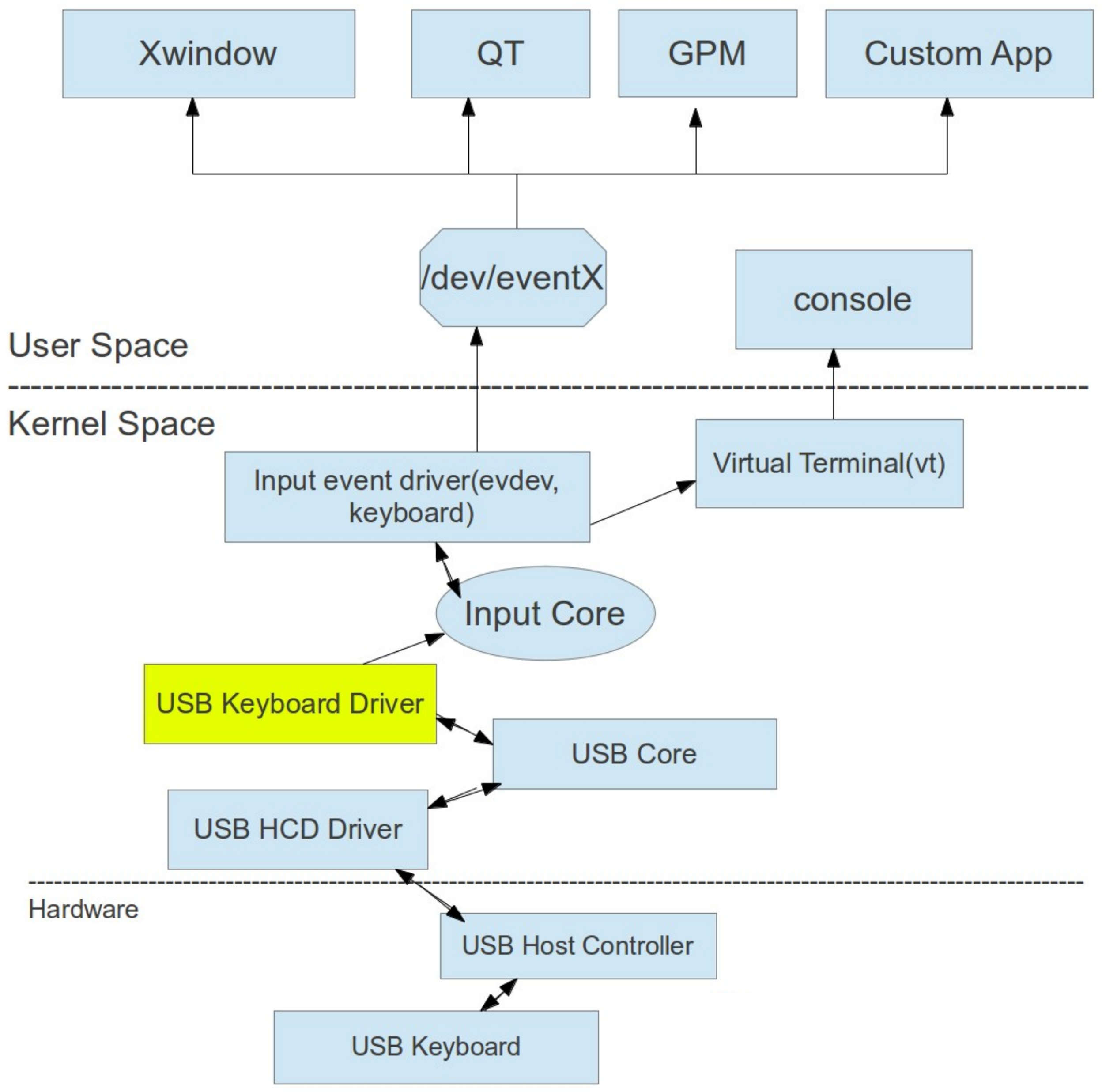
USB Keyboard Driver

VEDA Solutions
inspiring linux learning



This is the block diagram of how a keyboard data should flow. In the whole diagram, our device driver part is the one which is in yellow color. Our driver will get the data from the USB keyboard(through USB core and USB HCD Driver). Then the data is passed to the Linux Input subsystem. Linux Input subsystem passes the data to the applications like Xwindow, GPM, QT etc through the Input event drivers evdev. Data will be passed to the console through the Input event driver keyboard and Virtual Terminal(vt) code. So, our driver should register with the USB Core to get the data from the device and should register with Linux Input subsystem to pass data up to the applications. Now let's see each function of the driver.

