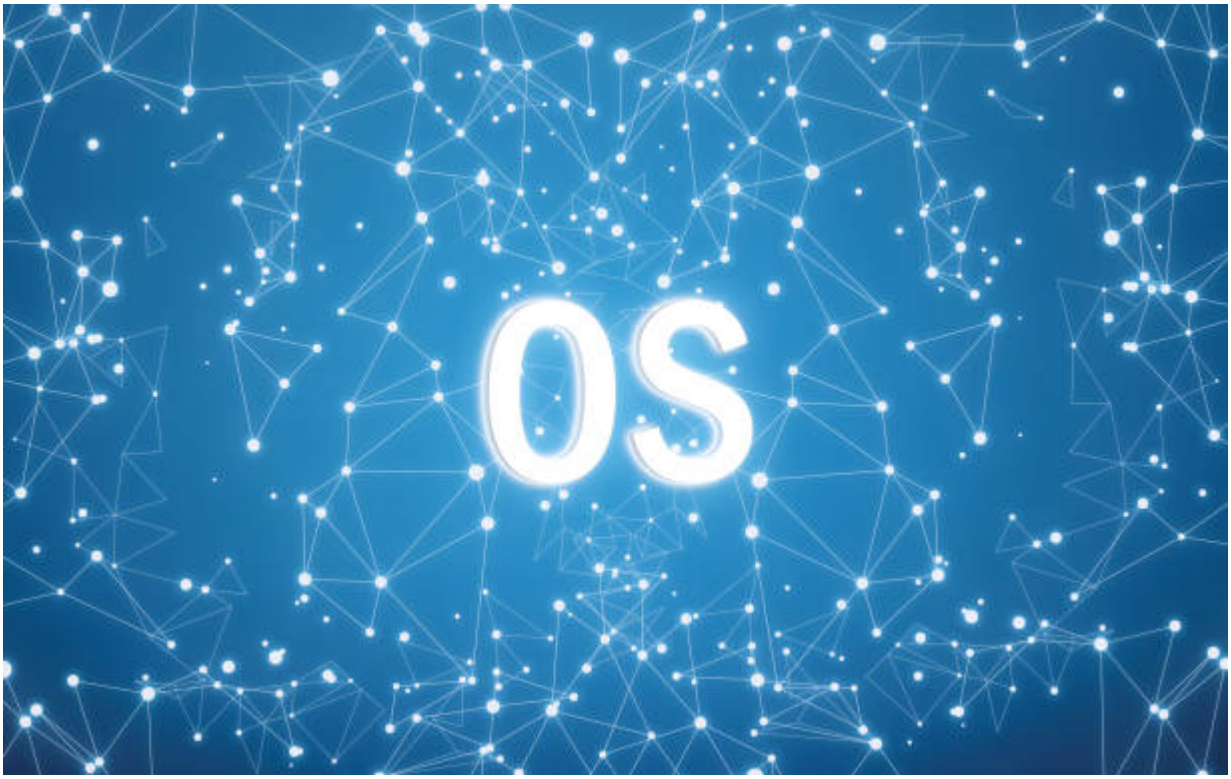
Εργαστήριο Λειτουργικών Συστημάτων

Χριστόπουλος Νικόλαος: 03117065
Γιαννούλης Παναγιώτης: 03117812
8 Δεκεμβρίου 2020



Εισαγωγή

Σκοπός μας είναι η δημιουργία ενός character device driver για ένα ασύρματο δίκτυο αισθητήρων στο λειτουργικό σύστημα Linux ο οποίος θα επιτελεί κάποιες λειτουργίες. Συγκεκριμένα, το δίκτυο αποτελείται από έναν αριθμό από αισθητήρες και ένα σταθμό βάσης ο οποίος συνδέεται μέσω USB με υπλογιστικό σύστημα Linux στο οποίο θα εκτελείται και ο ζητούμενος οδηγός συσκευής.

