Text Compare
Produced: 12/6/2015 9:04:18 PM

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

```
/*
 *  Copyright (c) 1999-2001 Vojtech Pavlik
 *
 *  USB HIDBP Keyboard support
 */

/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Should you need to contact me, the author, you can do so either by
 * e-mail - mail your message to <vojtech@ucw.cz>, or by paper mail:
 * Vojtech Pavlik, Simunkova 1594, Prague 8, 182 00 Czech Republic
 */

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/usb/input.h>
#include <linux/hid.h>

/*
 * Version Information
 */
#define DRIVER_VERSION ""
#define DRIVER_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"
#define DRIVER_DESC "USB HID Boot Protocol keyboard driver"
#define DRIVER_LICENSE "GPL"

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE(DRIVER_LICENSE);

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
};


/**
 * 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)
{
    printk(KERN_INFO "Group7 usbkbd event");
    unsigned long flags;
```

```
170 171     struct usb_kbd *kbd = input_get_drvdata(dev);
171 172
172 173     if (type != EV_LED)
173 174         return -1;
174 175
175 176     spin_lock_irqsave(&kbd->leds_lock, flags);
176 177     kbd->newleds = (!!test_bit(LED_KANA,    dev->led) << 3) | (!!test_bit(LED_COMPOSE, dev->led) << 3) |
177 178                (!!test_bit(LED_SCROLLL, dev->led) << 2) | (!!test_bit(LED_CAPSL,   dev->led) << 1) |
178 179                (!!test_bit(LED_NUML,    dev->led));
179 180
180 181     if (kbd->led_urb_submitted){
181 182         spin_unlock_irqrestore(&kbd->leds_lock, flags);
182 183         return 0;
183 184     }
184 185
185 186     if (*(kbd->leds) == kbd->newleds){
186 187         spin_unlock_irqrestore(&kbd->leds_lock, flags);
187 188         return 0;
188 189     }
189 190
190 191     *(kbd->leds) = kbd->newleds;
191 192
192 193     kbd->led->dev = kbd->usbdev;
193 194     if (usb_submit_urb(kbd->led, GFP_ATOMIC))
194 195         pr_err("usb_submit_urb(leds) failed\n");
195 196     else
196 197         kbd->led_urb_submitted = true;
197 198
198 199     spin_unlock_irqrestore(&kbd->leds_lock, flags);
199 200
200 201     return 0;
201 202 }
202 203
203 204 static void usb_kbd_led(struct urb *urb)
204 205 {
205 206     unsigned long flags;
206 207     struct usb_kbd *kbd = urb->context;
207 208
208 209     if (urb->status)
209 210         hid_warn(urb->dev, "led urb status %d received\n",
210 211              urb->status);
211 212
212 213     spin_lock_irqsave(&kbd->leds_lock, flags);
213 214
214 215     if (*(kbd->leds) == kbd->newleds){
215 216         kbd->led_urb_submitted = false;
216 217         spin_unlock_irqrestore(&kbd->leds_lock, flags);
217 218         return;
218 219     }
219 220
220 221     *(kbd->leds) = kbd->newleds;
221 222
222 223     kbd->led->dev = kbd->usbdev;
223 224     if (usb_submit_urb(kbd->led, GFP_ATOMIC)){
224 225         hid_err(urb->dev, "usb_submit_urb(leds) failed\n");
225 226         kbd->led_urb_submitted = false;
226 227     }
227 228     spin_unlock_irqrestore(&kbd->leds_lock, flags);
228 229
229 230 }
230 231
231 232 static int usb_kbd_open(struct input_dev *dev)
232 233 {
    234     printk(KERN_INFO "Group7 usbkbd open");
233 235     struct usb_kbd *kbd = input_get_drvdata(dev);
234 236
235 237     kbd->irq->dev = kbd->usbdev;
236 238     if (usb_submit_urb(kbd->irq, GFP_KERNEL))
237 239         return -EIO;
238 240
239 241     return 0;
240 242 }
241 243
242 244 static void usb_kbd_close(struct input_dev *dev)
243 245 {
    246     printk(KERN_INFO "Group7 usbkbd close");
244 247     struct usb_kbd *kbd = input_get_drvdata(dev);
245 248
246 249     usb_kill_urb(kbd->irq);
247 250 }
248 251
249 252 static int usb_kbd_alloc_mem(struct usb_device *dev, struct usb_kbd *kbd)
250 253 {
251 254     if (!(kbd->irq = usb_alloc_urb(0, GFP_KERNEL)))
252 255         return -1;
253 256     if (!(kbd->led = usb_alloc_urb(0, GFP_KERNEL)))
254 257         return -1;
255 258     if (!(kbd->new = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &kbd->new_dma)))
```

```
256 259         return -1;
257 260     if (!(kbd->cr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_KERNEL)))
258 261         return -1;
259 262     if (!(kbd->leds = usb_alloc_coherent(dev, 1, GFP_ATOMIC, &kbd->leds_dma)))
260 263         return -1;
261 264
262 265     return 0;
263 266 }
264 267
265 268 static void usb_kbd_free_mem(struct usb_device *dev, struct usb_kbd *kbd)
266 269 {
267 270     usb_free_urb(kbd->irq);
268 271     usb_free_urb(kbd->led);
269 272     usb_free_coherent(dev, 8, kbd->new, kbd->new_dma);
270 273     kfree(kbd->cr);
271 274     usb_free_coherent(dev, 1, kbd->leds, kbd->leds_dma);
272 275 }
273 276
274 277 static int usb_kbd_probe(struct usb_interface *iface,
275 278             const struct usb_device_id *id)
276 279 {
    280     printk(KERN_INFO "Group7 usbkbd probe");
277 281     struct usb_device *dev = interface_to_usbdev(iface);
278 282     struct usb_host_interface *interface;
