

Drivers for exams

August 24, 2024

1 DRIVER K24

```
1 // Write a character device driver with the structure below and these specifications
2 // Each write needs to return bytes equal to a single sizeof(patient_data)
3 // Sequential reads need to return sequential measurements only from when open happened and on
4 // If for any reason sequential reads can't happen, then the second read should return EOF
5 // after an EOF, for a new stream of data, the process needs to close() and open() again
6 // if a process reads all measurements, it sleeps and waits for new measurements through the
   interrupt
7
8 struct medical_dev {
9     // DONE: Lock type ?
10    spinlock_t lock;
11
12    waitqueue_t wq;
13    uint128_t cnt; /* Initialised to zero and will never wrap */
14
15    #define CIRC_BUF_SIZE (1024 * sizeof(struct patient_data))
16    char circ_buffer[CIRC_BUF_SIZE];
17 } medical_dev;
18
19 void intr(void) {
20     struct medical_dev *dev = &medical_dev;
21     struct patient_data pd;
22
23     get_patient_data_from_hw(&pd); /* Get data from real device */
24
25     spin_lock_irq(&medical_dev->lock);
26     memcpy(&dev->circ_buffer[dev->cnt % CIRC_BUF_SIZE],
27           &pd, sizeof(struct patient_data));
28     dev->cnt += sizeof(struct patient_data);
29     spin_unlock_irq(&medical_dev->lock);
30
31     wake_up_interruptable(&dev->wq);
32 }
33
34 struct chrdev_state {
35     // DONE: Lock type ?
36     struct semaphore lock;
37
38     struct medical_dev *medical_dev;
39
40     char local_buf[sizeof(struct patient_data)];
41     uint128_t local_cnt; /* Suppose it will never wrap */
42 }
43
44 static int medical_chrdev_open(struct inode *inode, struct file *filp) {
45     struct chrdev_state *state;
46     struct medical_dev *dev = &medical_dev;
47
48     if ((ret = nonseekable_open(inode, filp)) < 0) {
49         kfree(state);
50         return -ENODEV;
51     }
52
53     state = kmalloc(sizeof(chrdev_state), GFP_KERNEL);
54     if (!state) {
55         return -ENOMEM;
56     }
```

```

57     state->local_cnt = dev->cnt;
58     state->medical_dev = dev;
59     filp->private_data = state;
60     sema_init(&state->lock,1);
61
62
63     return ret;
64 }
65
66 static ssize_t medical_chrdev_read(struct file *filp, char __user *usrbuf,
67     size_t cnt, loff_t *f_pos) {
68
69     struct chrdev_state *state;
70     struct medical_dev *dev;
71
72     state = filp->private_data;
73     dev = state->medical_dev;
74
75     uint32_t bytes_to_copy = sizeof(patient_data);
76
77     if (down_interruptible(&state->lock)) {
78         return -ERESTARTSYS;
79     }
80
81     // Do we need to fetch a new measurement
82     if (dev->cnt == state->local_cnt) {
83         if (wait_event_interruptable(dev->wq, dev->cnt > state->local_cnt) > 0) {
84             up(&state->lock);
85             return -ERESTARTSYS;
86         }
87     }
88
89     if (*f_pos == 0) {
90         if (dev->cnt - state->local_cnt >= CIRC_BUF_SIZE) {
91             f_pos = 0;
92             return 0;
93         }
94     }
95
96     // Copy data from circular buffer
97     memcpy(state->local_buf,
98         &dev->circ_buffer[state->local_cnt % CIRC_BUF_SIZE],
99         bytes_to_copy);
100
101     // Send the data to user
102     if (copy_to_user(usrbuf, state->local_buf, bytes_to_copy)) {
103         up(&state->lock);
104         return -EFAULT;
105     }
106
107     state->local_cnt += bytes_to_copy;
108     *f_pos += bytes_to_copy;
109
110     up(&state->lock);
111     return bytes_to_copy;
112 }