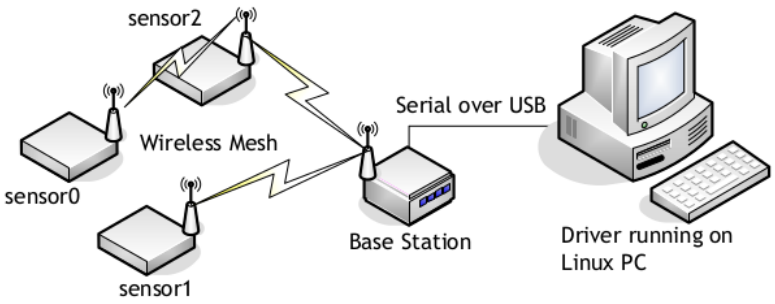


Figure 1: Architecture of the system

Τα πακέτα με δεδομένα μετρήσεων μεταφέρονται από τους αισθητήρες στο σταθμό βάσης και εκείνος τα προωθεί μέσω διασύνδεσης USB στο υπολογιστικό σύστημα. Ο οδηγός που καλούμαστε να υλοποιήσουμε πρέπει να αντιμετωπίζει ανεξάρτητα τον κάθε αισθήτηρα και το μετρούμενο μέγεθος μέσω διαφορετικών συσκευών. Επιπλεόν, πρέπει να επιτρέπει και να διαχειρίζεται κατάλληλα την ταυτόχρονη πρόσβαση στα εισερχόμενα δεδομένα, ενώ χρειάζεται να εφαρμόσει την αντίστοιχη πολιτική για τους χρήστες όσον αφορά την πρόσβαση σε αυτά.

Αρχιτεκτονική του Λογιστμικού του Συστήματος

Το σύστημα οργανώνεται σε δύο κύρια μέρη: Το πρώτο αναλαμβάνει τη συλλογή των δεδομένων από το σταθμό βάσης (Lunix line discipline) και την επεξεργασία τους με συγκεκριμένο πρωτόκολλο (Lunix protocol), έτσι ώστε να εξαχθούν οι τιμές των μετρούμενων μεγεθών, οι οποίες κρατούνται σε κατάλληλες δομές προσωρινής αποθήκευσης στη μνήμη (Lunix sensor buffers). Το δεύτερο μέρος αναλαμβάνει να παραλάβει τα δεδομένα από τους προσωρινούς buffers και να τα εξάγει στο χώρο χρήστη στην κατάλληλη μορφή, υλοποιώντας μια σειρά από συσκευές χαρακτήρων.

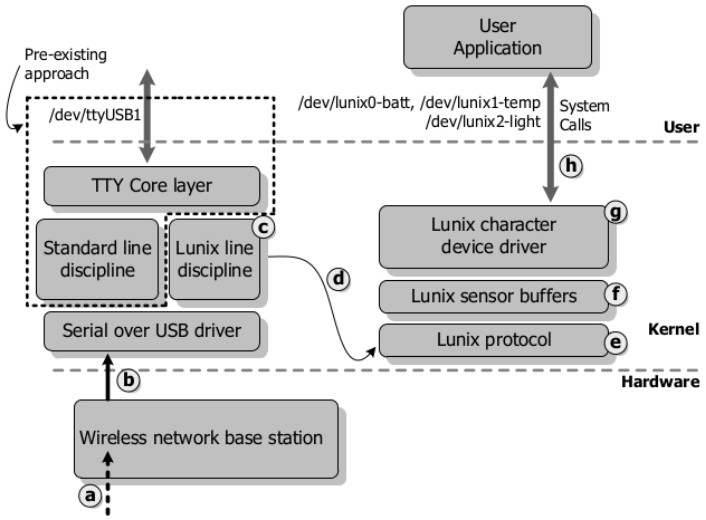


Figure 2: Architecture of the software

Το τελικό σύστημα θα λαμβάνει τα δεδομένα των μετρήσεων από το σταθμό βάσης. Αυτά θα προωθούνται μέσω USB στο υπολογιστικό σύστημα, θα παραλαμβάνονται από ένα φίλτρο, τη διάταξη γραμμής του Lunix, η οποία θα τα προωθεί στο κατάλληλο στρώμα ερμηνείας του πρωτοκόλλου των αισθητήρων. Το συγκεκριμένο είναι υπεύθυνο για την ερμηνεία των bytes των πακέτων και την αποθήκευση τους στους αντίστοιχους buffers του κάθε αισθητήρα. Ο device driver ανάλογα με το αρχείο που χρησιμοποιείται στο χώρο χρήστη θα ανακτά τα δεδομένα από τους buffers και θα τα παρουσιάζει στην κατάλληλη μορφή.

