Text Compare
Produced: 12/7/2015 6:07:15 PM

Mode: All
Left file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkdb\usbkbd.c
Right file: C:\Users\J14688\git\USBKeyboardDriverLinux\usbkbd-scancodemod\usbkbd-scancodemod.c

```
 1   1 /*
 2   2  *  Copyright (c) 1999-2001 Vojtech Pavlik
 3   3  *
 4   4  *  USB HIDBP Keyboard support
 5   5  */
 6   6
 7   7 /*
 8   8  * This program is free software; you can redistribute it and/or modify
 9   9  * it under the terms of the GNU General Public License as published by
10  10  * the Free Software Foundation; either version 2 of the License, or
11  11  * (at your option) any later version.
12  12  *
13  13  * This program is distributed in the hope that it will be useful,
14  14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  16  * GNU General Public License for more details.
17  17  *
18  18  * You should have received a copy of the GNU General Public License
19  19  * along with this program; if not, write to the Free Software
20  20  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
21  21  *
22  22  * Should you need to contact me, the author, you can do so either by
23  23  * e-mail - mail your message to <vojtech@ucw.cz>, or by paper mail:
24  24  * Vojtech Pavlik, Simunkova 1594, Prague 8, 182 00 Czech Republic
25  25  */
26  26
27  27 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
28  28
29  29 #include <linux/kernel.h>
30  30 #include <linux/slab.h>
31  31 #include <linux/module.h>
32  32 #include <linux/init.h>
33  33 #include <linux/usb/input.h>
34  34 #include <linux/hid.h>
35  35
36  36 /*
37  37  * Version Information
38  38  */
39  39 #define DRIVER_VERSION ""
40  40 #define DRIVER_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"
41  41 #define DRIVER_DESC "USB HID Boot Protocol keyboard driver"
42  42 #define DRIVER_LICENSE "GPL"
43  43
44  44 MODULE_AUTHOR(DRIVER_AUTHOR);
45  45 MODULE_DESCRIPTION(DRIVER_DESC);
46  46 MODULE_LICENSE(DRIVER_LICENSE);
47  47
    48 /*
    49  * Group7
    50  *
    51  * Swapped the position of elements in the 2nd row.  '50' and '31' have been swapped.
    52  *
    53  * Expectation: USB Keyboard physical 'm' key will become 's' and vice versa.
    54  *
    55  */
    56
48  57 static const unsigned char usb_kbd_keycode[256] = {
    58 //    0,  0,  0,  0, 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
49  59       0,  0,  0,  0, 30, 48, 46, 32, 18, 33, 34, 35, 23, 36, 37, 38,
50      50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
    60 //   50, 49, 24, 25, 16, 19, 31, 20, 22, 47, 17, 45, 21, 44,  2,  3,
    61      31, 49, 24, 25, 16, 19, 50, 20, 22, 47, 17, 45, 21, 44,  2,  3,
51  62       4,  5,  6,  7,  8,  9, 10, 11, 28,  1, 14, 15, 57, 12, 13, 26,
52  63      27, 43, 43, 39, 40, 41, 51, 52, 53, 58, 59, 60, 61, 62, 63, 64,
53  64      65, 66, 67, 68, 87, 88, 99, 70,119,110,102,104,111,107,109,106,
54  65     105,108,103, 69, 98, 55, 74, 78, 96, 79, 80, 81, 75, 76, 77, 71,
55  66      72, 73, 82, 83, 86,127,116,117,183,184,185,186,187,188,189,190,
56  67     191,192,193,194,134,138,130,132,128,129,131,137,133,135,136,113,
57  68     115,114,  0,  0,  0,121,  0, 89, 93,124, 92, 94, 95,  0,  0,  0,
58  69     122,123, 90, 91, 85,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
59  70       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
60  71       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
61  72       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
62  73       0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
63  74      29, 42, 56,125, 97, 54,100,126,164,166,165,163,161,115,114,113,
64  75     150,158,159,128,136,177,178,176,142,152,173,140
65  76 };
66  77
67  78
68  79 /**
69  80  * struct usb_kbd - state of each attached keyboard
```

```
 * @dev:    input device associated with this keyboard
 * @usbdev: usb device associated with this keyboard
 * @old:    data received in the past from the @irq URB representing which
 *      keys were pressed. By comparing with the current list of keys
 *      that are pressed, we are able to see key releases.
 * @irq:    URB for receiving a list of keys that are pressed when a
 *      new key is pressed or a key that was pressed is released.
 * @led:    URB for sending LEDs (e.g. numlock, ...)
 * @newleds:    data that will be sent with the @led URB representing which LEDs
 *      should be on
 * @name:   Name of the keyboard. @dev's name field points to this buffer
 * @phys:   Physical path of the keyboard. @dev's phys field points to this
 *      buffer
 * @new:    Buffer for the @irq URB
 * @cr:     Control request for @led URB
 * @leds:   Buffer for the @led URB
 * @new_dma:    DMA address for @irq URB
 * @leds_dma:   DMA address for @led URB
 * @leds_lock:  spinlock that protects @leds, @newleds, and @led_urb_submitted
 * @led_urb_submitted: indicates whether @led is in progress, i.e. it has been
 *      submitted and its completion handler has not returned yet
 *      without resubmitting @led
 */
struct usb_kbd {
    struct input_dev *dev;
    struct usb_device *usbdev;
    unsigned char old[8];
    struct urb *irq, *led;
    unsigned char newleds;
    char name[128];
    char phys[64];

    unsigned char *new;
    struct usb_ctrlrequest *cr;
    unsigned char *leds;
    dma_addr_t new_dma;
    dma_addr_t leds_dma;

    spinlock_t leds_lock;
    bool led_urb_submitted;

};

static void usb_kbd_irq(struct urb *urb)
{
    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;
    }

    for (i = 0; i < 8; i++)
        input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >> i) & 1);

    for (i = 2; i < 8; i++) {

        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);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) released.\n",
                     kbd->old[i]);
        }

        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);
            else
                hid_info(urb->dev,
                     "Unknown key (scancode %#x) pressed.\n",
                     kbd->new[i]);
        }
    }

    input_sync(kbd->dev);

    memcpy(kbd->old, kbd->new, 8);

resubmit:
```

```
    i = usb_submit_urb (urb, GFP_ATOMIC);
    if (i)
        hid_err(urb->dev, "can't resubmit intr, %s-%s/input0, status %d",
            kbd->usbdev->bus->bus_name,
            kbd->usbdev->devpath, i);
}

static int usb_kbd_event(struct input_dev *dev, unsigned int type,
             unsigned int code, int value)
{
    unsigned long flags;
    struct usb_kbd *kbd = input_get_drvdata(dev);

    if (type != EV_LED)
        return -1;

    spin_lock_irqsave(&kbd->leds_lock, flags);
    kbd->newleds = (!!test_bit(LED_KANA,    dev->led) << 3) | (!!test_bit(LED_COMPOSE, dev->led) << 3) |
               (!!test_bit(LED_SCROLLL, dev->led) << 2) | (!!test_bit(LED_CAPSL,   dev->led) << 1) |
               (!!test_bit(LED_NUML,    dev->led));

    if (kbd->led_urb_submitted){
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return 0;
    }

    if (*(kbd->leds) == kbd->newleds){
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return 0;
    }

    *(kbd->leds) = kbd->newleds;

    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))
        pr_err("usb_submit_urb(leds) failed\n");
    else
        kbd->led_urb_submitted = true;

    spin_unlock_irqrestore(&kbd->leds_lock, flags);

    return 0;
}

static void usb_kbd_led(struct urb *urb)
{
    unsigned long flags;
    struct usb_kbd *kbd = urb->context;

    if (urb->status)
        hid_warn(urb->dev, "led urb status %d received\n",
             urb->status);

    spin_lock_irqsave(&kbd->leds_lock, flags);

    if (*(kbd->leds) == kbd->newleds){
        kbd->led_urb_submitted = false;
        spin_unlock_irqrestore(&kbd->leds_lock, flags);
        return;
    }

    *(kbd->leds) = kbd->newleds;

    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC)){
        hid_err(urb->dev, "usb_submit_urb(leds) failed\n");
        kbd->led_urb_submitted = false;
    }
    spin_unlock_irqrestore(&kbd->leds_lock, flags);

}

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;
}

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

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