```

2 DRIVER K23

```

1 typedef struct {} wait_queue_head_t;
2 //             status             dca_request
3 //             FREE, RESERVED, FINISHED
4 typedef enum {
5     DCA_REQ_FREE,
6     DCA_REQ_RESERVED,
7     DCA_REQ_FINISHED,
8 } dca_status_t;
9
10 //             buffer             input             output
11 #define MAX_DATA_SZ (4 << 20)
12 typedef struct {
13     int input[MAX_DATA_SZ];

```

```

14     int result[MAX_DATA_SZ];
15     dca_status_t status;
16     wait_queue_head_t request_wq;
17 } dca_request_t;
18
19 #define MAX_REQS (64*sizeof(dca_request_t))
20 typedef struct {
21     spinlock_t lock;
22     wait_queue_head_t slots_wq;
23     dca_request_t reqs[MAX_REQS];
24 } dca_dev_t;
25
26 dca_dev_t dca_dev;
27
28 //                                state                request
29 void dca_req_set_status(dca_request_t *req, dca_status_t status) {
30     req->status = status;
31 }
32
33 dca_request_t find_free_request(dca_dev_t *cdev) {
34     int i;
35
36     for (i=0; i<MAX_REQS; i++) {
37         if (cdev->reqs[i].status == DCA_REQ_FREE) {
38             return cdev->reqs[i];
39         }
40     }
41     return NULL;
42 }
43
44 dca_request_t *dca_is_req_finished(dca_request_t *req) {
45     return (req->status == DCA_REQ_FINISHED);
46 }
47
48 // Implemented elsewhere
49 void dca_notify_device(dca_request_t *req);
50
51 void dca_intr(void) {
52     dca_dev_t *dev = &dca_dev;
53     int i;
54     // .. ? .. lock
55     spin_lock_irq(&dev->lock);
56     for (i=0; i<MAX_REQS; i++) {
57         if (dca_is_req_finished(&dca_dev->reqs[i])) {
58             wake_up_interruptable(&cdev->reqs[i].request_wq);
59             dca_req_set_status(&cdev->req[i], DCA_REQ_FREE);
60         }
61     }
62     spin_unlock_irq(&dev->lock);
63     // .. ? .. unlock
64 }
65
66 static int dca_chrdev_open(struct inode *inode, struct file *filp) {
67     int ret = 0;
68     dca_dev_t *dev = &dca_dev;
69
70     if((ret = nonseekable_open(inode, filp)) < 0) {
71         ret = -ENODEV;
72         goto out;
73     }
74
75     filp->private_data = dev;
76     return ret;
77 }
78
79 typedef struct {
80     uint8_t input[MAX_DATA_SZ];
81     uint8_t result[MAX_DATA_SZ];
82 } dca_user_request_t;
83
84 #define DCA_MAGIC 'D'
85 #define DCA_SUBMIT_REQ _IORW(DCA_MAGIC, 0, dca_user_request_t);
86
87 static long dca_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
88     dca_user_request_t __user *argp = (dca_user_request_t __user *) arg;
89     dca_dev_t *dev = filp->private_data;

```

```

90     int ret = -ENOTTY;
91     int retry = -ERESTARTSYS;
92     if(_IOC_TYPE(cmd) != DCA_MAGIC) return ret;
93
94     switch(cmd) {
95         case DCA_SUBMIT_REQ:
96
97             // slot device,
98
99             spin_lock_irq(&dev->lock);
100             dca_request_t req = find_free_request(&dev);
101             if (!req) {
102                 wait_event_interruptible(dev->slots_wq, find_free_request(&dev));
103                 req = find_free_request(&dev);
104             }
105
106             // user space request
107             if (copy_from_user(&req->input, &argp->input, sizeof(argp->input))) {
108                 return -EFAULT;
109             }
110
111             // status request reserved request
112             user dca_req_set_status(req, DCA_REQ_RESERVED);
113             spin_unlock_irq(&dev->lock);
114
115             //
116             // request finished
117             wait_event_interruptible(req->request_wq, req->status == DCA_REQ_FINISHED);
118             if (copy_to_user(argp->result, req->result, sizeof(req->result))) {
119                 return -EFAULT;
120             }
121             break;
122
123         default:
124             return -EINVAL;
125     }
126
127 }