Υλοποίηση:

Δομές

Πριν αναφερθούμε στην υλοποιήση αυτή καθ' αυτή, είναι σημαντικό να δούμε τις σημαντικότερες δομές που χρησιμοποιούνται στον οδηγό. Όπως είδαμε και προηγουμένως τα δεδομένα που προέρχονται από τον σταθμό βάσης αφού ερμηνευθούν από το κατάλληλο πρωτόκολλο αποθηκεύονται σε προσωρινούς buffers. Κάθε τέτοιος buffer αντιστοιχεί σε έναν συγκεκριμένο αισθητήρα. Επομένως, η παρακάτω δομή είναι απαραίτητη γι' αυτό το σκοπό.

```
enum lunix_msr_enum { BATT = 0, TEMP, LIGHT, N_LUNIX_MSR };

struct lunix_sensor_struct {
  /*
   * A number of pages, one for each measurement.
   * They can be mapped to userspace.
   */
  struct lunix_msr_data_struct *msr_data[N_LUNIX_MSR];

  /*
```

```
   * Spinlock used to assert mutual exclusion between
   * the serial line discipline and the character device driver
   */
  spinlock_t lock;

  /*
   * A list of processes waiting to be woken up
   * when this sensor has been updated with new data
   */
  wait_queue_head_t wq;
};

struct lunix_msr_data_struct {
  uint32_t magic;
  uint32_t last_update;
  uint32_t values[];
};
```

Τα επιμέρους στοιχεία της δομής θα αναλυθούν στη συνέχεια και θα αναδειχθεί ο ρόλος τους. Είναι επίσης σημαντικό να τονίσουμε τα global στοιχεία του driver μας, γιατί αυτά είναι διαθέσιμα σε οποιοδήποτε κομμάτι του driver αλλά ταυτόχρονα είναι τα πιο επιρρεπή σε σφάλματα τύπου race conditions.

```
/*
 * Global state for Lunix:TNG sensors
 */
int lunix_sensor_cnt = LUNIX_SENSOR_CNT;
struct lunix_sensor_struct *lunix_sensors;
```

Ο pointer lunix_sensors είναι ένας πίνακας από 16 pointers που ο καθένας αντιστοιχεί στην παραπάνω δομή για τον κάθε αισθητήρα. Η δομή αυτή είναι αρκετή για να μας επιτρέψει να ασχοληθούμε με το τι θα συμβεί όταν κάνουμε load το module μας στον πυρήνα. Το αρχείο lunix-module.c αποτελεί την main του driver μας και γι' αυτό το λόγο εκεί συμβαίνουν όλα τα initializations και οι απαραίτητες αναθέσεις στη μνήμη των δομών που θα χρησιμοποιηθούν. Οι αισθητήρες μας είναι 16 στον αριθμό, επομένως ζητάμε από το λειτουργικό μας την αντίστοιχη μνήμη:

```
lunix_sensors = kzalloc(sizeof(*lunix_sensors) * lunix_sensor_cnt,
    GFP_KERNEL);
```

Στη συνέχεια για κάθε σένσορα καλείται η **lunix_sensor_init** που αρχικοποιεί την δομή μας και ζητά σελίδες μνήμης για την δομή των μετρήσεων. Παρακάτω, βλέπουμε την εξής κλήση:

```
lunix_chrdev_init()
```

Η συγκεκριμένη συνάρτηση φροντίζει να αρχικοποιήσει τις συσκευές μας και να τις αναθέσει στον πυρήνα του συστήματος. Για την επίτευξη αυτού είναι απαραίτητη η δομή lunix_chrdev_cdev τύπου **cdev** που υπάρχει σε κάθε character device driver και αντιστοιχίζεται με βασικά χαρακτηριστικά όπως minor, major numbers, τον owner, τα file operations. Η δομή file operations ορίζει ποιες συναρτήσεις θα καλεστούν για τις λειτουργίες του driver όπως open , read, ioctl κ.α. Με βάση, λοιπόν, τα παραπάνω γίνονται τα εξής:

```
cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
```

Η cdev_init αρχικοποιεί την δομή cdev και αναθέτει τα file operations στο αντίστοιχο πεδίο της δομής αυτής. Δεδομένου ότι γνωρίζουμε εξαρχής τον αριθμό των αισθητήρων χρησιμοποιούμε την register_chrdev_region για την ανάθεση μιας περιοχής από device minor numbers. Επίσης έχει τεθεί ο αριθμός 60 ως major number για τον driver μας.

```
register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
```

Τέλος, χρησιμοποιείται η συνάρτηση cdev_add για την ενημέρωση του kernel για τους device drivers. Με την κλήση αυτή ο driver μας είναι πια ζωντανός στο σύστημα και οι λειτουργίες του μπορούν να καλεστούν οποιαδήποτε στιγμή. Ήρθε η ώρα να υλοποιήσουμε τις συναρτήσεις που θα χρησιμοποιούνται από τον driver μας.

