previamente: nombre del driver, punteros a función a las operaciones `probe` y `disconnect` del driver (funciones `blink_probe()` y `blink_disconnect()` definidas en el propio driver), y tabla de dispositivos compatibles. Esta tabla (variable global `blink_table`) se define como sigue en otra parte del módulo del kernel:

```
#define BLINKSTICK_VENDOR_ID    0x20A0
#define BLINKSTICK_PRODUCT_ID   0x41E5

/* table of devices that work with this driver */
static const struct usb_device_id blink_table[] = {
    { USB_DEVICE(BLINKSTICK_VENDOR_ID,  BLINKSTICK_PRODUCT_ID) },
    { }                 /* Terminating entry */
};
```

La tabla es un array que almacena todos los identificadores de dispositivos (`usb_device_id`) que el driver es capaz de gestionar. Cada posición del array es un descriptor de dispositivo, que ha de incializarse con la macro `USB_DEVICE()`. Esta macro acepta como parámetro el *vendor ID* y *product ID* del dispositivo en formato hexadecimal. Recordad que estos números (hardcodeados en las fuentes del driver) pueden obtenerse para cualquier dispositivo conectado al sistema haciendo uso del comando `usb-devices` o consultando la entrada correspondiente del *sysfs*. Para que el kernel sepa cuántas entradas tiene este array, es preciso poner explícitamente un terminador al final con "`{}`".

Gracias al modo en el que el driver se registra, y considerando la información en la tabla `blink_table`, cada vez que se conecte un dispositivo USB con `vendor_id=020A0` y `device_id=0x41e5` el USB core invocará la operación `probe` del driver USB, que en este caso se implementa mediante la función `blink_probe()` del módulo del kernel.

3 Estructura de estado

El driver `blinkdrv.c` es capaz de gestionar múltiples dispositivos Blinkstick Strip conectados al *host*. En general, todo driver USB que esté preparado para trabajar con mútiples dispositivos a la vez, debe asociar a cada dispositivo lógico una estructura que represente su estado. Los campos de esta estructura los decide el programador del driver, por lo tanto su definición es dependiente de implementación y de dispositivo.

El driver básico para Blinkstick Strip define la siguiente estructura de estado:

```
struct usb_blink {
    struct usb_device   *udev;
    struct usb_interface    *interface;
    struct kref     kref;
};
```

A continuación se describe el propósito de los distintos campos:

- `udev`: es un puntero al descriptor del dispositivo físico USB (`struct usb_device`). Esta estructura ha de pasarse como parámetro a muchas funciones de USB core para realizar transferencias de datos entre el *host* y el dispositivo. Por lo tanto está presente en casi todas las estructuras de estado de los drivers USB. Este puntero puede obtenerse en la operación `probe()` del driver, que es donde se crea e inicializa la estructura de estado del dispositivo.
- `interface`: se trata de puntero al descriptor de la interfaz USB que el driver gestiona (`struct usb_interface`). En este caso, representa al único dispositivo lógico que posee Blinkstick Strip. Al igual que el campo `udev`, este puntero debe recuperarse en `probe()`, y su inclusión en estructuras de estado de drivers USB es muy frecuente.

- `kref`: Es el contador de referencias (`struct kref`) de la estructura de estado. En el kernel, el tipo de datos (`struct kref`) se usa específicamente para implementar contadores de referencia de objetos del núcleo, que son críticos para determinar cuándo es seguro liberar la memoria de un objeto. Este tipo de datos tiene una serie de operaciones seguras desde el punto de vista de la concurrencia para realizar incrementos o decrementos del mismo. En particular la inicialización del contador se realiza con la función `kref_init()`, y los incrementos y decrementos del mismo se llevan a cabo invocando las funciones `kref_get()` y `kref_put()`, respectivamente.

  En el driver, el contador de referencia `kref` se usa para (1) gestionar correctamente la memoria de la estructura de estado, y (2) resolver problemas de concurrencia asociados a la desconexión física del dispositivo cuando el driver está en uso. Esenciamente, la memoria del objeto de estado se gestiona dinámicamente. La solicitud de memoria se produce cuando el dispositivo se conecta al sistema (`probe()`); en ese momento el valor interno del contador es 1. Este contador de referencias se incrementa con `kref_get()` en el código del driver cada vez que un proceso de usuario esté trabajando con el fichero especial dispositivo, y se decrementa con `kref_put()` cuando dicho proceso deja de usarlo. Gracias al correcto mantenimiento del contador, la memoria de la estructura de estado se libera solamente cuando al decrementar el contador se alcanza el valor cero.

