TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikka, ohjelmistotekniikka
Timo-Pekka Launonen

Tutkintotyö

Timo-Pekka Launonen
LCD-NÄYTÖN LAITEAJURI LINUX-KÄYTTÖJÄRJESTELMÄSSÄ

Työn ohjaaja Lehtori Tony Torp
Työn teettäjä Elektrobit Oy, valvojana Jussi Kallioniemi
Tampere 2006

TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikka, ohjelmistotekniikka
Launonen, Timo-Pekka LCD-näytön laiteajuri Linux-käyttöjärjestelmässä
Tutkintotyö 26 sivua + 19 liitesivua
Työn ohjaaja Lehtori Tony Torp
Työn teettäjä Elektrobit Oy, valvojana Jussi Kallioniemi
Kesäkuu 2006
Hakusanat laiteajuri, Linux, LCD, rajapinta

TIIVISTELMÄ

Tietokoneohjelmat eivät pysty käsittelemään tietokoneiden laitteistoa suoraan käyttöjärjestelmän tekemien suojausten takia, eikä olisi kovin järkevääkään kirjoittaa jokaiseen ohjelmaan laitteiston käsittelyrutiineja. Tätä varten on olemassa jokaiselle laitteelle oma laiteajuri, joka tarjoaa ohjelmille rajapinnan laitteiston käsittelyyn.

Tässä työssä on pyritty toteuttamaan Linux-käyttöjärjestelmään yksinkertainen LCD-näytön laiteajuri, joka mahdollistaa ohjelmille näytön toimintojen käyttämisen.

Työssä on esitelty yleisiä rajapintoja Linux-käyttöjärjestelmässä ja kuvattu erään LCD-näytön rajapintoja ja toimintoja.

TAMPERE POLYTECHNIC
Computer Systems Engineering, Software Engineering
Launonen, Timo-Pekka Device driver for LCD-display on Linux-operating system
Engineering Thesis 26 pages + 19 appendices
Thesis Supervisor Tony Torp
Commissioning Company Elektrobit Oy. Supervisor: Jussi Kallioniemi
June 2006
Keywords device driver, Linux, LCD, interface

ABSTRACT

Computer programs can't access hardware directly, because of restrictions made by operating system, and it wouldn't be wise to implement hardware routines to every software. That's why there is a device driver for every piece of hardware, which offers interface for software to access hardware.

In this work is aspired to create simple device driver for LCD display on Linux operating system which makes it possible for software to use displays functions.

There are introduced common Linux interfaces in this work and described certain LCD displays interfaces and functions.

TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikka, ohjelmistotekniikka
Timo-Pekka Launonen

Sisältö

KÄYTETYT LYHENTEET

BPP	Bits Per Pixel, bittiä per pikseli
CS	Chip Select, piirin valinta-signaali
DE	Data Enable, data saatavilla-signaali
HSYNC/HS	Horizontal Sync, horisontaalinen synkronointisignaali
LCD	Liquid Crystal Display, nestekidenäyttö
PCLK	Pixel Clock, pikselikello
SCL	Serial Clock, sarjaväylän kellosignaali
SDA	Serial Data, sarjaväylän datasignaali
SPI	Serial Peripheral Interface, sarjamuotoinen dataväylä
VSYNC/VS	Vertical Sync, vertikaalinen synkronointisignaali

1. JOHDANTO

Nykyisillä prosessoreilla on vähintään kaksi suojaustasoa ja joillain, kuten x86-perheellä, on useita suojaustasoja. Unix-järjestelmissä ydin ajetaan ylimmällä tasolla, missä kaikki on sallittua ja ohjelmat ajetaan alimmalla tasolla, missä käyttöjärjestelmä rajoittaa laitteiston ja muistin käyttöä. Näistä tasoista käytetään yleisesti nimityksiä kernel space ja user space.

Suojaustasojen tarkoitus on suojata laitteistoa ei-luotetuilta ohjelmilta. Kernel spacessa toimivia ohjelmia pidetään luotettavina ja niille sallitaan laitteiston suora käyttö. Kuitenkin user space-ohjelmilla saattaa olla tarvetta käsitellä laitteistoa, joten käyttöjärjestelmän ydin tarjoaa järjestelmäkutsuja tätä varten. Systeemikutsut ovat osa ydintä, joten ne ajetaan kernel spacessa ja täten voivat käsitellä laitteistoa, mutta niitä voidaan kutsua user spacesta käsin. Käyttöjärjestelmän ydin tarjoaa yleisiä systeemikutsuja ja laiteajurit laitekohtaisia rajapintoja user space-ohjelmille.

Työssä toteutetaan LCD-näyttömoduulille laiteajuri Linux 2.6-ytimelle OMAP-mikrokontrollerilla toimivalle laitealustalle. Laiteajuri asettaa näytön toimintakuntoon ja sen avulla voidaan käyttää näyttömoduulin eri ominaisuuksia user space-ohjelmista.