```

3 DRIVER K22

```
1 // Create a device driver that computes sparse to dense vectors and the other way around
2 // user space connects to the device via cvec_ioctl
3 // statuses for each slot in the buffer of the device: FREE, OCCUPIED, PROCESSED
4 // the driver only processes buffer slots that are OCCUPIED and an interrupt happens when
5 // the processing of a buffer slot is done to change the status of the slot to PROCESSED
6 // if there is no FREE slot in the buffer, than the process sleeps until there is one
7 #define DENSE_TO_SPARSE 0
8 #define SPARSE_TO_DENSE 0
9
10 // =====
11 //                                     device
12 // =====
13 //
14 // static int __init cvec_device_init(void) {
15 //     int i;
16 //     printk(KERN_INFO "Initializing cvec device\n");
17 //
18 //     // Initialize the device lock
19 //     mutex_init(&cvec_dev.lock);
20 //
21 //     // Initialize the wait queue
22 //     init_waitqueue_head(&cvec_dev.wq);
23 //
24 //     // Initialize buffer elements
25 //     for (i = 0; i < BUF_LEN; i++) {
26 //         cvec_dev.buffer[i].cvdesc = NULL;
27 //         cvec_dev.buffer[i].status = FREE;
28 //         cvec_dev.buffer[i].conversion_mode = DENSE_TO_SPARSE; // Default conversion mode
29 //     }
30 //
31 //     return 0;
32 // }
33 //
34 // module_init(cvec_device_init);
35
36 // Assume these two are implemented
37 int get_free_slot(struct cvec_device *cvdev);
38 int get_processed_slot(struct cvec_device *cvdev);
39
40 struct cvec_state {
41     // TODO: what kind of lock here?
42     struct semaphore lock;
43     // TODO: done
44     int conversion_mode;
45 }
46
47 struct cvec_descriptor{
48     int len;
49     int *input;
50     int *output;
51 }
52
53 struct cvec_device {
54     #define BUF_LEN 1024
55     struct {
56         cvec_descriptor *cvdesc;
57         int conversion_mode;
58         int status;
59     } buffer[BUF_LEN];
60     // TODO: what kind of lock here?
61     spinlock_t lock;
62     // TODO: done
63     wait_queue_head_t wq;
64 } cvec_dev;
65
66 void open(struct inode *inode, struct file *filp) {
67     int ret = 0;
68     struct cvec_state *state;
69     struct cvec_device *cvdev = &cvec_dev;
70     if ((ret = nonseekable_open(inode, filp)) < 0) {
71         ret = -ENODEV;
72         goto out;
73     }
```

```

73     }
74     state = kmalloc(sizeof(struct cvec_state), GFP_KERNEL);
75     // TODO: structs
76     state->conversion_mode = DENSE_TO_SPARSE; // Default
77     filp->private_data = state;
78     sema_init(&cvec_state->lock,1);
79     // TODO: Done
80
81     return ret;
82 }
83
84 void intr(unsigned int intr_mask) {
85     struct cvec_device *cvdev = &cvec_dev;
86     // TODO: lock?
87     spin_lock_irq(&cvdev->lock);
88     // TODO: change status to PROCESSED in slot that was recently processed
89     int slot = get_processed_slot(cvdev);
90     if (slot != -1) {
91         cvdev->buffer[slot].status = PROCESSED;
92     }
93     // TODO: unlock?
94     spin_unlock_irq(&cvdev->lock);
95     wake_up_interruptable(&cvdev->wq);
96 }
97
98 static ssize_t cvec_ioctl(struct file *filp, unsigned int cmd, unsigned long uarg) {
99     struct cvec_device *cvdev = &cvec_dev;
100     struct cvec_descriptor *cvdesc;
101     // TODO: standard ioctl stuff
102     // File descriptor not associated with character special device, or the request does not
    apply
103     // to the kind of object the file descriptor references. (ENOTTY)
104     int ret = -ENOTTY;
105     int retry = -ERESTARTSYS;
106     if(_IOC_TYPE(cmd) != CVEC_IOC_MAGIC) return ret;
107     if(_IOC_NR(cmd) > CVEC_IOC_MAXNR) return ret;
108     // TODO: Done with standard ioctl stuff
109
110     switch(cmd){
111         case CONVERT_VECTOR:
112             cvdesc = kzalloc(sizeof(*cvdesc), GFP_KERNEL);
113
114             // TODO: case of CONVERT_VECTOR
115             // .. initialize structs & copy data from user space
116             struct cvec_state *state;
117             state = filp->private_data;
118             if (down_interruptible(&state->lock)) return retry;
119             if (copy_from_user(&cvdesc, (cvec_descriptor __user *) uarg, sizeof(
cvec_descriptor))) {
120                 return -EFAULT;
121             }
122
123             // .. check if there is a free slot in the buffer otherwise sleep
124             int slot = get_free_slot(cvdev);
125             if (slot == -1) {
126                 wait_event_interruptible(cvdev->wq, get_free_slot(cdev));
127                 slot = get_free_slot(cdev);
128             }
129             up(&state->lock);
130             // .. submit the computation and sleep until it finishes
131             spin_lock_irq(&cdev->lock);
132             cvdev->buffer[slot].cvdesc = cvdesc;
133             cvdev->buffer[slot].status = OCCUPIED;
134             spin_unlock_irq(&cdev->lock);
135             wait_event_interruptible(cvdev->wq, cvdev->buffer[slot].status == PROCESSED);
136             // .. copy data to user space
137             if (copy_to_user((char *)cvdesc.output, cvdesc.output, cvdesc.len)) {
138                 return -EFAULT;
139             }
140             // .. update the buffer
141             break;
142         case SET_CONVERSION:
143             // TODO: change conversion mode
144             struct cvec_state *state = filp->private_data;
145             if (down_interruptible(&state->lock)) return retry;
146             if (state->conversion_mode == DENSE_TO_SPARSE) {

