```c
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/pci.h>
#include <linux/signal.h>
#include <linux/sched/signal.h>
#include <linux/uaccess.h>

#define AD_BASE                 0
#define AD_LOW_BYTE             AD_BASE
#define AD_HIGH_BYTE            AD_BASE+1
#define AD_RANGE                AD_BASE+2
#define AD_START_CHANNEL        AD_BASE+4
#define AD_STOP_CHANNEL         AD_BASE+5
#define AD_CONTROL              AD_BASE+6
#define AD_STATUS               AD_BASE+7
#define AD_CLEAR_INTERRUPT      AD_BASE+8
#define AD_CLEAR_FIFO           AD_BASE+9
#define DA0_LOW                 AD_BASE+10
#define DA0_HIGH                AD_BASE+11
#define DA_REF                  AD_BASE+14
#define DIO1                    AD_BASE+16
#define DIO2                    AD_BASE+17

#define COUNTER0                AD_BASE+24
#define COUNTER1                AD_BASE+26
#define COUNTER2                AD_BASE+28
#define COUNTER_CONTROL         AD_BASE+30

#define DA_BASE                 0xec00
#define DA_CHANNEL0_LOW         DA_BASE
#define DA_CHANNEL0_HIGH        DA_BASE+1
#define DA_RANGE                DA_BASE+8

/*
 * The name for our device, as it will appear in /proc/devices
 */
#define  DEVICE_NAME   "pci_adda"
#define  pci_adda_MAJOR 255

static int Major;
static int Device_Open = 0;
static int app_id=-1;
static int ad1_value;
static unsigned long pci_io_base=0, pci_io_length=0;
static long timer_value=100;

#define VENDOR_ID 0x13fe
#define DEVICE_ID 0x1711

static struct pci_device_id  pci_device_id_adda[] =
{
  {VENDOR_ID, DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
  {}  // end of list
};

void kill_proc(int pid, int sig)
{
    struct task_struct* p;
    struct task_struct* t = NULL;
    struct pid* pspid = NULL;

    rcu_read_lock();
    p = &init_task;
    do {
        if (p->pid == pid){
            t = p;
```

```
72                  break;
73              }
74              p=next_task(p);
75          } while(p != &init_task);
76          if(t != NULL) {
77              pspid = t->pids[PIDTYPE_PID].pid;
78              if (pspid != NULL) kill_pid(pspid,sig,1);
79          }
80          rcu_read_unlock();
81      }
82
83      static irqreturn_t pci_adda_handler(int irq, void *dev_id)
84      {
85          kill_proc(app_id,SIGUSR1);
86          ad1_value=inw(pci_io_base+AD_LOW_BYTE) & 0xfff;
87          outb(0, pci_io_base+AD_CLEAR_INTERRUPT); /* clears interrupt */
88          outb(0, pci_io_base+AD_CLEAR_FIFO); /* clears FIFO */
89          return IRQ_HANDLED;
90      }
91
92      void init_adda(void)
93      {
94          outb(1, pci_io_base+AD_START_CHANNEL);
95          outb(1, pci_io_base+AD_STOP_CHANNEL);
96          outb(0, pci_io_base+AD_RANGE); /* single-ended bipolar -10+10 */
97
98          outb(0x12, pci_io_base+AD_CONTROL); /* AD conversion mode : Timer Trigger,
            Interrupt */
99          //outb(0x01, pci_io_base+AD_CONTROL); /* AD conversion mode : Software Trigger,
            No Interrupt */
100         outb(0xff, DA_RANGE);
101         outb(0x5,pci_io_base+DA_REF);
102     }
103     void init_counter(void)
104     {
105         unsigned short count;
106         /* Initilize the timer  */
107         count=100;
108         outb(0x74, pci_io_base+COUNTER_CONTROL); /* 0111 0100 */
109         outb((count & 0xff), pci_io_base+COUNTER1);
110         outb(((count >> 8) & 0xff), pci_io_base+COUNTER1);
111         count=(unsigned short)timer_value;
112
113         outb(0xB4, pci_io_base+COUNTER_CONTROL); /* 1011 0100 */
114         outb((count & 0xff), pci_io_base+COUNTER2);
115         outb(((count >> 8) & 0xff), pci_io_base+COUNTER2);
116         /* count=100,count=50 : 2KHz for 10MHz clock input : PCI-1711 */
117         /* count=100,count=10 : 10KHz for 10MHz clock input : PCI-1711 */
118
119         outb(0, pci_io_base+AD_CLEAR_INTERRUPT); /* clears interrupt */
120         outb(0, pci_io_base+AD_CLEAR_FIFO); /* clears FIFO */
121     }
122
123     int device_probe(struct pci_dev *dev, const struct pci_device_id *id)
124     {
125         int ret;
126         int return_val;
127
128         ret = pci_enable_device(dev);
129         if (ret < 0) {
130             printk(KERN_WARNING "pci_adda: unable to initialize PCI device\n");
131             return ret;
132         }
133         if (dev->irq)
134             printk(KERN_ERR "pci_adda: IRQ %d.\n", dev->irq);
135         else
136             printk(KERN_ERR "pci_adda: No irq required/requested.\n");
137         pci_io_base = pci_resource_start( dev, 2 );
138         pci_io_length = pci_resource_len( dev, 2 );
139         if ((pci_io_length > 0) && ( NULL == request_region( pci_io_base, pci_io_length,
            dev->dev.kobj.name ) ) )
```

```
140         {
141             printk(KERN_ERR "I/O address conflict for device \"%s\",
                pci_io_base=%lu=0x%x, =%lu=0x%x\n",
142             dev->dev.kobj.name, pci_io_base, (unsigned int)pci_io_base, pci_io_length,
                (unsigned int)pci_io_length);
143             goto cleanup_irq;
144         }
145         if (pci_io_length)
146             printk(KERN_CRIT "pci_adda: request_region( pci_io_base=%lu=0x%x,