La creación de la estructura de estado que usa el driver (`struct usb_blink`), tiene lugar en la funcion `blink_probe()` del driver. Esta operación se invoca al conectar el dispositivo Blinkstick Strip al equipo, y recibe como primer parámetro la estructura que representa la interfaz USB. A continuación, se muestra el fragmento de código donde se produce la inicialización de la estructura de estado:

```
static int blink_probe(struct usb_interface *interface,
              const struct usb_device_id *id)
{
    struct usb_blink *dev;
    int retval = -ENOMEM;

   /* Allocate memory for a usb_blink structure. */
    dev = kmalloc(sizeof(struct usb_blink), GFP_KERNEL);

    if (!dev) {
        dev_err(&interface->dev, "Out of memory\n");
        goto error;
    }

    /* Initialize the various fields in the usb_blink structure */
    kref_init(&dev->kref);
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;

    /* save our data pointer in this interface device */
    usb_set_intfdata(interface, dev);
...
..
}
```

Como puede observarse, la memoria de la estructura se reserva con `kmalloc()`, y a continuación se procede a inicializar manualmente sus tres campos. El puntero a la estructura `struct usb_device` se recupera a partir del parámetro `interface` usando de forma encadenada dos macros del USB core:

```
dev->udev = usb_get_dev(interface_to_usbdev(interface));
```

Tras la inicialización se asocia la estructura de estado con la interfaz, usando la siguiente llamada:

```
/* save our data pointer in this interface device */
usb_set_intfdata(interface, dev);
```

Esto es esencial para que la estructura de estado pueda recuperarse desde las funciones del driver que implementan operaciones sobre los ficheros especiales de caracteres que representan dispositivos BlinkStick Strip. En particular, sin esta asociación no sería posible realizar transferencias de datos entre el dispositivo y el kernel en la operación `blink_write()`, que es la que ha de modificarse como parte de la práctica.

Para recuperar la estructura `usb_blink` desde la función de escritura `blink_write()`, se llevan a cabo las siguientes acciones en la implementación del driver:

1. Cuando un programa abre el dispositivo, la operación correspondiente – `blink_open()` – recupera el puntero a la estructura a partir de la estructura `inode` pasada como parámetro
2. El puntero a `usb_blink` se almacena en el campo `private_data` del parámetro `file` (`struct file*`), pasado a `blink_open()`
3. El parámetro `file` se pasa también a `blink_write()`. De este modo se recupera el puntero a `usb_blink` mediante el campo `private_data`.

4 Gestión de ficheros especiales de dispositivo

Para poder interactuar con el driver desde espacio de usuario es preciso registrar los dispositivos reconocidos (uno por cada Blinkstick Strip conectado al sistema) en una clase del Linux Device Model (LDM). USB core asigna major numbers diferentes para distintas clases de dispositivos USB: `input`, `usb`, `ttyACM`, `tty_usb`,`hiddev`, etc.

La clase `usb` (representada en `/sys/class/usbmisc`) es la clase genérica en Linux para la mayoría de dispositivos USB. Los dispositivos de esta clase tienen asociado el major number 180. Para hacer uso de esta clase y major number, el driver básico de Blinkstick Strip registra cada dispositivo conectado mediante la función `usb_register_dev()`, que realiza las siguientes acciones:

1. Registro del dispositivo en la clase genérica `usb` y asignación un minor number para el mismo.
2. Creación automática del fichero de dispositivo en `/dev`, comunicándose con el servicio Udev.
3. Asignación de la interfaz de operaciones (`file_operations`) al fichero de dispositivo

La función acepta dos parámetros:

```
extern int usb_register_dev(struct usb_interface *intf,
                struct usb_class_driver *class_driver);
```

El primer parámetro `intf` es el descriptor de la interfaz USB que ya viene como parámetro de la operación `probe()` del driver.

El segundo parámetro `class` es un puntero a una variable global de tipo `struct usb_class_driver` que ha de definirse en el código del driver. Al iniciar esta compleja estructura indicaremos al kernel Linux (1) qué minor number queremos asignar al dispositivo, (2) cómo se ha de llamar el fichero de dispositivo en `/dev`, y qué permisos tiene, y (3) qué operaciones define el driver sobre el fichero de dispositivo.