Open

Για την υλοποίηση της open() χρειάζεται να αναφερθούμε σε μια ακόμη δομή που θα δημιουργείται σε κάθε κλήση της. Συγκεκριμένα, αυτή η δομή μας πληροφορεί για το state κάθε συσκευής που ανοίγει από κάποια διεργασία. Στην ίδια δομή υπάρχει ένας buffer που θα αποθηκεύονται τα δεδομένα από τους sensor-buffers και από εκεί θα μεταφέρονται στο χώρο χρήστη. Επιπλέον, υπάρχει μια μεταβλητή που μας ενημερώνει για το ποια ήταν η τελευταία ανανέωση που έγινε στον buffer αυτόν και το πεδίο buf_lim που μας δείχνει πόσες θέσεις του buffer έχουν χρησιμοποιηθεί.

```
struct lunix_chrdev_state_struct {
  enum lunix_msr_enum type;
  struct lunix_sensor_struct *sensor;

  /* A buffer used to hold cached textual info */
  int buf_lim;
  unsigned char buf_data[LUNIX_CHRDEV_BUFSZ];
  uint32_t buf_timestamp;

  struct semaphore lock;
  int raw_data;
  //int vmas;
  /*
   * Fixme: Any mode settings? e.g. blocking vs. non-blocking
   */

};
```

Αρχικά, ζητάμε την απαραίτητη μνήμη από το λειτουργικό:

```
lunix_chrdev_state = kzalloc(sizeof(*lunix_chrdev_state), GFP_KERNEL);
```

Η συνάρτηση open βλέπουμε ότι έχει δύο πολύ σημαντικά arguments, τις δομές inode, και file. Την πρώτη την χρειαζόμαστε για να μάθουμε τον minor number της συσκευής που ανοίχτηκε. Ο minor number θα μας υποδείξει τον αριθμό του αισθήτηρα και τον τύπο των μετρήσεων. Μέσω αυτής της πληροφορίας αναθέτουμε στο πεδίο sensor τον αντίστοιχο σένσορα. Αρχικοποιούμε κατάλληλα τα υπόλοιπα πεδία και τέλος χρησιμοποιούμε το πεδίο

private_data της δομής file για να αποθηκεύσουμε το state και να το χρησιμοποιήσουμε στις υπόλοιπες συναρτήσεις.

Η συνάρτηση release() είναι η αντίστοιχη της close(). Σε αυτήν, λοιπόν, απελευθερώνουμε την μνήμη που δεσμεύσαμε για την δομή state.

Read/Refresh/Update

Τώρα μεταφερόμαστε στο σημαντικότερο κομμάτι της υλοποιήσης, στο οποίο γίνεται η βασική λειτουργία του read. Πριν όμως υλοποιήσουμε την read, χρησιμοποιούμε δύο βοηθητικές συναρτήσεις γι' αυτήν, την refresh και την update. Η πρώτη αυτό που κάνει είναι να βλέπει αν χρειάζεται να γίνει ανανέωση των δεδομένων που έχουν διαβαστεί ως τώρα από τους σένσορες. Πως θα γίνει αυτό; Πολύ απλά συγκρίνουμε την τελευταία φορά που πήραμε δεδομένα από τους sensor buffers και την τελευταία φορά που ήρθαν δεδομένα σε αυτούς. Με αυτό τον τρόπο συμπεραίνουμε αν ο δικός μας buffer χρειάζεται refresh.

```
return state->buf_timestamp != sensor->msr_data[BATT]->last_update;
```

Η δεύτερη αν χρειάζεται παίρνει τα καινούργια δεδομένα και τα μεταφέρει κατάλληλα μορφοποιημένα στον buffer του state ώστε να είναι έτοιμα να περαστούν στο χώρο χρήστη. Σε αυτό το σημείο χρειάζεται να προσέξουμε μερικά πολύ σημαντικά σημεία.

```
spin_lock_irq(&sensor->lock);
uint32_t timestamp = sensor->msr_data[BATT]->last_update;
uint32_t new_value = sensor->msr_data[state->type]->values[0];
spin_unlock_irq(&sensor->lock);
```