```

```
147         state->conversion_mode = SPARSE_TO_DENSE;
148     } else {
149         state->conversion_mode = DENSE_TO_SPARSE;
150     }
151     up(&state->lock);
152     break;
153 default:
154     ret = -EINVAL;
155     break;
156 }
157
158 return ret;
159 }
```

4 DRIVER K11

```
1 struct input_data {
2     spinlock_t lock;
3     wait_queue_head_t wq;
4     uint32_t cnt;
5
6 #define CIRC_BUF_SIZE (1024 * sizeof(measurement_t))
7     char circ_buffer[CIRC_BUF_SIZE];
8
9 } input_data;
10
11 void intr(void) {
12     struct input_data *inp = &input_data;
13     spin_lock_irq(&inp->lock);
14     memcpy(&inp->circ_buffer[inp->cnt % CIRC_BUF_SIZE],
15         device_memory, sizeof(measurement_t));
16     // cnt
17
18     //
19
20     // CIRC_BUFFER LRU
21     inp->cnt += sizeof(measurement_t);
22     spin_unlock_irq(&inp->lock);
23     // processes data
24     wake_up_interruptible(&inp->wq);
25 }
26
27 struct chrdev_state {
28     struct semaphore lock;
29     struct input_data *inp;
30     // ;
31 }
32
33 #define wait_event_interruptible(waitqueue, condition)
34 unsigned long copy_to_user(void __user *dst, const void *src, unsigned long len);
35
36 static int chrdev_open(struct inode *inode, struct file *filp) {
37     struct chrdev_state *state = kmalloc(sizeof(*state), GFP_KERNEL);
38     filp->private_data = state;
39     //
40     int ret = -ENODEV;
41
42     if ((ret = nonseekable_open(inode, filp)) < 0) {
43         kfree(state);
44         return ret;
45     }
46     sema_init(&state->lock, 1);
47 out:
48     return 0;
49 }
50
51 static ssize_t chrdev_read(struct file *filp, char __user *usrbuf,
52     size_t cnt, loff_t *f_pos) {
53     struct chrdev_state *state = filp->private_data;
54     struct input_data *inp = state->inp;
55     //
56     uint32_t bytes_to_copy = sizeof(measurement_t);
57     uint32_t available = inp->cnt;
58
59     if (down_interruptible(&state->lock)) {
60         return -ERESTARTSYS;
61     }
62
63     while(cnt > 0) {
64         if (inp->cnt == 0) {
65             if (wait_event_interruptible(inp->wq, inp->cnt > 0)) {
66                 up(&state->lock);
67                 return -ERESTARTSYS;
68             }
69         }
70     }
71 }
```



```

70     if (copy_to_user(usrbuf, &inp->circ_buffer[(CIRC_BUF_SIZE - available) % CIRC_BUF_SIZE
71     ],
72         bytes_to_copy)){
73         return -EFAULT;
74     }
75     *f_pos += bytes_to_copy;
76     usrbuf += bytes_to_copy;
77     cnt -= bytes_to_copy;
78     available -= bytes_to_copy;
79
80
81 }
82
83 up(&state->lock);
84 return 0;
85 }

```

5 LUNIX

```
1  /*
2  * linux-chrdev.h
3  *
4  * Definition file for the
5  * Linux:TNG character device
6  *
7  * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
8  */
9
10 #ifndef _LUNIX_CHRDEV_H
11 #define _LUNIX_CHRDEV_H
12
13 /*
14 * Linux:TNG character device
15 */
16 #define LUNIX_CHRDEV_MAJOR 60 /* Reserved for local / experimental use */
17 #define LUNIX_CHRDEV_BUFSZ 20 /* Buffer size used to hold textual info */
18
19 /* Compile-time parameters */
20
21 #ifdef __KERNEL__