Driver Init Function

usb_kbd_init() is the initialization routine of the driver. We are implementing this driver as a kernel module. usb_kbd_init is the module initialization routine. This will be called when the module is inserted.

Only one task it should do is to register with the USB Core using usb_register() function. Like this.

```
/*
 * usb_kbd_init, module initialization routine, register with the USB Core.
 */
static int __init usb_kbd_init(void)
{
	int ret;

	ret  = usb_register(&usb_kbd_driver);
	if (ret == 0)
		printk("usbkbd: Registered with USB Core");
	return ret;
}
```

usb_kbd_init() function will register with the USB Core using the usb_register() function. usb_register() function takes pointer to struct usb_driver object as its parameter. In our driver struct usb_driver object is defined as follows.

```
static struct usb_driver usb_kbd_driver = {
	.name =		"usbkbd",  // our device name
	.probe =	usb_kbd_probe, // called when our keyboard is found
	.disconnect =	usb_kbd_disconnect, // called when keyboard is removed
	.id_table =	usb_kbd_id_table,
};
```

We are filling the usb_kbd_driver object with the name of the driver, address of the probe function, address of the disconnect function and id table. id_table tells kernel for which device we are writing driver. usb_kbd_id_table is defined as follows.

```
/*
 * udb_kbd_id_table: Table that specify what device we are dealing with. We
 * are interrested in Human Interface device(HID). We match our device against
 * interface descriptor, So we tell usb to search for a device which has class
 * as 3(HID), subclass as 1(boot interface) and protocol as 1(keyboard).
 * refer  4.1, 4.2 and 4.3 sections of USB HID Specification 1.1.
 */
static struct usb_device_id usb_kbd_id_table [] = {
	{ USB_INTERFACE_INFO(3, 1, 1) },
	{ }					/* Terminating entry */
};
```

This table tells that we are interrested in those USB devices that have the interface class is HID, and subclass as 1(boot interface) and protocol as 1(keyboard). When ever a device with this particular parameters is inserted, our drivers probe call back function will be called.

Driver Probe Routine

Drivers probe routine is usb_kbd_probe(). This function will be invoked by the USB Core whenever the USB device that we want is attached to the computer. Basic responsibilities of the probe function will be to get the required endpoints, allocate required memory and to register with the input subsystem.

First step is to get the endpoint information. We want only Interrupt In endpoint, so get that. Code for this is like this.

```
/*
        * Get usb_device object where this interface is on.
        */
         struct usb_device *dev = interface_to_usbdev(iface);
         struct usb_host_interface *interface;
         struct usb_endpoint_descriptor *endpoint;
         struct usb_kbd *kbd; // our private structure.
         struct input_dev *input_dev; // to register with input subsystem
         int i, pipe, maxp;
         int error = -ENOMEM;

         interface = iface->cur_altsetting;

         /*
        * We dont support more than one endpoint. We support interrupt In
        * endpoint along with default controll endpoint.
        */
         if (interface->desc.bNumEndpoints != 1)
                 return -ENODEV;
         /*
        * Get pointer to the end point descriptor and check if it
        * interrupt In endpoint
        * bEndpointAddress field of
        * endpoint descriptor is like this.
        *
        * Bit 3...0: The endpoint number
        * Bit 6...4: Reserved, reset to zero
        * Bit 7:    Direction, ignored for control endpoints
        *          0 = OUT endpoint
        *          1 = IN endpoint
        * To check if this is IN endpoint code looks like this:
        *
        *    if ((endpoint->bEndpointAddress & 0x80)).
        *
        * bAttributes field of endpoint descriptor is like this:
        *
        * Bits 1..0: Transfer Type
        *           00 = Control
        *           01 = Isochronous
        *           10 = Bulk
        *           11 = Interrupt
        * Bits 3..2: Synchronization Type
        *           00 = No Synchronization
        *         01 = Asynchronous
         *         10 = Adaptive
        *         11 = Synchronous
        * Bits 5..4: Usage Type
        *         00 = Data endpoint
        *         01 = Feedback endpoint
        *         10 = Implicit feedback Data endpoint
        *         11 = Reserved
        *
        * To Check if an endpoint is Interrupt Endpoint cod looks like:
        *     if ((endpoint->bmAttributes & 3) != 3)
        *
        * Both the above tasks can be done using a single function
        * usb_endpoint_is_int_in().
        */
         endpoint = &interface->endpoint[0].desc;
         if (!usb_endpoint_is_int_in(endpoint))
                 return -ENODEV;

         /*
        * Get interrupt pipe. A pipe is combination of usb_device, interface
        * and endpoint.
        */
         pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
         /*
        * Get the max packet size that we can deal with this endpoint. returns
        * wMaxPacketSize field of endpoint interface descriptor.
        */
         maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
```