Πρέπει να χρησιμοποιήσουμε spin locks ώστε να αποφύγουμε ένα πιθανό race condition. Ποιο είναι αυτό; Αν δεν είχαμε αυτό το κλείδωμα (ένα για κάθε αισθητήρα) είναι πιθανό όσο διαβάζουμε εμείς κάποιες τιμές (last_update, values/μετρήσεις) που αφορούν στον κάθε αισθητήρα, ταυτόχρονα αυτές να ανανεώνονται και επομένως να λάβουμε λανθασμένες τιμές. Οπότε κάθε φορά που είτε διαβάζουμε είτε επεξαργαζόμαστε τα πεδία της δομής του σένσορα πρέπει να κλειδώνουμε την πόρτα μας. Το σκοπό αυτό λοιπόν, εξυπηρετεί το παρακάτω κλείδωμα:

```
spinlock_t lock;
```

Κάποιος μπορεί να ρωτήσει γιατί χρησιμοποιούμε κλείδωμα και όχι κάποιο σεμαφόρο/mutex. Γιατί δε θέλουμε με τίποτα να μεταφερθούμε σε sleep mode αφού όταν ανανεώνονται τα δεδομένα στους sensor buffers βρισκόμαστε σε interrupt context και δεν υπάρχει κάποια διεργασία για να κοιμηθεί. Επομένως, τα spin locks είναι αυτά που χρειαζόμαστε γιατί χρησιμοποιούν την μέθοδο του polling για να πάρουν κάποιο lock. Πρέπει όμως να είμαστε αρκέτα προσεκτικοί και να μην κρατάμε το lock για μεγάλο χρονικό διάστημα, γιατί έτσι κάποιος άλλος που προσπαθεί να το πάρει θα σπαταλάει αρκετό χρόνο από την CPU. Επιπλεόν είναι απαραίτητο να απενεργοποιήσουμε τα interrupts στην περίπτωση που λαμβάνουμε καινούργια δεδομένα από τους sensor buffers. Δε θέλουμε να έρθει κάποιο interrupt που θα θέλει να κάνει ανανέωση των δεδομένων όσο έχουμε το lock γιατί έτσι οι sensors δε θα μπορέσουν ποτέ να πάρουν το κλείδωμα και θα προκύψει deadlock.

Αν, λοιπόν, πάνε όλα καλά και πάρουμε τα δεδομένα μας στη συνέχεια τα μορφοποιούμε κάνοντας χρήση των ειδικών πινάκων, ανανεώνουμε το timestamp και μεταφέρουμε τα δεδομένα μας στον buffer.

```
state->buf_timestamp = timestamp;

value = lookup_voltage[new_value];

state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%ld.%ld\n
    ", value/1000, value%1000);
```

Τέλος, ας δούμε την υλοποίηση της read(). Κατά τη διάρκεια που εκτελείται η read() πρέπει να χρησιμοποιείται ένας σεμαφόρος για την ασφάλεια των δεδομένων της δομής state. Αυτό συμβαίνει γιατί υπάρχουν περιπτώσεις που περισσότερες από μια διεργασίες έχουν κοινό lunix driver state. Πως μπορεί να επιτευχθεί αυτό; Όταν σε μια διεργασία κάνουμε open κάποια συσκευή και έπειτα κάνουμε fork, τότε ο fd της open κληρονομείται και στις διεργασίες παιδιά, επομένως αν πατέρας και παιδί διαβάσουν ταυτόχρονα καινούργια δεδομένα καλώντας την read τότε θα προκύψει race condition στα πεδία του state. Αρκεί, λοιπόν, ένας σεμαφόρος για να λύσει το πρόβλημά μας.

```
/*struct semaphore lock; */

down_interruptible(&state->lock);
...
up(&state->lock);
```