279 283     struct usb_endpoint_descriptor *endpoint;
280 284     struct usb_kbd *kbd;
281 285     struct input_dev *input_dev;
282 286     int i, pipe, maxp;
283 287     int error = -ENOMEM;
284 288
285 289     interface = iface->cur_altsetting;
286 290
287 291     if (interface->desc.bNumEndpoints != 1)
288 292         return -ENODEV;
289 293
290 294     endpoint = &interface->endpoint[0].desc;
291 295     if (!usb_endpoint_is_int_in(endpoint))
292 296         return -ENODEV;
293 297
294 298     pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
295 299     maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
296 300
297 301     kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
298 302     input_dev = input_allocate_device();
299 303     if (!kbd || !input_dev)
300 304         goto fail1;
301 305
302 306     if (usb_kbd_alloc_mem(dev, kbd))
303 307         goto fail2;
304 308
305 309     kbd->usbdev = dev;
306 310     kbd->dev = input_dev;
307 311     spin_lock_init(&kbd->leds_lock);
308 312
309 313     if (dev->manufacturer)
310 314         strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));
311 315
312 316     if (dev->product) {
313 317         if (dev->manufacturer)
314 318             strlcat(kbd->name, " ", sizeof(kbd->name));
315 319         strlcat(kbd->name, dev->product, sizeof(kbd->name));
316 320     }
317 321
318 322     if (!strlen(kbd->name))
319 323         snprintf(kbd->name, sizeof(kbd->name),
320 324              "USB HIDBP Keyboard %04x:%04x",
321 325              le16_to_cpu(dev->descriptor.idVendor),
322 326              le16_to_cpu(dev->descriptor.idProduct));
323 327
324 328     usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
325 329     strlcat(kbd->phys, "/input0", sizeof(kbd->phys));
326 330
327 331     input_dev->name = kbd->name;
328 332     input_dev->phys = kbd->phys;
329 333     usb_to_input_id(dev, &input_dev->id);
330 334     input_dev->dev.parent = &iface->dev;
331 335
332 336     input_set_drvdata(input_dev, kbd);
333 337
334 338     input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
335 339         BIT_MASK(EV_REP);
336 340     input_dev->ledbit[0] = BIT_MASK(LED_NUML) | BIT_MASK(LED_CAPSL) |
337 341         BIT_MASK(LED_SCROLLL) | BIT_MASK(LED_COMPOSE) |
338 342         BIT_MASK(LED_KANA);
339 343
340 344     for (i = 0; i < 255; i++)
341 345         set_bit(usb_kbd_keycode[i], input_dev->keybit);
342 346     clear_bit(0, input_dev->keybit);
```

```
343 347
344 348     input_dev->event = usb_kbd_event;
345 349     input_dev->open = usb_kbd_open;
346 350     input_dev->close = usb_kbd_close;
347 351
348 352     usb_fill_int_urb(kbd->irq, dev, pipe,
349 353              kbd->new, (maxp > 8 ? 8 : maxp),
350 354              usb_kbd_irq, kbd, endpoint->bInterval);
351 355     kbd->irq->transfer_dma = kbd->new_dma;
352 356     kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
353 357
354 358     kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
355 359     kbd->cr->bRequest = 0x09;
356 360     kbd->cr->wValue = cpu_to_le16(0x200);
357 361     kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
358 362     kbd->cr->wLength = cpu_to_le16(1);
359 363
360 364     usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
361 365                  (void *) kbd->cr, kbd->leds, 1,
362 366                  usb_kbd_led, kbd);
363 367     kbd->led->transfer_dma = kbd->leds_dma;
364 368     kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
365 369
366 370     error = input_register_device(kbd->dev);
367 371     if (error)
368 372         goto fail2;
369 373
370 374     usb_set_intfdata(iface, kbd);
371 375     device_set_wakeup_enable(&dev->dev, 1);
372 376     return 0;
373 377
374 378 fail2:
375 379     usb_kbd_free_mem(dev, kbd);
376 380 fail1:
377 381     input_free_device(input_dev);
378 382     kfree(kbd);
379 383     return error;
380 384 }
381 385
382 386 static void usb_kbd_disconnect(struct usb_interface *intf)
383 387 {
    388     printk(KERN_INFO "Group7 usbkbd disconnect");
384 389     struct usb_kbd *kbd = usb_get_intfdata (intf);
385 390
386 391     usb_set_intfdata(intf, NULL);
387 392     if (kbd) {
388 393         usb_kill_urb(kbd->irq);
389 394         input_unregister_device(kbd->dev);
390 395         usb_kill_urb(kbd->led);
391 396         usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
392 397         kfree(kbd);
393 398     }
394 399 }
395 400
396 401 static struct usb_device_id usb_kbd_id_table [] = {
397 402     { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
398 403         USB_INTERFACE_PROTOCOL_KEYBOARD) },
399 404     { }                     /* Terminating entry */
400 405 };
401 406
402 407 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
403 408
404 409 static struct usb_driver usb_kbd_driver = {
405 410     .name =     "usbkbd",
406 411     .probe =    usb_kbd_probe,
407 412     .disconnect =   usb_kbd_disconnect,
408 413     .id_table = usb_kbd_id_table,
409 414 };
410 415
411 416 module_usb_driver(usb_kbd_driver);
```