The above code is self explanatory as I have put a lot of comments in the code. Next step is to allocate memory for our private data structure.

```
       /*
        * Allocate memory for our private structure and initialize with zero.
        */
         kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
```

Then allocate the memory for *struct input_dev* object. This is the object we need to pass to the Linux Input subsytem when we register with input subsystem.

```
           /*
          * Allocate struct input_dev object, this object will be given
          * to the input subsystem
          */
         input_dev = input_allocate_device();
         if (!kbd || !input_dev)
                 goto fail1;
```

Allocate remaining resources like URBs. URBs are the data structure in which we pass the request to the USB devices. We will fill in the URB and submit it to the USB Core.

```
/*
        * Allocate remaining resources like urbs
        */
         if (usb_kbd_alloc_mem(dev, kbd))
                 goto fail2;
       /*
        * Store ref to usb_dev and input_dev in the private object
        */

         kbd->usbdev = dev;
         kbd->dev = input_dev;
```

usb_kbd_alloc_mem() function looks like this.

```
/*
 * usb_kbd_alloc_mem: Allocate resources like urb and buffers
 */
static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
{
       /*
        * Allocate URB for Interrupt IN
        */
         if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
                 return -1;
          /*
        * Allcate buffer to store read data
        */
         if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd-
>new_dma)))
                 return -1;

         return 0;
}
```

This function will allocate memory for URB and memory to store the received data from the keyboard.

Next thing in the code will be to get the keyboard manufacturer name, product name etc.

```
/*Get the keyboard manufacturer id and product id*/
       if (dev->manufacturer)
              strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));

       if (dev->product) {
              if (dev->manufacturer)
                     strlcat(kbd->name, " ", sizeof(kbd->name));
              strlcat(kbd->name, dev->product, sizeof(kbd->name));
       }

       if (!strlen(kbd->name))
              snprintf(kbd->name, sizeof(kbd->name),
                     "USB HIDBP Keyboard %04x:%04x",
                     le16_to_cpu(dev->descriptor.idVendor),
                     le16_to_cpu(dev->descriptor.idProduct));
```

Now fill in the input_dev object with the proper information.

```
      strlcat(kbd->phys, "/input0", sizeof(kbd->phys));

       /*Set the input device name, phyp, id parent device*/
       input_dev->name = kbd->name;
       input_dev->phys = kbd->phys;
       usb_to_input_id(dev, &input_dev->id);
       input_dev->dev.parent = &iface->dev;
       /*store out private data pointer in device object*/
       input_set_drvdata(input_dev, kbd);
```

Tell the input subsytem that we handle keys and we need the key repeat mechanism by the input subsystem. And tell which keys we will handle.

```
/* Tell input subsystem our capabilities*/
      input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);

/*Tell the input subsystem the keys that we handle*/
      for (i = 0; i < 255; i++)
            set_bit(usb_kbd_keycode[i], input_dev->keybit);
      clear_bit(0, inpu /*Tell the input subsystem the keys that we handle*/
      for (i = 0; i < 255; i++)
            set_bit(usb_kbd_keycode[i], input_dev->keybit);
      clear_bit(0, input_dev->keybit);t_dev->keybit);
```

In the above code we have used an array of keys usb_kbd_keycode. This table is a map of scan code (here we should call USB keycode) to the Linux kernel keys. Linux kernel keys are the generic keycodes used by the input subsytem. These are the keycodes passed up to the applications. Please see the <include/linux/input.h> for all kernel keycodes definitions.