22
23 #include <linux/fs.h>
24 #include <linux/kernel.h>
25 #include <linux/module.h>
26
27 #include "linux.h"
28
29 /*
30 * Private state for an open character device node
31 */
32 struct linux_chrdev_state_struct {
33     enum linux_msr_enum type;
34     struct linux_sensor_struct *sensor;
35
36     /* A buffer used to hold cached textual info */
37     int buf_lim;
38     unsigned char buf_data[LUNIX_CHRDEV_BUFSZ];
39     uint32_t buf_timestamp;
40
41     struct semaphore lock;
42
43     /*
44      * Fixme: Any mode settings? e.g. blocking vs. non-blocking
45      */
46     int cmd_arg;
47 };
48
49 /*
50 * Function prototypes
51 */
52 int linux_chrdev_init(void);
53 void linux_chrdev_destroy(void);
54
55 #endif /* __KERNEL__ */
56
57 #include <linux/ioctl.h>
58
59 /*
60 * Definition of ioctl commands
61 */
62 #define LUNIX_IOC_MAGIC LUNIX_CHRDEV_MAJOR
63 #define LUNIX_IOC_EXAMPLE _IOR(LUNIX_IOC_MAGIC, 0, void *)
64
65 #define LUNIX_IOC_MAXNR 0
66
67 #endif /* _LUNIX_H */
68
69
70 /*
71 * linux-chrdev.c
72 *
73 * Implementation of character devices
```

```

5  * for Linux:TNG
6  *
7  * < Your name here >
8  *
9  */
10
11 #include <linux/mm.h>
12 #include <linux/fs.h>
13 #include <linux/init.h>
14 #include <linux/list.h>
15 #include <linux/cdev.h>
16 #include <linux/poll.h>
17 #include <linux/slab.h>
18 #include <linux/sched.h>
19 #include <linux/ioctl.h>
20 #include <linux/types.h>
21 #include <linux/module.h>
22 #include <linux/kernel.h>
23 #include <linux/mmzone.h>
24 #include <linux/vmalloc.h>
25 #include <linux/spinlock.h>
26
27 #include "linux.h"
28 #include "linux_chrdev.h"
29 #include "linux_lookup.h"
30
31 /*
32  * Global data
33  */
34 // struct cdev:
35
36 struct cdev linux_chrdev_cdev;
37
38 /*
39  * Just a quick [unlocked] check to see if the cached
40  * chrdev state needs to be updated from sensor measurements.
41  */
42 /*
43  * Declare a prototype so we can define the "unused" attribute and keep
44  * the compiler happy. This function is not yet used, because this helpcode
45  * is a stub.
46  */
47 // State need Refresh
48 //                                refresh            timestamp
49
50 static int linux_chrdev_state_needs_refresh(struct linux_chrdev_state_struct *state)
51 {
52     struct linux_sensor_struct *sensor;
53
54     WARN_ON ( !(sensor = state->sensor));
55     /* ? */
56     debug("exiting state need refresh");
57     return state->buf_timestamp != sensor->msr_data[state->type]->last_update;
58 }
59
60 /*
61  * Updates the cached state of a character device
62  * based on sensor data. Must be called with the
63  * character device state lock held.
64  */
65
66 // Update State
67 static int linux_chrdev_state_update(struct linux_chrdev_state_struct *state)
68 {
69     struct linux_sensor_struct *sensor = state->sensor;
70     long int proper_data;
71     uint32_t raw_time = sensor->msr_data[state->type]->last_update;
72     uint32_t raw_value = sensor->msr_data[state->type]->values[0];
73     char sign;
74     debug("entering state update");
75
76
77     // debug("leaving\n");
78