Επίσης, πρέπει να επισημάνουμε ότι στην περίπτωση κάποιου interrupt αυτός που έχει το κλείδωμα πρέπει να το αφήνει. Για το σκοπό αυτό χρησιμοποιείται συγκεκριμένα η συνάρτηση down_interruptible.

```
if (*f_pos == 0) {
    while (lunix_chrdev_state_update(state) == -EAGAIN) {
      /* ? */
      /* The process needs to sleep */
      /* See LDD3, page 153 for a hint */
      up(&state->lock);
      debug("\"%s\" reading: going to sleep\n", current->comm);
      if (wait_event_interruptible(sensor->wq,
   lunix_chrdev_state_needs_refresh(state)))
          return -ERESTARTSYS;
      if (down_interruptible(&state->lock))
          return -ERESTARTSYS;
    }
  }
```

Ας αγνοήσουμε για αρχή το *f_pos. Στο παραπάνω κομμάτι κώδικα αυτό που ελέγχουμε είναι αν χρειάζεται να γίνει κάποιο update. Στην περίπτωση που δεν έχουν έρθει καινούργια δεδομένα θέλουμε οι διεργασίες μας να πηγαίνουν για ύπνο ώστε να μην χρησιμοποιούν την CPU χωρίς λόγο. Προσοχή όμως, όταν οι διεργασίες πηγαίνουν για ύπνο πρέπει να αφήνουν το κλείδωμα. Όταν ενημερωθούν οι sensor buffers είναι υπεύθυνοι να ξυπνήσουν τις διεργασίες αυτές:

```
  /*
   * And wake up any sleepers who may be waiting on
   * fresh data from this sensor.
   */
  wake_up_interruptible(&s->wq);
```

Επίσης πρέπει να παρατηρήσουμε ότι το wait_event είναι interruptible, που σημαίνει ότι οι διεργασίες μπορούν να ξυπνήσουν μέσω ενός interrupt και αυτός είναι ένας λόγος που βλέπουμε ως δεύτερο όρισμα την lunix_chrdev_state_needs_refresh, ώστε να γίνει επανέλεγχος αφού ξυπνήσουν οι διεργασίες ότι όντως έχουμε καινούργια δεδομένα και να αποσαφηνιστεί ο λόγος της αφύπνισης(interrupt or new data).

Η μεταβλητή *f_pos δείχνει πόσα bytes έχουν διαβαστεί εώς τώρα από τον χρήστη. Για παράδειγμα, αν έχουμε 10 bytes δεδομένων και έχουν διαβαστεί τα 5, το *f_pos θα δείχνει στο 6ο byte. Ο λόγος, λοιπόν, που ανανεώνουμε τα δεδομένα μας μόνο όταν το *f_pos είναι 0, είναι γιατί περιμένουμε ο χρήστης να διαβάσει πρώτα όλα τα προηγούμενα. Αν δεν συνέβαινε αυτό ο χρήστης θα λάμβανε λάθος μετρήσεις. Όταν, λοιπόν, διαβαστούν όλα τα bytes το *f_pos μηδενίζεται.

```
if (*f_pos == state->buf_lim) {
  *f_pos = 0;
}
```

Τα δεδομένα μεταφέρονται με ασφάλεια στον χώρο χρήστη μέσω της εντολής copy_to_user(). Σε περιπτώσεις κάποιου null dereference ή αποτυχίας μεταφοράς όλων των δεδομένων επιστρέφεται η τιμή -EFAULT.

```
if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)) {
    ret = -EFAULT;
    goto out;
  }
\end{document}
```

Ioctl

Τέλος, υλοποιήθηκε και η ioctl μέσω της οποίας ο χρήστης μπορεί να επιλέξει αν θέλει τα δεδομένα να του έρχονται σε raw μορφή ή μορφοποιημένα. Γι' αυτό το σκοπό χρησιμοποιούμε ένα επιπλέον πεδίο στο structure του state, την μεταβλήτη raw_data. Επίσης ορίζουμε την εντολή που θα χρησιμοποιηθεί από την ioctl και το νούμερο της, η οποία θα παίρνει και μία παράμετρο:

```
#define LUNIX_RAW_DATA _IOR(LUNIX_IOC_MAGIC, 0, int)
```

Τέλος, στην συνάρτηση έχουμε το εξής:

```
switch(cmd) {
    case LUNIX_RAW_DATA:
        down_interruptible(&state->lock);
        state->raw_data = (int) arg;
        up(&state->lock);
        break;
```

```
    default:
        return -ENOTTY;
  }
```