2. LINUX

Linux-käyttöjärjestelmä kehitettiin alun perin pieneksi Unix-tyyppisten käyttöjärjestelmien kopioksi, mutta siitä on vähitellen kasvanut yksi varteenotettavimmista ja monipuolisimmista Unix-käyttöjärjestelmistä. Linux-käyttöjärjestelmä koostuu Linux-ytimestä ja GNU-projektin käyttöjärjestelmäalustasta ja joukosta muita vapaita ohjelmia. /3/

Linux-käyttöjärjestelmä on nimetty Linus Torvaldsin kehittämän Linux-ytimen

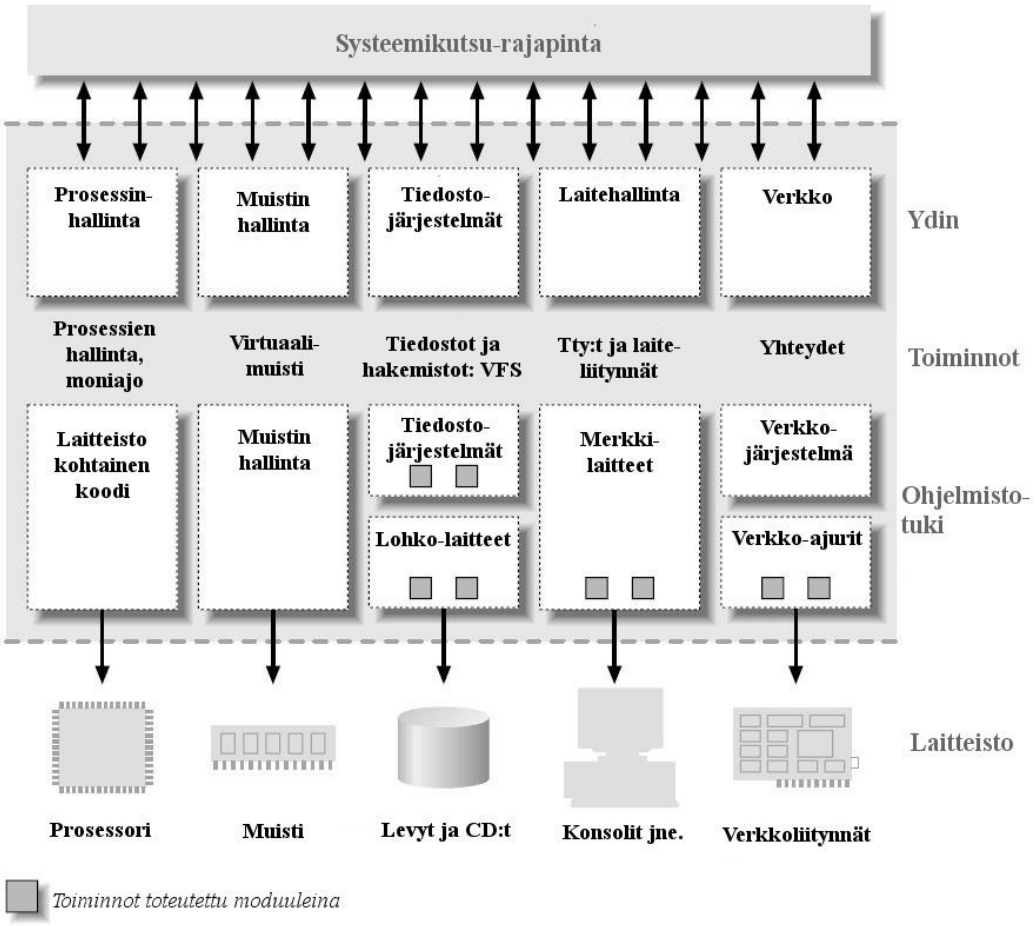
mukaan. Nimitystä Linux käytetään yleisesti tarkoittaen kokonaista käyttöjärjestelmää oheisohjelmineen joita ajetaan Linux ytimen päällä. Nimi Linux viittaa kuitenkin pelkkään ytimeen, ja kokonaista käyttöjärjestelmää pitäisi joidenkin mielestä kutsua nimellä GNU/Linux käytettävien ohjelmien ja ytimen mukaan. Linux-nimellä levitetään useita valmiiksi paketoituja ohjelmakokoelmia, joita kutsutaan levitysversioiksi, jakeluiksi, distribuutioiksi tai tuttavallisesti distroiksi (engl. Distribution). /3/

Linus Torvalds johtaa edelleenkin ytimen kehitystyötä, mutta ei osallistu levitysversioiden tai niihin sisältyvien ohjelmien kehittämiseen. Ydintä kehittää satoja ihmisiä ympäri maapalloa ja sillä on satojatuhansia aktiivisia käyttäjiä. Testaajien suuresta määrästä johtuen ytimen virheet tulevat esille hyvinkin nopeasti ja aktiivisten kehittäjien ansiosta ne yleensä korjataan muutamassa tunnissa.