```

```

79  /*
80  * Grab the raw data quickly, hold the
81  * spinlock for as little as possible.
82  */
83  /* ? */
84  spin_lock_irq(&sensor->lock);
85  raw_time = sensor->msr_data[state->type]->last_update;
86  raw_value = sensor->msr_data[state->type]->values[0];
87  spin_unlock_irq(&sensor->lock);
88  /* Why use spinlocks? See LDD3, p. 119 */
89
90  /*
91  * Any new data available?
92  */
93  /* ? */
94  if (linux_chrdev_state_needs_refresh(state)) {
95      spin_lock_irq(&sensor->lock);
96      raw_time = sensor->msr_data[state->type]->last_update;
97      raw_value = sensor->msr_data[state->type]->values[0];
98      spin_unlock_irq(&sensor->lock);
99  }
100  else {
101      // -EAGAIN = no data available right now, try again later
102      return -EAGAIN;
103  }
104
105  /*
106  * Now we can take our time to format them,
107  * holding only the private state semaphore
108  */
109
110  switch(state->type){
111      case BATT:
112          proper_data = lookup_voltage[raw_value];
113          break;
114      case TEMP:
115          proper_data = lookup_temperature[raw_value];
116          break;
117      case LIGHT:
118          proper_data = lookup_light[raw_value];
119          break;
120      default:
121          return -EAGAIN;
122  }
123
124  if (proper_data >= 0) {
125      sign='+';
126  }
127  else {
128      sign='-';
129      proper_data = (-1)*proper_data;
130  }
131
132
133  state->buf_lim = snprintf(state->buf_data,
134      LINUX_CHRDEV_BUFSZ, "%c%ld,%ld",
135      sign, proper_data/1000,
136      proper_data%1000);
137  state->buf_timestamp = raw_time;
138
139  /* ? */
140
141  debug("leaving\n");
142  return 0;
143 }
144
145 /*****
146  * Implementation of file operations
147  * for the Linux character device
148  *****/
149
150 // Open System Call
151 static int linux_chrdev_open(struct inode *inode, struct file *filp)
152 {
153     /* Declarations */
154     /* ? */