Μέσω του argument cmd της ioctl αναγνωρίζουμε το command που έχει χρησιμοποιήσει ο χρήστης και διαβάζουμε την παράμετρο η οποία μας πληροφορεί για το αν θέλει τα δεδομένα σε μορφή raw ή οχι. Εννοείται, όταν αλλάζουμε τη δομή του state παίρνουμε και το κλείδωμα. Παρακάτω μπορούμε να δούμε όλο τον κώδικα του αρχείου lunix_driver.c.

```
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * Christopni (aka Christopoulos Nikos), PanGiann (aka Giannoulis
   Panagiotis)
 *
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mmzone.h>
#include <linux/vmalloc.h>
#include <linux/spinlock.h>
#include <linux/page-flags.h>
#include "lunix.h"
#include "lunix-chrdev.h"
#include "lunix-lookup.h"




/*
 * Global data
 */
struct cdev lunix_chrdev_cdev;

/*
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */
```

```
static int lunix_chrdev_state_needs_refresh(struct
    lunix_chrdev_state_struct *state)
{
  struct lunix_sensor_struct *sensor;

  WARN_ON ( !(sensor = state->sensor));
  /* ? */
  return state->buf_timestamp != sensor->msr_data[BATT]->last_update;
  /* The following return is bogus, just for the stub to compile */
   /* ? */
}

/*
 * Updates the cached state of a character device
 * based on sensor data. Must be called with the
 * character device state lock held.
 */
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *
    state)
{
  struct lunix_sensor_struct *sensor;

  debug("leaving\n");
  int ret;
  /*
   * Grab the raw data quickly, hold the
   * spinlock for as little as possible.
   */
   WARN_ON( !(sensor = state->sensor));


  /* ? */
  /* Why use spinlocks? See LDD3, p. 119 */

  /*
   * Any new data available?
   */


   spin_lock_irq(&sensor->lock);
   uint32_t timestamp = sensor->msr_data[BATT]->last_update;
   uint32_t new_value = sensor->msr_data[state->type]->values[0];
   spin_unlock_irq(&sensor->lock);

  /* ? */

  /*
   * Now we can take our time to format them,
   * holding only the private state semaphore
   */
```

```
  /* ? */
  long value;
  if (lunix_chrdev_state_needs_refresh(state)) {
    state->buf_timestamp = timestamp;
    if (!state->raw_data) {
        switch (state->type) {
            case BATT:
                value = lookup_voltage[new_value];
                break;
            case TEMP:
                value = lookup_temperature[new_value];
                break;
            case LIGHT:
                value = lookup_light[new_value];
                break;
            default:
                debug("We shouldn't be here\n");
                return -EAGAIN;

        }


        debug("new data are %ld\n", value);
        state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%
   ld.%ld\n", value/1000, value%1000);
        debug("New data came and specifically %d bytes\n", state->buf_lim
   );
    }
    else {
        debug("New data without formatting them\n");
        state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%
   ld\n", new_value);
    }
  }
  else {
    debug("No new data\n");
    return -EAGAIN;
  }

out:
  debug("leaving\n");
  return 0;
}

/*************************************
 * Implementation of file operations
 * for the Lunix character device
 *************************************/
```

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
  /* Declarations */
  /* ? */
  struct lunix_chrdev_state_struct *lunix_chrdev_state;
  int ret;

  debug("entering\n");
  ret = -ENODEV;
  if ((ret = nonseekable_open(inode, filp)) < 0)
    goto out;

  /*
   * Associate this open file with the relevant sensor based on
   * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
   */


  /* Allocate a new Lunix character device private state structure */

  lunix_chrdev_state = kzalloc(sizeof(*lunix_chrdev_state), GFP_KERNEL);
  if (!lunix_chrdev_state) {
    ret = -ENOMEM;
    printk(KERN_ERR "Failed to allocate memory for Lunix driver state");
    goto out;
  }
  int minor_num = iminor(inode);
  int lunix_sensor_num = minor_num >> 3;
  lunix_chrdev_state->sensor = &(lunix_sensors[lunix_sensor_num]);
  sema_init(&lunix_chrdev_state->lock, 1);
  lunix_chrdev_state->type = minor_num & 0x7;
  lunix_chrdev_state->buf_lim = 0;
  lunix_chrdev_state->buf_timestamp = 0;
  lunix_chrdev_state->raw_data = 0;
  filp->private_data = lunix_chrdev_state;
  debug("successfully allocated lunix_chrdev_state\n");
out:
  debug("leaving, with ret = %d\n", ret);
  return ret;
}