Linux julkaistaan GPL-lisenssin alaisena eli sitä voi käyttää ja kopioida vapaasti. Linuxia saa myös muokata vapaasti, mutta julkaistessa muutokset on julkaistava muutosten lähdekoodikin. Linuxia saa myös myydä, mutta silloinkin on lähdekoodi julkaistava. Lähdekoodia ei ole pakko toimittaa kopion mukana, mutta se on toimitettava pyynnöstä.

Linux-ydin on modulaarinen monoliittinen ydin. Monoliittisessa ytimessä koko ydin toimii yhdessä prosessissa. Modulaarisuuden ansiosta ytimeen voidaan lisätä ja siitä voidaan poistaa osia ajon aikana (moduuleja). Monoliittisten ytimien heikkoutena on se, että esimerkiksi yksi virhetilanne ytimessä saattaa kaataa koko ytimen. Monoliittisen ytimen vahvuutena on sen nopeus verrattuna muun tyyppisiin ytimiin.

Unix-järjestelmissä useat yhtäaikaiset prosessit hoitavat eri asioita. Jokainen prosessi tarvitsee resursseja, kuten prosessoriaikaa, muistia, verkkoyhteyksiä tai muita resursseja. Ydin on iso paketti ajettavaa koodia joka hoitaa kaikkien resurssien hallinnan. Vaikka ytimen eri toimintojen erottelu ei aina ole selvää, voidaan ytimen toiminnot jakaa kuvan 1 mukaisiin lohkoihin. /1, s.4/



Kuva 1: Ytimen lohkokaavio /1, s. 6/

Prosessien hallinta

Ydin hoitaa prosessien luomisen ja tuhoamisen sekä niiden syöte- ja tulostus-liitynnät. Ydin hoitaa myös eri prosessien välisen kommunikaation. Lisäksi skeduleri joka kontrolloi miten prosessit saavat prosessoriaikaa on osa prosessien hallintaa. Lyhyesti ytimen prosessien hallinta hoitaa usean prosessin toiminnan yhden tai useamman prosessorin päällä. /1, s. 4/

Muistin hallinta

Tietokoneen muisti on tärkeä resurssi ja sen käyttötavalla on suuri vaikutus järjestelmän suoritustehoon. Ydin luo virtuaalisen osoiteavaruuden käytössä olevien rajoitettujen resurssien yläpuolelle. Ytimen eri osat käyttävät muistin hallinta-järjestelmää usean funktiokutsun kautta, yksinkertaisista malloc- ja free-

funktioista paljon monimutkaisempiin toimintoihin. /1, s. 4/

Tiedostojärjestelmät

Unix-järjestelmät perustuvat paljolti tiedostoihin, melkein kaikkea Unixeissa voidaan kohdella tiedostoina. Ydin luo rakenteellisen tiedostojärjestelmän laitteiston päälle jota käytetään paljolti koko järjestelmässä. Lisäksi Linux tukee useita tiedostojärjestelmiä, jotka ovat erilaisia tapoja järjestää data fyysisellä laitteella. Esimerkiksi levyt voidaan alustaa Linuxin standardilla ext3 tiedostojärjestelmällä tai yleisemmin käytetyllä FAT tiedostojärjestelmällä tai usealla muulla. /1, s. 4/

Laitehallinta

Lähes jokainen järjestelmän toiminto liittyy lopulta johonkin fyysiseen laitteeseen. Kaikki laitteiston käsittelyyn liittyvät operaatiot hoitaa koodi joka on spesifistä kyseiselle laitteelle. Tätä koodia kutsutaan laiteajuriksi. Jokaisen järjestelmän laitteen laiteajuri kovalevystä näppäimistöön ja nauha-asemaan pitää olla sisällytettynä ytimeen. /1, s. 5/

Verkko

Käyttöjärjestelmän tulee hoitaa verkkoyhteydet, koska saapuvat paketit ovat asynkronisia. Paketit täytyy kerätä, tunnistaa ja toimittaa ennen kuin prosessit saavat ne käyttöönsä. Järjestelmä on vastuussa datapakettien toimittamisesta ohjelmien ja verkkoliitäntöjen välillä ja sen täytyy kontrolloida ohjelmien ajoaikaa niiden verkkoaktiivisuuden perusteella. Lisäksi kaikki reititys ja osoitteiden selvitys hoidetaan ytimessä. /1, s. 5/

3. LAITEAJURIN SUUNNITTELU

Työssä suunnitellaan ja toteutetaan LCD-näytön laiteajuri Linux 2.6-ytimelle OMAP-mikrokontrollerilla toimivalle laitealustalle. Laiteajuri saattaa näytön toimintakuntoon ja tarjoaa rajapinnat user space-ohjelmille näyttömoduulin käyttämiseksi.