Fill in our open and read call backs in the input_dev object. Open call back will be called by the input subsystem whenever there is a event generation request from the input event handlers. And close will be called by input subsystem when input handlers want to stop receive events from our input device.

```
/*Tell our open and close entry points from the input subsystem*/
      input_dev->open = usb_kbd_open;
      input_dev->close = usb_kbd_close;
```

Fill in the URB object. This URB object will be submitted to the USB core. USB core passes the request to the keyboard device through the USB Controller. Keyboard will respond with appropriate data for the request.

```
/*
 * Fill in the Interrupt URB. Submit it to the usb core in the
 * usb_kbd_open() and interrupt call back usb_kbd_irq(). In URB
 * context we store our private object so that we can access it in
 * the usb_kbd_irq().
 */
usb_fill_int_urb(kbd->irq, dev, pipe,
             kbd->new, (maxp > 8 ? 8 : maxp),
             usb_kbd_irq, kbd, endpoint->bInterval);
```

Fill in the URBs trasnfer_dma field with the DMA address that we got in the usb_kbd_alloc_mem().

```
/*
 * transfer_dma, where received data is kept
 */
kbd->irq->transfer_dma = kbd->new_dma;
kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

Finally register with the input subsystem.

```
/*Ok, we are done, tell it to the input subsystem*/
 error = input_register_device(kbd->dev);
 if (error)
       goto fail2;

 usb_set_intfdata(iface, kbd);

 /*tell the user we are ready*/
 printk(KERN_INFO "My USB Keyboard: %s is up", kbd->name);
 return 0;
```

That's it. We are done with the probe function of the driver. Major things that we have done in the probe function are:

1) Get Endpoint(Interrupt In) and Create pipe to talk to the USB keyboard.

2) Allocate the required memory for URBs
3) Allocate memory for struct input_device object.
4) Fill in the input_dev object with the required information.
5) Fill in the URB
6) Register with the input subsystem.

Driver Open Function

Driver open function is usb_kbd_open(). This call back will be called by the input subsystem when the input event handlers want to receive events from this input device. The function looks like this.

```
/*
 * Set interrupt URB device and submit the urb. Once the request is
 *  data our usb_kbd_irq() call back function is called.
 */
static int usb_kbd_open(struct input_dev *dev)
{
        struct usb_kbd *kbd = input_get_drvdata(dev);

        kbd->irq->dev = kbd->usbdev;
        if (usb_submit_urb(kbd->irq, GFP_KERNEL))
                return -EIO;

        return 0;
}
```

In this function we will submit the URB request to the device to receive the keycodes. This URB will be passed to the USB Core using the usb_submit_urb() function, USB Core intern passes the request to the USB Host controller driver, which will submit the request to the device through the USB host controller. Keyboard will respond to the request when it gets the data(i.e when a key is pressed) . Then the data is passed to the USB Core through the USB Host controller driver. USB Core calls our URB status call back. Note that we have passed the URB status call back in the usb_fill_int_urb(), like this.