La estructura `usb_class_driver` se define como sigue:

```
struct usb_class_driver {
    char *name;
    char *(*devnode)(struct device *dev, umode_t *mode);
    const struct file_operations *fops;
    int minor_base;
};
```

El propósito de sus campos es el siguiente:

- `name`: es una cadena de caracteres (patrón) que codifica el nombre al dispositivo, así como el de su fichero especial asociado

- `devnode`: es una función a definir en el driver que permite indicar:
  1. La ruta relativa (a `/dev`) donde se creará el fichero de dispositivo
  2. Los permisos del fichero de dispositivo (parámetro `mode`)
- `file_operations`: es un puntero a las operaciones que el driver ejecutará cuando algún programa acceda al dispositivo. El driver debe instanciar la estructura `file_operations` e implementar las operaciones.
- `minor_base`: Indica el comienzo del rango de minors asignados para este driver.

El siguiente fragmento de código muestra la definición de la estructura global de tipo `struct usb_class_driver`, que se pasa como parámetro a la función `usb_register_dev()` en `blink_probe()`:

```
#define USB_BLINK_MINOR_BASE    0

static const struct file_operations blink_fops = {
    .owner =    THIS_MODULE,
    .write =    blink_write, /* write() syscall */
    .open =     blink_open,  /* open() syscall */
    .release =  blink_release, /* close() syscall */
};

char* set_device_permissions(struct device *dev, umode_t *mode)
{
    if (mode)
        (*mode)=0666; /* RW permissions */
    return kasprintf(GFP_KERNEL, "usb/%s", dev_name(dev));
}

static struct usb_class_driver blink_class = {
    .name =     "blinkstick%d",
    .devnode=   set_device_permissions,
    .fops =     &blink_fops,
    .minor_base =   USB_BLINK_MINOR_BASE,
};
```

Como puede observarse en la inicialización de `blink_class`, el campo `name` denota un patrón. Según este patrón el nombre de los ficheros de dispositivo serán *blinkstick0*, *blinkstick1*, *blinkstick2*, etc.

El segundo campo (`devnode`) se inicializa con la dirección de la función `set_device_permissions()` definida más arriba en el código. El valor de retorno de esta función indica a Udev dónde crear el fichero especial de dispositivo dentro de `/dev`. Más concretamente, la implementación establece que la ruta será `/dev/usb`. Además el parámetro de retorno `mode`, se emplea para devolver la máscara octal de permisos de este fichero de dispositivo (`666` según la implementación).

El tercer campo (`fops`) se inicializa con la dirección de memoria de la variable `blink_fops`, en cuya inicialización (parte superior del código) se asocian tres operaciones al driver – `blink_write()` , `blink_open()`, y `blink_release()`–, que se invocan cuando se ejecutan respectivamente las llamadas al sistema `write()`, `open()` y `close()` desde un programa de usuario sobre cualquier fichero especial de caracteres asociado al driver.

Finalmente el campo `minor_base` se inicializa con una macro definida al comienzo del código.

El siguiente fragmento de código de `blink_probe()` es el que invoca la función `usb_register_dev()`:

```
static int blink_probe(struct usb_interface *interface,
              const struct usb_device_id *id)
{
    struct usb_blink *dev;
    /* Initialization of the usb_blink structure */
    ...
```

```
    /* we can register the device now, as it is ready */
    retval = usb_register_dev(interface, &blink_class);
    if (retval) {
        /* something prevented us from registering this driver */
        dev_err(&interface->dev,
            "Not able to get a minor for this device.\n");
        usb_set_intfdata(interface, NULL);
        goto error;
    }

    /* let the user know what node this device is now attached to */
    dev_info(&interface->dev,
         "Blinkstick device now attached to blinkstick-%d",
         interface->minor);
    return 0;
    ...
}
```

De este código merece la pena reseñar que el driver utiliza la familia de macros `dev_xxx()` que constituyen un *wrapper* de `printk()` para imprimir mensajes en el fichero de log del kernel incluyendo un prefijo específico del dispositivo sobre el que se informa. Estás macros aceptan como primer parámetro un puntero al descriptor del dispositivo `struct device*`, que en este caso se obtiene a partir de la interfaz USB.

5 Modificando el estado de los LEDs de Blinkstick Strip

Para establecer el color de un LED del dispositivo Blinkstick Strip es preciso enviar un URB de control al dispositivo USB usando el *endpoint* 0. Este URB debe llevar un mensaje de 6 bytes con la siguiente estructura:

Byte	Contenido
0	`\x05`(wValue)
1	`0` (wIndex)
2	Número de LED que se quiere modificar (0..7)
3	Componente roja del color (0..FF)
4	Componente verde del color (0..FF)
5	Componente azul del color (0..FF)

Como se indica en la tabla, los dos primeros bytes tienen un valor fijo (5 y 0), que codifican un comando prejiado por el hardware para establecer el color de un LED. El tercer byte permite especificar el LED cuyo color deseamos alterar; los LEDs están numerados del 0 al 7, siendo el 0 el más próximo al conector USB, y el 7 el más lejano. Los 3 últimos bytes especifican la componente roja, verde y azul del color

Para enviar un URB mensaje de control al dispositivo el driver utiliza la función `usb_control_msg()` , que pertenece a la API síncrona de USB core. Esta función es bloqueante; la función no retorna hasta que el *host* tiene garantías de que el mensaje se ha recibido por parte del dispositivo (se espera a la confirmación de recepción), o hasta que se produce un error.