3.1. Näyttömoduulin tekniset ominaisuudet ja käyttäminen

Työssä käytettävä näyttömoduuli on LCD-tyyppinen näyttö, joka kykenee 800*480 resoluutioon 16- tai 18-bittisillä väreillä 55Hz:n virkistystaajuudella. Näytössä on LED-tyyppinen taustavalo, jota ohjataan ulkoisella DA-muuntimella.

Näyttömoduulista on olemassa useita kehitysversioita ja yksi tuotantoversio, joilla kaikilla on erilainen käynnistysprosessi. Ajuri tarkastaa näytön version ja suorittaa kyseiselle versiolle sopivan käynnistysprosessin.

3.2. RGB-liitäntä

Näyttömoduuli käyttää 16- tai 18-bittistä RGB-liitäntää, joka sisältää hsync-, vsync-, data enable ja pixelclock-signaalit sekä 18 datasignaalia.

Pixel clock-signaalia (PCLK) käytetään tahdistamaan muiden signaalien luku pixel clock-signaalin nousevalla reunalla.

Vertical synchronization-signaali (VS / VSYNC) ilmaisee näytölle uuden kuvan alkamisen. VSYNC on nolla-aktiivinen signaali ja sen tila luetaan PCLK-signaalin nousevalla reunalla.

Horizontal synchronization-signaali (HS / HSYNC) ilmaisee näytölle uuden juovan alkamisesta. HSYNC on nolla-aktiivinen signaali ja sen tila luetaan PCLK-signaalin nousevalla reunalla.

Data enable-signaali (DE) ilmaisee näytölle, että RGB-väylällä on data valmiina ja data voidaan lukea väylältä. DE-signaali on ylhäällä aktiivinen ja sen tila

luetaan PCLK-signaalin nousevalla reunalla.

Dataväylä välittää halutun kuvainformaation näytölle. Käytettäessä 18- bittisiä värejä ovat kaikki 18 datalinjaa käytössä ja kullekin värille on kuusi datalinjaa. 16-bittisillä väreillä kaksi datalinjoista ei ole käytössä ja punaiselle ja siniselle värille on viisi datalinjaa kummallekin ja vihreälle värille on kuusi. Näyttö lukee datan väylältä, kun DE-signaali on ylhäällä ja on PCLK-signaali on nousevalla reunalla.

Näytönohjainpiiri ohjaa näitä signaaleja, eikä niitä tarvitse ohjata ajurilla.

3.3. Sarjaliitäntä

Näyttömoduulissa on 3-linjainen 9-bittinen SPI-sarjaväylä, jonka kautta ajuri voi antaa näytölle komentoja. Mahdolliset komennot on kuvattu taulukossa 1.

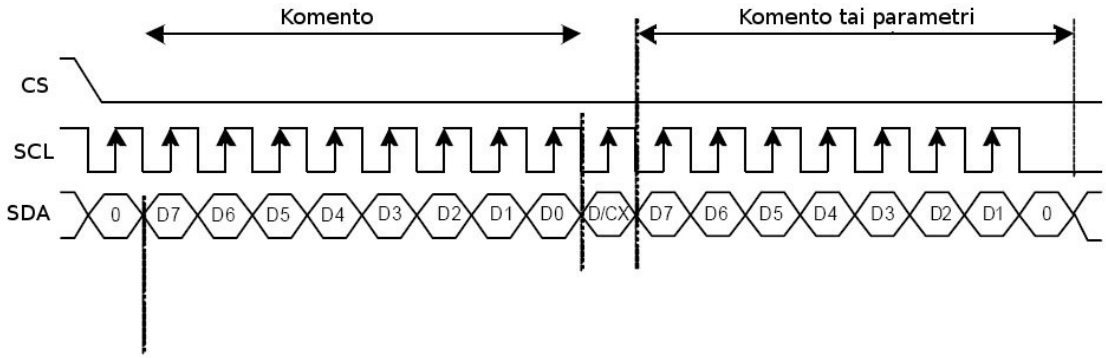
SPI-väylässä on Chip select-, kello- ja datasignaalit ja sen maksimikellotaajuus on 10MHz.
SPI-väylässä voi olla samanaikaisesti useita laitteita, jolloin tarvitaan Chip select-signaaleja (CS), joilla kerrotaan laitteille, mikä kulloinkin on aktiivisena.

Kellosignaalilla (SCL) tahdistetaan laitteiden luku- ja kirjoitustapahtumat.

