USB Device Driver

Devices that can be supported:

`struct usb_device_id:`

`idVendor:`	e.g. `0A5c` is Broadcom
`idProduct:`	e.g `4500` is USB hub
`bdeviceClass:`	e.g. 9 Hub
`bdevSubclass:`	e.g. 0 Unused
`bdevProtocol:`	e.g. 0 Full speed hub

Macros for initialization:

```
USB_DEVICE(vendor, product)
USB_DEVICE_INFO(class, subclass, protocol)
```

Initialization:

```
#define USB_SKEL_VENDOR_ID 0x0A5C
#define USB_SKEL_PRODUCT_ID 0x4500
static struct usb_device_id skel_table [] = {
   { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
   { }              /* Terminating entry */};
MODULE_DEVICE_TABLE (usb, skel_table);
```

Register Driver

Callbacks:

struct usb_class_driver skel_class:

`struct module *owner`

Pointer to the module owner of this driver

`const char *name`

Pointer to the name of the driver

`const struct usb_device_id *id_table`

Pointer to the struct usb_device_id table that contains a list of all of the different kinds of USB devices this driver can accept

`int (*probe) ()`

Called when the driver thinks it has a struct usb_interface that the driver can handle

`void (*disconnect) ()`

Called when the struct usb_interface has been removed from the system or when the driver is being unloaded from the USB core.

Register Driver

Other Callbacks:

struct usb_class_driver skel_class:

`int (*ioctl) ()`

An ioctl function – allows commands to be issued from userland

`int (*suspend) ()`

Called to suspend the device

`int (*resume) ()`

Called to resume the device

Register Driver

Init function:

```
static int __init usb_skel_init(void) {
 int result;

 /* register this driver with the USB subsystem */
 result = usb_register(&skel_driver);
 if (result)
   printk(KERN_INFO "usb_register failed. Error \
          number %d", result);
 return result;
}
```

exit function:

```
static void __exit usb_skel_exit(void) {
 /* deregister this driver with the USB subsystem */
 usb_deregister(&skel_driver);
}
```

Probe

Called when a device is installed that the USB core thinks this driver should handle

Probe should make checks on information passed to it to make sure it should be handled

Should detect what the endpoint address and buffer sizes are for the device: they're needed to communicate with the device

Register the device with the USB subsystem using

```
  usb_register_dev(interface, &skel_class);
```

Note that `skel_class` contains a pointer to file_operations

File Operations

open function:

Get the interface with

```
interface = usb_find_interface(&skel_drvr, subminor);
```

Retrieve the data with

```
dev = usb_get_intfdata(interface);
```

read function:

Build a URB, send if off, wait for completion

udev	- pointer to device to send the message to
pipe	- endpoint "pipe" to send the message to
buffer	- pointer to the data to send
len	- length in bytes of the data to send
cnt	- pointer to loc to put # bytes actually transferred
HZ	- msecs to wait for completion before timing out

```
usb_bulk_msg(udev,usb_rcvbulkpipe(),buf,len,cnt,HZ)
```

File Operations

release function:

Decrement the device count with

```
kref_put(&dev->kref, skel_delete);
```

write function:

Allocate a URB
Allocate a DMA buffer
Copy data to the DMA buffer
Initialize the URB – include a callback for completing
Send the URB with the following

```
  usb_submit_urb(urb, GFP_KERNEL)
```

Free the URB space

After a Write Completes

callback function:

Check status of the URB to see if it completed normally

If not, an error is returned

Then the allocated buffer that was assigned to the URB is freed

```
skel_write_bulk_callback(struct urb *urb);
```