```
/*
         * Fill in the Interrupt URB. Submit it to the usb core in the
         * usb_kbd_open() and interrupt call back usb_kbd_irq(). In URB
         * context we store our private object so that we can access it in
         * the usb_kbd_irq().
         */
        usb_fill_int_urb(kbd->irq, dev, pipe,
                        kbd->new, (maxp > 8 ? 8 : maxp),
                        usb_kbd_irq, kbd, endpoint->bInterval);
```

Here, **usb_kbd_irq()** is the URB status call back function.

URB Status Call Back

As we have seen above, URB status call back function is the usb_kbd_irq(). This is the very important function of this driver. This is where we receive the keycodes from USB keyboard and this is where we pass the keycodes to the Input subsytems up to the applications. Important tasks of this function will be:

1) Get the Key Scancode(USB keycode)
2) Convert it to the kernel keycode
3) Pass it to the input subsystem using input_report_key() function
4) Call input_sync() function to push the key events up to the input event handlers from the input subsystem.

Let's here see all above steps with code.

A few lines of the code is to handle the errors from the device.

```
        /*
         * Get our private object, stored in URB context with
         * usb_fill_int_urb() in usb_kbd_probe().
         */
        struct usb_kbd *kbd = urb->context;
        int i;

        switch (urb->status) {
        case 0:                 /* success */
```

```
            break;
        case -ECONNRESET:       /* unlink */
        case -ENOENT:
        case -ESHUTDOWN:
            return;
        /* -EPIPE:  should clear the halt */
        default:                /* error */
            goto resubmit;
        }
```

When the keyboard sends data to the device driver, that will be stored in the buffer, **new,** that we have allocated in the probe function. This buffer is of 8 bytes of size. So, the USB keyboard returns 8 bytes of data for each request. This is called INPUT report. Let's see the format of this 8 bytes of data. As the report descriptor defines, the INPUT report is formatted as follows;

one byte of modifier byte and 6 bytes keycode array, like this.

byte	
0	Modifier byte
1	reserved
2	keycode array (0)
3	keycode array (1)
4	keycode array (2)
5	keycode array (3)
6	keycode array (4)
7	keycode array (5)

The bitmap of modifier byte is defined like this. These are the special keys information.

Bit	
0	LEFT CTRL
1	LEFT SHIFT
2	LEFT ALT
3	LEFT GUI
4	RIGHT CTRL

5	RIGHT SHIFT
6	RIGHT ALT
7	RIGHT GUI

And the second byte(1st byte) is reserved. Remaining 6 bytes contains keycodes for the keys pressed. So, the keycode array can hold up to 6 keys which are simultaneously pushed. The order of keycode in the array is arbitrary.

Now, first send the modifier byte status to the input subsystem. As there are 8 special keys(modifier keys), we need to call input_report_key() eight times. Like this.

```
/*First send the modifier byte status to input subsystem*/
        for (i = 0; i < 8; i++)
                input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >>
i) & 1);
```

Correspoding kernel keycodes for these special keys are, 29, 42, 56,125, 97, 54,100, 126 respectively. They are in the usb_kbd_keycode[] starting from array index 224, like this.

```
/*
 * Scancode to kernel keycode table, see linux/input.h
 * for all Linux keycodes
 */
static const unsigned char usb_kbd_keycode[256] = {
          0,  0,  0,  0, 30, 48, 46, 32, 18, 33, 34, 35, 23, 36, 37, 38,
         50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
          4,  5,  6,  7,  8,  9, 10, 11, 28,  1, 14, 15, 57, 12, 13, 26,
         27, 43, 43, 39, 40, 41, 51, 52, 53, 58, 59, 60, 61, 62, 63, 64,
         65, 66, 67, 68, 87, 88, 99, 70,119,110,102,104,111,107,109,106,
        105,108,103, 69, 98, 55, 74, 78, 96, 79, 80, 81, 75, 76, 77, 71,
         72, 73, 82, 83, 86,127,116,117,183,184,185,186,187,188,189,190,
        191,192,193,194,134,138,130,132,128,129,131,137,133,135,136,113,
        115,114,  0,  0,  0,121,  0, 89, 93,124, 92, 94, 95,  0,  0,  0,
        122,123, 90, 91, 85,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         29, 42, 56,125, 97, 54,100,126,164,166,165,163,161,115,114,113,
        150,158,159,128,136,177,178,176,142,152,173,140
};
```

The bold numbers are the correspoding kernel keycodes for the above special keys.

First parameter to the input_report_key() is pointer to the struct input_dev object, second one is the kernel keycode correspoding to the key and third parameter is wether key is pressed or not. We will check bit of the modifier byte and pass one(key pressed) or zero(key not pressed).

Next is to send the keys information to the input subsytem. As we see above there can be 6 keys pressed simultaneously. So, we need to pass these all 6 keys to the input subsytem. After we pass these keys to the input subsystem, we will store these keycodes in an another array **old**. Next time when you get the keys again we need to check the new keys with the old keys. If old keys are not there in the new keys old keys have been released. We need now to compare if any of the new key is not there in the old key, if not, tell the input subsytem that correspoding key has been released. Like this.