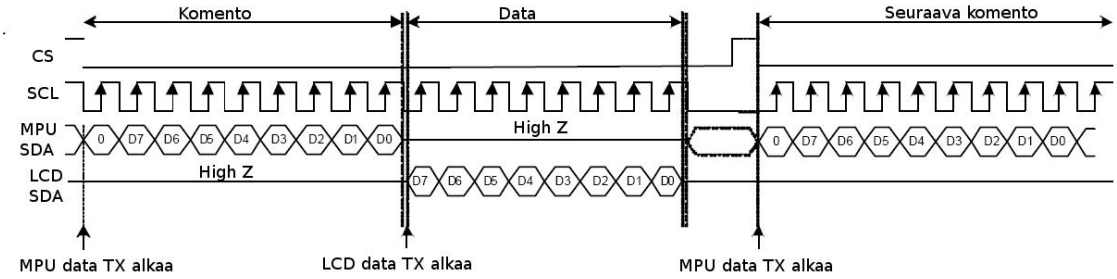
Datasignaali (SDA) on yhteinen luku- ja kirjoitustapahtumille. Datansiirto tapahtuu, kun laitteen CS-signaali on alhaalla ja SCL-signaali on nousevalla reunalla.

Näytölle kirjoitettavan datan ensimmäinen bitti kertoo, onko kyseessä komento vai dataa. Ensimmäisen bitin ollessa ”0” näyttö ymmärtää seuraavat kahdeksan bittiä komennoksi ja ensimmäisen bitin ollessa ”1” ymmärretään ne dataksi tai komennon parametreiksi.

Näytöltä luettaessa prosessori kirjoittaa näytölle käskyn ja vapauttaa datalinjan korkeaimpedanssiseen tilaan, jolloin näyttö voi alkaa kirjoittaa väylälle seuraavalla nousevalla kellopulssilla. Kirjoitettuaan väylälle näyttö vapauttaa datalinjan korkeaimpedanssiseen tilaan, jolloin prosessori voi taas kirjoittaa väylälle seuraavan komennon.
SPI-väylän kirjoitus on esitetty kuvassa 2 ja lukeminen kuvassa 3.



Kuva 2: SPI-väylälle kirjoittaminen



Kuva 3: SPI-väylältä lukeminen

SPI-väylää ohjaa erillinen ajuri, joka tarjoaa rajapinnan, jonka kautta muut ajurit voivat käyttää väylää. Rajapinta tarjoaa omat funktiot väylälle kirjoittamiseen, lukemiseen ja molempien peräkkäiseen suorittamiseen. Rajapinta käsittää seuraavat funktiot:

```
u32 omap_spi_readwritechannel(int channel, u32
*data_in, u32 *data_out, int write_read_count )
```

Funktiolla kirjoitetaan ja luetaan dataa SPI-väylältä. Parametreina annetaan SPI-

väylän numero, osoitin kirjoitettavaan dataan, osoitin puskuriin johon data luetaan ja kirjoitettavan datan pituus tavuina.

```
void omap_spi_writetochannel(int channel, u32 data)
```

Funktio kirjoitetaan dataa SPI-väylälle. Parametreina annetaan SPI-väylän numero ja kirjoitettava data.

```
u32 omap_spi_readfromchannel(int channel)
```

Funktiolla luetaan dataa SPI-väylältä. Parametrina annetaan SPI-väylän numero. Funktio palauttaa väylältä luetun datan.

Taulukko 1: Sarjaväylän kautta annettavien komentojen listaus

Komento	Funktio	Luku R / Kirjoitus W / Komento C	Parametrien määrä	Parametrit
00	NOP	C	0	-
01	Software Reset	C	0	-
06	Read Red Colour	R	1	Punaisen värin arvo näytön 1. pikselillä
07	Read Green Colour	R	1	Vihreän värin arvo näytön 1. pikselillä
08	Read Blue Colour	R	1	Sinisen värin arvo näytön 1. pikselillä
0E	Read Signal Mode	R	1	RGB väylän ohjauksen status
10	Sleep in	C	0	-
11	Sleep out	C	0	-
26	Gamma Set	W	1	1 tavu halutulle gamma-käyrälle
28	Display off	C	0	-
29	Display on	C	0	-
36	Memory Access Control	W	1	1 tavu halutulle piirtosuunnalle
3A	Interface pixel format	W	1	1 tavu halutulle BPP tiedolle

3.4. Rajapinnat user space-ohjelmille

Normaaleilla user space-ohjelmilla ei ole pääsyä kernel spaceen, jossa laiteajurit toimivat, eivätkä ne siten voi käyttää laitteistoa suoraan, joten niiden on käytettävä laitteita ajureiden kautta. Tätä varten ajurit tarjoavat sovelluksille rajapinnan.

Linuxissa on useita tapoja toteuttaa user space-rajapinta. Näistä yleisimmin

käytetty on dev-rajapinta, sen yksinkertaisen toteutuksen sekä ajurin että user space-ohjelmankin puolella. Muita mahdollisia rajapintoja on proc-rajapinta ja sysfs. Jokainen ajuri tarjoaa yleensä ainakin yhden näistä rajapinnoista. Tässä työssä toteutetaan vain dev-rajapinta.

Dev-rajapinnan kautta lähetyt viestit voivat olla yksinkertaisia komentoja tai niiden avulla voidaan välittää dataa kernel spacen ja user spacen välillä.