La función se define como sigue:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
    __u8 request, __u8 requesttype, __u16 value, __u16 index,
    void *data, __u16 size, int timeout);
```

Los 5 párametros más importantes son los siguientes:

- `dev`: puntero al descriptor del dispositivo USB. Se encuentra almacenado en la estructura de estado
- `pipe`: Canal para la comunicación. Se construye mediante la macro `usb_sndctrlpipe()`, a la que debe pasarse el número de *endpoint* por donde deseamos enviar el mensaje.
- `data`: puntero al buffer que contiene el mensaje. Este puntero ha de reservarse con `kmalloc()` pasando la opción `GFP_DMA` .
- `size`: tamaño en bytes del mensaje pasado mediante el parámetro `data`
- `timeout`: tiempo máximo (en ms) que puede durar la transacción. Si se pasa un valor mayor que cero, la transacción se cancela si se supera el tiempo indicado. Si se pasa el valor 0, no hay límite de tiempo.

Entender el propósito del resto de los parámetros exige conocimientos técnicos sobre la especificación USB 2.0 y más concretamente sobre el capítulo 9 de esta especificación. A continuación se proporciona una breve descripción de estos parámetros:

- `request`: Tipo de petición de control según el estándar. Para cada tipo de petición de control hay una macro definida en `<uapi/linux/usb/ch9.h>`. En el caso del Blinkstick Strip, el tipo de petición requerida para cambiar el estado de un LED es `USB_REQ_SET_CONFIGURATION`
- `requestType`: Máscara de bits que contiene las características de la petición (dirección, tipo de dispositivo y receptor del mensaje). Estas opciones también se definen en `<uapi/linux/usb/ch9.h>`. La máscara de bits usada en las peticiones de cambio de estado de los LEDs de Blinkstick Strip es `USB_DIR_OUT| USB_TYPE_CLASS | USB_RECIP_DEVICE`.
- `value` e `index`: Argumentos que codifican un comando, y dependen de la petición y del dispositivo. Los valores que es preciso pasar en la implementación del driver son `5` y `0`, que coinciden con los dos primeros bytes del mensaje de cambio de estado de un LED.

Finalmente, este fragmento de código (función `blink_write()`) recoge el punto exacto donde se invoca `usb_control_msg()`:

```
#define NR_BYTES_BLINK_MSG 6
#define NR_LEDS 8
static ssize_t blink_write(struct file *file, const char *user_buffer,
              size_t len, loff_t *off) {
    struct usb_blink *dev=file->private_data;
    int i=0;
    unsigned char* message=kmalloc(NR_BYTES_BLINK_MSG, GFP_DMA);

    /* Fill up the message accordingly */
    ...
    for (i=0;i<NR_LEDS;i++){
        message[2]=i; /* Change Led number in message */

        retval=usb_control_msg(dev->udev,
             usb_sndctrlpipe(dev->udev,00), /* Specify endpoint #0 */
             USB_REQ_SET_CONFIGURATION,
             USB_DIR_OUT| USB_TYPE_CLASS | USB_RECIP_DEVICE,
             0x5,    /* wValue */
             0,      /* wIndex=Endpoint # */
             message,    /* Pointer to the message */
             NR_BYTES_BLINK_MSG, /* message's size in bytes */
             0);
        ...
    }
...
```

En particular, la función `blink_write()` –la única que debe modificarse en la práctica– se invoca cada vez que se produce una escritura (p.ej., con `echo`) en el fichero especial de dispositivo. Como el Blinkstick Strip está provisto de 8

LEDs, en el código encontramos un bucle de 8 iteraciones, que en cada iteración envía un mensaje para alterar el estado de un LED específico.

En el fragmento de código mostrado más arriba no se incluye la parte de `blink_write()` que se ocupa de inicializar el mensaje almacenado en el buffer `message`. Forma parte de la práctica entender cómo se construye cada uno de los 8 mensajes que se envían. La invocación de `usb_control_msg()` puede comprenderse fácilmente gracias los comentarios y a la descripción de los parámetros proporcionada anteriormente.