```

```

155 //          linux_chrdev.h                      struct
156 //
157 //
158 //                                          struct
159 struct linux_chrdev_state_struct *linux_chrdev_state;
160 int ret;
161 int minor = iminor(inode);
162 int sensor = minor >> 3;
163 int type = minor%8; //??
164
165 debug("entering\n");
166 ret = -ENODEV;
167 if ((ret = nonseekable_open(inode, filp)) < 0)
168     goto out;
169
170 /*
171  * Associate this open file with the relevant sensor based on
172  * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
173  */
174
175 /* Allocate a new Linux character device private state structure */
176 /* ? */
177 //          allocate
178 //          struct state
179 linux_chrdev_state = kzalloc(sizeof(*linux_chrdev_state), GFP_KERNEL);
180 //          GFP_KERNEL flag          allocation
181 //          process
182 //          struct
183 //          minor
184 //          struct          :
185 //          -type {BATT, TEMP, LIGHT, N_LUNIX_MSR}
186 //          -sensor,
187 //          sensors
188 //          minor/8.
189 //          -buf_lim
190 //          -buf_data
191 //          -lock
192 //          -buf_timestamp
193 //          -raw_data
194
195 linux_chrdev_state->sensor = &linux_sensors[sensor];
196 linux_chrdev_state->type = type;
197 linux_chrdev_state->buf_lim = 0;
198 linux_chrdev_state->buf_timestamp = 0;
199 // struct file: void *private_data;
200 // The open system call sets this pointer to NULL before calling the open method for the
201 // driver.
202 // The driver is free to make its own use of the field or to ignore it.
203 // The driver can use the field to point to allocated data, but then must free memory
204 // in the release method before the file structure is destroyed by the kernel.
205 // private_data is a useful resource for preserving state information across system calls
206 // and is used by most of our sample modules.
207 filp->private_data = linux_chrdev_state;
208
209 // Initialize lock
210 sema_init(&linux_chrdev_state->lock,1);
211 out:
212 debug("leaving, with ret = %d\n", ret);
213 return ret;
214 }
215
216 // Release System Call (free allocated memory of file)
217 static int linux_chrdev_release(struct inode *inode, struct file *filp)
218 {
219     /* ? */
220     kfree(filp->private_data);
221     return 0;
222 }
223
224 // Device Specific Commands
225 static long linux_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
226 {
227     // File descriptor not associated with character special device, or the request does not

```

```

226     apply
227     // to the kind of object the file descriptor references. (ENOTTY)
228     int ret = -ENOTTY;
229     int retry = -ERESTARTSYS;
230     struct linux_chrdev_state_struct *state;
231
232     // https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch05.html
233     if(_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return ret;
234     if(_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return ret;
235     state = filp->private_data;
236
237     switch(cmd) {
238     case LUNIX_IOC_EXAMPLE:
239         if(down_interruptible(&state->lock)) return retry;
240         debug("in LUNIX_IOC_EXAMPLE area");
241         up(&state->lock);
242         break;
243     default: return ret;
244     }
245     /* Why? */
246     // return -EINVAL; Invalid Argument
247     debug("ioctl done my guy ");
248     return 0;
249 }
250
251 static ssize_t linux_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *
252     f_pos)
253 {
254     ssize_t ret = 0;
255     int retry = -ERESTARTSYS;
256
257     struct linux_sensor_struct *sensor;
258     struct linux_chrdev_state_struct *state = filp->private_data;
259     WARN_ON(!state);
260
261     sensor = state->sensor;
262     WARN_ON(!sensor);
263
264     /* Lock? */
265     // semaphore waitqueue process
266     if(down_interruptible(&state->lock)){
267         debug("down interruptable failed in read");
268         return retry;
269     }
270     /*
271     * If the cached character device state needs to be
272     * updated by actual sensor data (i.e. we need to report
273     * on a "fresh" measurement, do so
274     */
275     if (*f_pos == 0) {
276         // Nothing to read
277         while (linux_chrdev_state_update(state) == -EAGAIN) {
278             /* ? */
279             /* The process needs to sleep */
280             /* See LDD3, page 153 for a hint */
281             up(&state->lock);
282             if (wait_event_interruptible(sensor->wq, linux_chrdev_state_needs_refresh(state))){
283                 debug("wait interruptable failed in read");
284                 return -ERESTARTSYS;
285             }
286             if (down_interruptible(&state->lock)){
287                 debug("down interruptable failed in read");
288                 return -ERESTARTSYS;
289             }
290         }
291     }
292
293     /* End of file */
294     /* ? */
295
296     /* Determine the number of cached bytes to copy to userspace */
297     /* ? */
298     // user
299     ,
300     //