Device nodet ovat erikoistiedostoja, joita käytetään dev-rajapinnan tapauksessa kernel spacen ja user spacen väliseen tiedonsiirtoon. Device nodeille määrätään pää- ja alinumerot sekä tyyppi, char tai block, joiden perusteella tunnistetaan, mikä ajuri nodeen on liitetty ja minkätyyppinen se on. Perinteisesti device nodet sijaitsevat /dev-hakemistossa Linuxin tiedostohierarkiassa.

User space-ohjelman halutessa käsitellessä laitteistoa ohjelma kirjoittaa ioctl-komennon laitteen device nodeen, josta ajuri lukee pyynnön. Ajuri suorittaa komennon vaatimat toimenpiteet ja mahdollisesti palauttaa device noden kautta dataa user space-ohjelmalle.

Kaikilla ioctl-komennoilla on uniikki numero, jonka perusteella ne erotetaan toisistaan. Ioctl-komennon uniikki numero muodostetaan määriteltyjen kahden parametrin, tyypin ja numeron sekä komennon tyypin yhdistelmästä. Muodostaminen tehdään eri arkkitehtuureilla eri tavalla. Yleisesti tyyppi-parametri valitaan uniikiksi ja numero-parametri erottaa tietyn laitteen komennot toisistaan. Selvyyden vuoksi käytämme tekstissä ioctl-komennoista pelkkää numeroa.

Komennolla ”man ioctl_list” saa Linuxissa ytimen 1.3.27 listauksen kaikista sen tarjoamista ioctl-komennoista numeroineen, nimineen ja parametreineen.

Proc-tiedostojärjestelmä on ohjelmiston luoma tiedostojärjestelmä, jota ydin käyttää tiedon välittämiseen user spaceen. Proc-tiedostojärjestelmä liitetään yleensä /proc-hakemistoon Linuxin tiedostohierarkiassa. Jokainen tiedosto

/proc-hakemistossa on sidottu ytimen funktioon, joka generoi tiedoston sisällön joka kerta, kun tiedostoa luetaan. /1, s. 83 /

3.5. Laiteajurin lataaminen ja alustus

Laiteajurin lataaminen Linux-käyttöjärjestelmässä aloitetaan init-lohkosta, jossa ajuri rekisteröidään, alustetaan tarvittaessa keskeytysohjaimet, alustetaan oheislaitteet, varataan käytettävä muisti, luodaan device nodet ja user space rajapinnat.

Ajurin poistaminen taas tapahtuu exit-lohkossa, jossa kaikki init-lohkossa luodut asiat tuhotaan.

Linuxissa laiteajurit voivat olla joko ytimeen sisälle käännettyinä tai erillisinä, dynaamisesti ladattavina moduuleina. Valinta siitä, käännetäänkö ajuri ytimeen sisälle vai erilliseksi moduuliksi, tehdään ennen ytimen kääntämistä ytimen konfigurointi-työkaluilla.

Laiteajurit voivat olla riippuvaisia jostain toisesta laiteajurista, kuten esimerkiksi käytettävän väylän ajurista. Jotta ajurit ladattaisiin oikeassa järjestyksessä tarvitaan eritasoisia käynnistystasoja. Ytimen sisään käännetyt moduulit ajetaan moduulissa määritellyn initcall-tason mukaisessa järjestyksessä. Initcall-tasoja on seitsemän ja ne ajetaan järjestyksessä: core, postcore, arch, subsys, fs, device ja late. Saman initcall-tason moduulit ajetaan linkkausjärjestyksessä. Initcall-tasot on määritelty ytimen lähdekoodissa include/linux/init.h-tiedostossa.

Moduuliksi käännetty laiteajuri voidaan ladata manuaalisesti insmod- tai modprobe-komennoilla tai voidaan käyttää ytimen automaattista moduulin lataustoimintoa. Jos ytimen automaattinen moduulin lataustoiminto on otettu käyttöön, lataa ytimen moduuli-taustaohjelma ”kmod” halutun toiminnon sisältävän moduulin ”modprobe” komennolla. Modprobe-komennolle annetaan

parametrina joko moduulin nimi, tai tarvittavan device noden tyyppi ja pää- ja alinumerot. Tarvittaessa modprobe hakee moduulin nimen /etc/modprobe.conf -tiedostosta. Ennen moduulin lataamista modprobe tarkistaa moduulin riippuvuudet /lib/modules/version/modules.dep -tiedostosta ja tarvittaessa lataa muita moduuleja riippuvuuksien täyttämiseksi. Itse moduulin lataamiseksi modprobe kutsuu ”insmod” -ohjelmaa.

4. LAITEAJURIN TOTEUTUS

LCD-näytön laiteajurin tehtävä on alustaa näyttömoduuli ja saattaa se toimintakuntoon sekä tarvittaessa vaihtaa näyttömoduulin tilaa ja kertoa näytön sen hetkinen tila. Tästä syystä ajuri on yksinkertainen ja se sisältääkin pääasiassa näyttömoduulin käsittelyyn vaadittavat toiminnot ja niihin liittyvät user space rajapinnat.