                pci_io_length=%lu=0x%x, dev->dev.kobj.name=%s) successfull.\n",
147             pci_io_base, (unsigned int)pci_io_base, pci_io_length, (unsigned
                int)pci_io_length, dev->dev.kobj.name);
148         else
149             printk(KERN_CRIT "pci_adda: request_region( pci_io_base=%lu=0x%x,
                pci_io_length=%lu=0x%x, dev->dev.kobj.name=%s) not necessary.\n",
150             pci_io_base, (unsigned int)pci_io_base, pci_io_length, (unsigned
                int)pci_io_length, dev->dev.kobj.name);
151
152         init_adda();init_counter();
153         irq_set_irq_type(dev->irq, IRQ_TYPE_LEVEL_LOW);
154         return_val = request_irq(dev->irq, &pci_adda_handler, /*IRQF_DISABLED
            |*/IRQF_TRIGGER_LOW, "pci_adda", dev);
155         if( return_val< 0 ) {
156             printk(KERN_ERR "pci_adda_init() : Can't reqeust irq %#010x\n", dev->irq);
157             return -1;
158         }
159         printk(KERN_INFO "pci_adda: device_probe successful\n");
160         return ret;
161
162         cleanup_irq:
163             if (dev->irq)
164                 free_irq( dev->irq, dev );
165             return (-EIO);
166     }
167
168     void device_remove(struct pci_dev *dev)
169     {
170         if (dev->irq)
171         free_irq( dev->irq, dev );
172
173         pci_release_regions(dev);
174         pci_disable_device(dev);
175         printk(KERN_INFO "pci_adda: device removed\n");
176     }
177
178     struct pci_driver  pci_driver_adda =
179     {
180         name: "pci_driver_adda",
181         id_table: pci_device_id_adda,
182         probe: device_probe,
183         remove: device_remove
184     };
185
186     int pci_adda_open (struct inode *inode, struct file *file)
187     {
188         if (Device_Open) {
189             return -EBUSY;
190         }
191         Device_Open++;
192         return 0;
193     }
194     int pci_adda_release (struct inode *inode, struct file *file)
195     {
196         Device_Open--;
197         return 0;
198     }
199     static long pci_adda_ioctl(/*struct inode *inode, */struct file *file,
200                     unsigned int cmd,unsigned long gdata)
201     {
202         unsigned long value;
203         if (copy_from_user(&value,(char *)gdata,4)) return -EFAULT;
```

```
204         switch (cmd) {
205            case 0:
206                 app_id = value;
207                 break;
208            case 1:
209                 timer_value = value;
210                 init_adda();
211                 init_counter();
212                 break;
213             default: break;
214         }
215         return 0;
216     }
217
218     ssize_t pci_adda_write (struct file *file,
219                     const char *buffer,  /* buffer */
220                     size_t length,       /* length of buffer */
221                     loff_t * offset)     /* offset in the file */
222     {
223         short value[3];
224         if (copy_from_user(value,buffer,6)) return -EFAULT;
225         outw(value[0], pci_io_base+DA0_LOW);
226         outw(value[2], pci_io_base+DIO1);
227         return (length);
228     }
229
230     ssize_t pci_adda_read (struct file *file,
231                     char *buffer,     /* buffer */
232                     size_t length,    /* length of buffer */
233                     loff_t * offset)  /* offset in the file */
234     {
235         short value[3];
236
237         value[0] = ad1_value;
238         value[2] = inw(pci_io_base+DIO1);
239         length = 6;
240         if (copy_to_user(buffer, value, length)) return -EFAULT;
241         return (length);
242     }
243
244     static struct file_operations pci_adda_fops =
245     {
246         open:           pci_adda_open,
247         write:          pci_adda_write,
248         read:           pci_adda_read,
249         release:        pci_adda_release,
250         unlocked_ioctl: pci_adda_ioctl,
251     };
252
253     static int __init pci_adda_init(void)
254     {
255         /* Register the character device */
256         Major = register_chrdev (pci_adda_MAJOR, DEVICE_NAME, &pci_adda_fops);
257         if (Major < 0) {
258             printk ("pci_adda init_module: failed with %d\n", pci_adda_MAJOR);
259             return Major;
260         }
261         printk ("pci_adda driver registred: major = %d\n", pci_adda_MAJOR);
262
263         if( 0 == pci_register_driver(&pci_driver_adda)) {
264             return 0;
265         }
266         else {
267             unregister_chrdev( pci_adda_MAJOR,DEVICE_NAME);
268             return (-EIO);
269         }
270     }
271
272     /*
273      * Cleanup - unregister the appropriate file from /proc
274      */
```

```
void __exit pci_adda_exit(void)
{
    int ret=0;

    pci_unregister_driver(&pci_driver_adda);
    unregister_chrdev (pci_adda_MAJOR, DEVICE_NAME);
    if (ret < 0) {
        printk ("unregister_chrdev: error %d\n", ret);
    }
    else {
        printk ("module clean up ok\n");
    }
}

module_init(pci_adda_init);
module_exit(pci_adda_exit);

MODULE_AUTHOR("Dong-Jin Lim");
MODULE_LICENSE("GPL");
```