```
248 259
249 260 static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
250 261 {
251 262     if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
252 263         return -1;
253 264     if (!(kbd->led = usb_alloc_urb(0, GFP_KERNEL)))
254 265         return -1;
255 266     if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd->new_dma)))
256 267         return -1;
257 268     if (!(kbd->cr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL)))
258 269         return -1;
259 270     if (!(kbd->leds = usb_alloc_coherent(dev, 1, GFP_ATOMIC, &kbd->leds_dma)))
260 271         return -1;
261 272
262 273     return 0;
263 274 }
264 275
265 276 static void usb_kbd_free_mem(struct usb_device *dev, struct usb_kbd *kbd)
266 277 {
267 278     usb_free_urb(kbd->irq);
268 279     usb_free_urb(kbd->led);
269 280     usb_free_coherent(dev, 8, kbd->new, kbd->new_dma);
270 281     kfree(kbd->cr);
271 282     usb_free_coherent(dev, 1, kbd->leds, kbd->leds_dma);
272 283 }
273 284
274 285 static int usb_kbd_probe(struct usb_interface *iface,
275 286                          const struct usb_device_id *id)
276 287 {
277 288     struct usb_device *dev = interface_to_usbdev(iface);
278 289     struct usb_host_interface *interface;
279 290     struct usb_endpoint_descriptor *endpoint;
280 291     struct usb_kbd *kbd;
281 292     struct input_dev *input_dev;
282 293     int i, pipe, maxp;
283 294     int error = -ENOMEM;
284 295
285 296     interface = iface->cur_altsetting;
286 297
287 298     if (interface->desc.bNumEndpoints != 1)
288 299         return -ENODEV;
289 300
290 301     endpoint = &interface->endpoint[0].desc;
291 302     if (!usb_endpoint_is_int_in(endpoint))
292 303         return -ENODEV;
293 304
294 305     pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
295 306     maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
296 307
297 308     kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
298 309     input_dev = input_allocate_device();
299 310     if (!kbd || !input_dev)
300 311         goto fail1;
301 312
302 313     if (usb_kbd_alloc_mem(dev, kbd))
303 314         goto fail2;
304 315
305 316     kbd->usbdev = dev;
306 317     kbd->dev = input_dev;
307 318     spin_lock_init(&kbd->leds_lock);
308 319
309 320     if (dev->manufacturer)
310 321         strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));
311 322
312 323     if (dev->product) {
313 324         if (dev->manufacturer)
314 325             strlcat(kbd->name, " ", sizeof(kbd->name));
315 326         strlcat(kbd->name, dev->product, sizeof(kbd->name));
316 327     }
317 328
318 329     if (!strlen(kbd->name))
319 330         snprintf(kbd->name, sizeof(kbd->name),
320 331              "USB HIDBP Keyboard %04x:%04x",
321 332              le16_to_cpu(dev->descriptor.idVendor),
322 333              le16_to_cpu(dev->descriptor.idProduct));
323 334
324 335     usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
325 336     strlcat(kbd->phys, "/input0", sizeof(kbd->phys));
326 337
327 338     input_dev->name = kbd->name;
328 339     input_dev->phys = kbd->phys;
329 340     usb_to_input_id(dev, &input_dev->id);
330 341     input_dev->dev.parent = &iface->dev;
331 342
332 343     input_set_drvdata(input_dev, kbd);
333 344
334 345     input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
335 346         BIT_MASK(EV_REP);
```