4.1. Ajurin lataaminen ja poistaminen

Moduulin suoritus aloitetaan moduulin init-lohkosta, jossa konfiguroidaan SPI-väylä, vapautetaan näytön reset-signaali, alustetaan näyttö, rekisteröidään laiteajuri misc device-tyyppisenä laitteena ja sytytetään näytön taustavalo.

Moduulin init-funktio esitellään lcd.c-tiedostossa (liite 1) ”`module_init(initialise_lcd);`”. Itse init-funktio sijaitsee samassa lcd.c-tiedostossa ja on muotoa ”`static int __init initialise_lcd(void)`”.

Koska näyttömoduulista on olemassa useita erilaisia versioita ja kaikilla on erilainen käynnistysprosessi, täytyy kukin versio tunnistaa ja käsitellä eri versioita eri tavalla. Näyttömoduulin versio luetaan SPI-väylän kautta ja näyttömoduuli alustetaan version vaatimalla tavalla. Näytön version lukeminen

ja alustus tapahtuu ”`static int lcd_wakeup(void)`”-funktiossa.

Exit-funktio esitellään samalla tavalla ”`module_exit(exit_lcd);`” ja itse funktio on tyypiltään ”`static void __exit exit_lcd(void)`”
Exit-funktiossa vapautetaan SPI-väylä, vapautetaan laiteajuri ja sammutetaan näytön taustavalo. Itse näyttöä ei tarvitse erikseen sammuttaa.

4.2. dev-rajapinta

Näytön tilaa ei usein tarvitse vaihtaa näytön alustamisen jälkeen, mutta kaikkia näytön asetuksia on pystyttävä tarvittaessa muuttamaan myöhemminkin. Sen takia kaikille näytön toiminnoille on toteutettu ioctl-komento.

Ajuri tarjoaa user space-ohjelmille dev-rajapinnan näytön tilojen muuttamiseen ja nykyisen tilan kysymiseen. Ajuri on misc device-tyyppinen, joten sen device noden pää-numero on 10 ja ali-numeroksi on määritetty 156. Device noden nimeksi on määritetty ”lcd”, joten device node on /dev/lcd-niminen tiedosto. Ajuri tarjoaa taulukon 2 mukaiset ioctl-komennot user space sovelluksille.

Taulukko 2: Ajurin ioctl-komennot

ioctl	tyyppi	nro	parametrit	selite
LCD_SOFTWARE_RESET	io	0	-	Resetoi näytön
LCD_READ_RED_COLOUR	ior	2	unsigned char*	Kertoo näytön ensimmäisen pikselin punaisen värin arvon.
LCD_READ_GREEN_COLOUR	ior	3	unsigned char*	Kertoo näytön ensimmäisen pikselin vihreän värin arvon.
LCD_READ_BLUE_COLOUR	ior	4	unsigned char*	Kertoo näytön ensimmäisen pikselin sinisen värin arvon.
LCD_READ_SIGNAL_MODE	ior	10	struct lcd_signal_mode	Kertoo näytön signaalien tilan.
LCD_SLEEP_IN	io	12		Asettaa näytön sleep tilaan.
LCD_SLEEP_OUT	io	13		Palauttaa näytön sleep tilasta.
LCD_SET_GAMMA	iow	14	unsigned char	Asettaa näytölle halutun gamma-asetuksen.
LCD_DISPLAY_OFF	io	15		Sammuttaa näytön.
LCD_DISPLAY_ON	io	16		Käynnistää näytön.
LCD_SET_MADCTL	iow	17	struct lcd_mad	Asettaa näytön asetuksia.
LCD_SET_PIXEL_FORMAT	iow	18	unsigned char	Asettaa näytön 16 tai 18 bpp tilaan.
LCD_READ_ID1	ior	19	unsigned char*	Lukee näytön ROM:ltä ID1:n.
LCD_READ_ID2	ior	20	unsigned char*	Lukee näytön ROM:ltä ID2:n.
LCD_READ_ID3	ior	21	unsigned char*	Lukee näytön ROM:ltä ID3:n.
LCD_SET_BACKLIGHT	iow	22	unsigned char*	Asettaa näytön taustavalon kirkkauden.
LCD_HARDWARE_RESET	io	23		Resetoi näytön.

Ajurin ioctl-komennot määritellään lcd.h-tiedostossa. (Liite 2.) Esimerkkinä yksi kutakin käytettyä ioctl tyyppiä: pelkän komennon suorittava, kernel spacesta user spaceen dataa siirtävä ja user spacesta kernel spaceen dataa siirtävä.

```
#define LCD_SOFTWARE_RESET    _IO   (LCD_IOC_MAGIC, 0)
```

Määrittää ioctl-komennon mikä ei siirrä dataa kumpaankaan suuntaan. Komennon numero on 0.

```
#define LCD_READ_RED_COLOUR   _IOR  (LCD_IOC_MAGIC, 2,
unsigned char*)
```

Määrittää ioctl-komennon mikä siirtää dataa ajurilta user spaceen. Komennon numero on 2.

```
#define LCD_SET_GAMMA    _IOW  (LCD_IOC_MAGIC, 14,
unsigned char)
```

Määrittää ioctl-komennon mikä siirtää dataa user spacesta ajurille. Komennon numero on 14.