```

```

298     if (*f_pos + cnt >= state->buf_lim) {
299         cnt = state->buf_lim - *f_pos;
300     }
301
302     if (copy_to_user(usrbuf, state->buf_data, cnt)) {
303         ret=-EFAULT;
304         goto out;
305     }
306
307     // offset
308     *f_pos += cnt;
309     ret = cnt;
310
311
312     /* Auto-rewind on EOF mode? */
313     /* ? */
314     if (*f_pos == state->buf_lim){*f_pos = 0;}
315
316 out:
317     /* Unlock? */
318     up(&state->lock);
319     return ret;
320 }
321
322 // Mmap
323 static int linux_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
324 {
325     return -EINVAL;
326 }
327
328 // cdev
329 static struct file_operations linux_chrdev_fops =
330 {
331     .owner          = THIS_MODULE,
332     .open           = linux_chrdev_open,
333     .release        = linux_chrdev_release,
334     .read           = linux_chrdev_read,
335     .unlocked_ioctl = linux_chrdev_ioctl,
336     .mmap           = linux_chrdev_mmap
337 };
338
339 // Initialization
340 int linux_chrdev_init(void)
341 {
342     /*
343      * Register the character device with the kernel, asking for
344      * a range of minor numbers (number of sensors * 8 measurements / sensor)
345      * beginning with LINUX_CHRDEV_MAJOR:0
346      */
347     int ret;
348     dev_t dev_no;
349     unsigned int linux_minor_cnt = linux_sensor_cnt << 3;
350
351     // sensors * 2^3
352     // minor numbers
353
354     debug("initializing character device\n");
355     cdev_init(&linux_chrdev_cdev, &linux_chrdev_fops);
356     // cdev
357
358     linux_chrdev_cdev.owner = THIS_MODULE;
359
360     dev_no = MKDEV(LINUX_CHRDEV_MAJOR, 0);
361     // device ID
362     /* ? */
363     /* register_chrdev_region? */
364     // device numbers
365
366     // (
367     // Major Number),
368     // ,
369     ret = register_chrdev_region(dev_no, linux_minor_cnt, "linux");
370     if (ret < 0) {
371         debug("failed to register region, ret = %d\n", ret);
372         goto out;
373     }

```

```

370  /* ? */
371  /* cdev_add? */
372  //
373  //                                :                cdev,
374  //                                minor numbers.
375  ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
376  if (ret < 0) {
377      debug("failed to add character device\n");
378      goto out_with_chrdev_region;
379  }
380  debug("completed successfully\n");
381  return 0;
382
383 out_with_chrdev_region:
384     unregister_chrdev_region(dev_no, lunix_minor_cnt);
385 out:
386     return ret;
387 }
388
389 // Destroy
390 void lunix_chrdev_destroy(void)
391 {
392     dev_t dev_no;
393     unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;
394
395     debug("entering\n");
396     dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
397     cdev_del(&lunix_chrdev_cdev);
398     unregister_chrdev_region(dev_no, lunix_minor_cnt);
399     debug("leaving\n");
400 }

```