```
    input_dev->ledbit[0] = BIT_MASK(LED_NUML) | BIT_MASK(LED_CAPSL) |
        BIT_MASK(LED_SCROLLL) | BIT_MASK(LED_COMPOSE) |
        BIT_MASK(LED_KANA);

    for (i = 0; i < 255; i++)
        set_bit(usb_kbd_keycode[i], input_dev->keybit);
    clear_bit(0, input_dev->keybit);

    input_dev->event = usb_kbd_event;
    input_dev->open = usb_kbd_open;
    input_dev->close = usb_kbd_close;

    usb_fill_int_urb(kbd->irq, dev, pipe,
             kbd->new, (maxp > 8 ? 8 : maxp),
             usb_kbd_irq, kbd, endpoint->bInterval);
    kbd->irq->transfer_dma = kbd->new_dma;
    kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
    kbd->cr->bRequest = 0x09;
    kbd->cr->wValue = cpu_to_le16(0x200);
    kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
    kbd->cr->wLength = cpu_to_le16(1);

    usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
                (void *) kbd->cr, kbd->leds, 1,
                usb_kbd_led, kbd);
    kbd->led->transfer_dma = kbd->leds_dma;
    kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    error = input_register_device(kbd->dev);
    if (error)
        goto fail2;

    usb_set_intfdata(iface, kbd);
    device_set_wakeup_enable(&dev->dev, 1);
    return 0;

fail2:
    usb_kbd_free_mem(dev, kbd);
fail1:
    input_free_device(input_dev);
    kfree(kbd);
    return error;
}

static void usb_kbd_disconnect(struct usb_interface *intf)
{
    struct usb_kbd *kbd = usb_get_intfdata (intf);

    usb_set_intfdata(intf, NULL);
    if (kbd) {
        usb_kill_urb(kbd->irq);
        input_unregister_device(kbd->dev);
        usb_kill_urb(kbd->led);
        usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
        kfree(kbd);
    }
}

static struct usb_device_id usb_kbd_id_table [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
        USB_INTERFACE_PROTOCOL_KEYBOARD) },
    { }                     /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);

static struct usb_driver usb_kbd_driver = {
    .name =     "usbkbd",
    .probe =    usb_kbd_probe,
    .disconnect =   usb_kbd_disconnect,
    .id_table = usb_kbd_id_table,
};

module_usb_driver(usb_kbd_driver);
```