```
/* Now send up the keycode*/
        for (i = 2; i < 8; i++) {
                /*
                 * Check if the old key is in the new keys list, if not, send key
                 * release signal to input subsystem
                 */
                if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) == kbd-
>new + 8) {
                        if (usb_kbd_keycode[kbd->old[i]])
                                input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
                }
```

In the above code, memscan() will scan in the given memory for a character. If the character is found in the given momory, memscan will return the address of that character else it returns the address of the size of the memory plus 1, that means out of boundary address. We are passing the memory of the 6 bytes new array and sending the one old key that is to be searched in the given new array. If the givne old key is not in the new key array we will pass the key released event to the input subsystem.

If the new keys are not in the old keys, pass the key pressed event to the input subsystem, Like this.

```
            /*
             * Check if the new key is in the old keys list, if not send key pressed
             * signal to the input subsystem
             */
            if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) == kbd-
>old + 8) {
                  if (usb_kbd_keycode[kbd->new[i]])
                        input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
            }
```

Now ask the input subsystem to push the keys up to the input event handlers so that to the applications. And copy the new keys array into old keys array.

```
      /* Tell input subsystem to push up the keys*/
       input_sync(kbd->dev);

       /*backup current key info*/
       memcpy(kbd->old, kbd->new, 8);
```

Finally resubmit the URB request so that we will get next keys from the keyboard.

```
      /*Now resubmit the URB */
      i = usb_submit_urb (urb, GFP_ATOMIC);
      if (i)
            printk("can't resubmit intr, %s-%s/input0, status %d",
                  kbd->usbdev->bus->bus_name,
                  kbd->usbdev->devpath, i);
```

Close Function

usb_kbd_close() is the drivers close function. This function will be called by input subsystem when input event handlers want to stop receive events from our input device. Here we will just kill already submitted URB.

```
static void usb_kbd_close(struct input_dev *dev)
{
        struct usb_kbd *kbd = input_get_drvdata(dev);

        usb_kill_urb(kbd->irq);
}
```

USB Disconnect Function

usb_kbd_disconnect() is the USB disconnect function of the driver. This function will be called by the USB Core when our keyboard device has been unplugged. This will also be called when our driver deregister with the USB Core. We will just free all resources that have been allocated in the USB Probe function.

```
/*
 * usb_kbd_disconnect: Called by USB Core when our keyboard is unplugged or
 * our module is removed.
 */
static void usb_kbd_disconnect(struct usb_interface *intf)
{
        /*
         * Get our driver private structure stored in interface object using
         * usb_set_intfdata() in usb_kbd_probe().
         */
        struct usb_kbd *kbd = usb_get_intfdata (intf);

        usb_set_intfdata(intf, NULL);
        /*
         * If our usb_kbd_probe() probe function had been called, we need
         * to free the resources allocated.
         *
         */
        if (kbd) {
                /*Kill the urb if already submitted*/
                usb_kill_urb(kbd->irq);
                /*Unregister from the input subsystem*/
                input_unregister_device(kbd->dev);
                /*Free the memory we have allocated*/
                usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
                /*Free our private data*/
                kfree(kbd);
        }
}
```

Above code has sufficient commnets to understand what we do here.

Driver Exit Function

usb_kbd_exit() is the driver exit function. This is the module exit function which will be called when our driver module is removed. We just need to deregister with the USB Core.

```
static void __exit usb_kbd_exit(void)
{
        usb_deregister(&usb_kbd_driver);
}

module_init(usb_kbd_init);
module_exit(usb_kbd_exit);
```