static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{

  /* ? */
  struct lunix_chrdev_state_struct *state;
  state = filp->private_data;
  WARN_ON(!state);

  kfree(state);
```

```
  printk(KERN_INFO "Lunix chatacter device closed successfully\n");
  return 0;
}

static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd,
   unsigned long arg)
{
  if (_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return -ENOTTY;
  if (_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return -ENOTTY;

  struct lunix_chrdev_state_struct *state;
  state = filp->private_data;
  switch(cmd) {
    case LUNIX_RAW_DATA:
        down_interruptible(&state->lock);
        state->raw_data = (int) arg;
        up(&state->lock);
        break;
    default:
        return -ENOTTY;
  }
  return 0;
}

static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf,
   size_t cnt, loff_t *f_pos)
{
  ssize_t ret;
  int flag;
  struct lunix_sensor_struct *sensor;
  struct lunix_chrdev_state_struct *state;

  state = filp->private_data;
  WARN_ON(!state);

  sensor = state->sensor;
  WARN_ON(!sensor);

  /* Lock? */
  down_interruptible(&state->lock);
  /*
   * If the cached character device state needs to be
   * updated by actual sensor data (i.e. we need to report
   * on a "fresh" measurement, do so
   */
  if (*f_pos == 0) {
    while (lunix_chrdev_state_update(state) == -EAGAIN) {
      /* ? */
      /* The process needs to sleep */
      /* See LDD3, page 153 for a hint */
```

```
      up(&state->lock);
      debug("\"%s\" reading: going to sleep\n", current->comm);
      if (wait_event_interruptible(sensor->wq,
   lunix_chrdev_state_needs_refresh(state)))
          return -ERESTARTSYS;
      if (down_interruptible(&state->lock))
          return -ERESTARTSYS;
    }
  }

  cnt = ( cnt >= state->buf_lim - *f_pos ? state->buf_lim - *f_pos : cnt
   );
  /* End of file */
  /* ? */

  /* Determine the number of cached bytes to copy to userspace */
  /* ? */



  if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)) {
    ret = -EFAULT;
    goto out;
  }
  debug("We read %d bytes of data\n", cnt);
  *f_pos += cnt;
  ret = cnt;

  /* Auto-rewind on EOF mode? */
  /* ? */
  if (*f_pos == state->buf_lim) {
    *f_pos = 0;
  }
out:
  /* Unlock? */
  up(&state->lock);
  return ret;
}




static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *
   vma)
{

  /*if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
                  vma->vm_end - vma->vm_start,
                  vma->vm_page_prot))
          return -EAGAIN;
```

```
  vma->vm_ops = &simple_remap_vm_ops;
  vma->vm_private_data = filp->private_data;
  simple_vma_open(vma);*/
  return 0;
}

static struct file_operations lunix_chrdev_fops =
{
  .owner          = THIS_MODULE,
  .open           = lunix_chrdev_open,
  .release        = lunix_chrdev_release,
  .read           = lunix_chrdev_read,
  .unlocked_ioctl = lunix_chrdev_ioctl,
  .mmap           = lunix_chrdev_mmap
};

int lunix_chrdev_init(void)
{
  /*
   * Register the character device with the kernel, asking for
   * a range of minor numbers (number of sensors * 8 measurements /
   sensor)
   * beginning with LINUX_CHRDEV_MAJOR:0
   */
  int ret;
  dev_t dev_no;
  unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

  debug("initializing character device\n");
  cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
  lunix_chrdev_cdev.owner = THIS_MODULE;

  dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
  /* ? */
  /* register_chrdev_region? */
  ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
  if (ret < 0) {
    debug("failed to register region, ret = %d\n", ret);
    goto out;
  }
  /* ? */
  /* cdev_add? */
  ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
  if (ret < 0) {
    debug("failed to add character device\n");
    goto out_with_chrdev_region;
  }
  debug("completed successfully\n");
```

```
  return 0;

out_with_chrdev_region:
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
out:
  return ret;
}

void lunix_chrdev_destroy(void)
{
  dev_t dev_no;
  unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

  debug("entering\n");
  dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
  cdev_del(&lunix_chrdev_cdev);
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
  debug("leaving\n");
}
```