On olemassa myös molempiin suuntiin dataa siirtävä ioctl tyyppi, mutta sellaista ei tässä työssä tarvita. Read / write tyyppisen ioctl komennon määrittely on tyypillesesti seuraavanlainen:

```
#define IOCTL_READ_WRITE _IOWR  (LCD_IOC_MAGIC, 42,
unsigned char*)
```

Ioctl-komennot ja device noden käsittelyyn liittyvät funktion esitellään lcd.c-tiedostossa (liite 1).

Device noden avaamiseen ja sulkemiseen ja ioctl-komentojen käsittelyfunktioiden osoittimet sijoitetaan file_operations tyyppiseen structiin.

```
static struct file_operations lcd_device_fops = {
     .owner = THIS_MODULE,
```

```
        .open = lcd_open,
        .ioctl = lcd_ioctl,
        .release = lcd_release
};
```

File_operations struct ja tiedot device nodesta sijoitetaan miscdevice-tyyppiseen structiin, joka ajurin init-lohkossa rekisteröidään. Tällöin user space-ohjelman suorittaessa ioctl-komennon osaa ajuri lukea komennon device nodesta ja suorittaa komennon vaatimat toimenpiteet.

```
static struct miscdevice lcd_dev = {
        LCD_MINOR,
        LCD_MISC_NAME,
        &lcd_device_fops
};
```

Ioctl-komentojen suoritus alkaa lcd_ioctl-funktiossa, missä yksinkertaisella switch-case -rakenteella tunnistetaan mikä ioctl-komento halutaan suorittaa ja tehdään komennon vaatimat toimenpiteet.
`static int lcd_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)`

Jos komennon suorittamiseen tarvitaan datan lukemista user spacesta, tehdään se copy_from_user- tai get_user-komennoilla. Datan siirtäminen user spaceen tehdään put_user- tai copy_to_user-komennoilla.

User space-sovelluksiin, joista halutaan käyttää ioctl-komentoja, täytyy sisällyttää sys/ioctl.h-tiedosto, joka sisältää ioctl-komentojen käyttämiseen tarvittavat funktiot.

User space-ohjelmasta device node avataan kuten normaali tiedosto open-funktiolla. `int fd = open("/dev/lcd", O_RDWR);`

Jos device noden avaaminen onnistuu, voidaan heti suorittaa ioctl-komento ioctl-funktiolla, esimerkiksi: `ioctl(fd, LCD_SOFTWARE_RESET);`

Ioctl-funktion ensimmäiseksi parametriksi annetaan avatun device noden file descriptor ja toiseksi parametriksi ioctl-komento. Ioctl-funktio palauttaa nollan jos komento onnistui ja negatiivisen virhenumeron virhetilanteessa.

Ioctl-komennot on määritelty ajurin header-tiedostossa, josta ne täytyy kopioida user space-ohjelman omaan header-tiedostoon. Ytimen header-tiedostoja ei saisi suoraan sisällyttää user space-ohjelmaan.

Komentojen suorituksen jälkeen on hyvä sulkea device node normaalisti close-komennolla, jotta ajuri poistaa lukituksen ja muutkin ohjelmat voivat käyttää rajapintaa.

```
close(fd);
```

4.3. Näytön ohjaaminen

Näyttömoduulin ohjaaminen tapahtuu lähettämällä näytölle komentoja SPI-väylän kautta. Komentojen lähettämiseen käytetään SPI-väyläajurin tarjoamia rajapintoja. Mahdolliset komennot on kuvattu taulukossa 1.

Näytön alustuksen jälkeen näytön asetuksia ei juurikaan tarvitse muuttaa, mutta sähkön kulutuksen takia näyttö asetetaan usein virransäästötilaan laitteen ollessa käyttämättömänä. Esimerkkinä ajurin toiminta ohjelman lähettäessä näytölle käskyn siirtyä virransäästötilaan.

Ajurin saadessa LCD_SLEEP_IN ioctl-komennon kutsutaan ioctl-käsittelijässä lcd_spi_exec -funktiota parametrilla SLEEP_IN.

```
case LCD_SLEEP_IN:
          DPRINTK("LCD_SLEEP_IN\n");
          ret = lcd_spi_exec(SLEEP_IN);
```

```
            if (ret) {
                 DERROR("Unable to sleep in\n");
            }
            break;
```

Lcd_spi_exec-funktiossa kutsutaan SPI-väyläajurin funktiota omap_spi_writetochannel joka kirjoittaa näytön SPI-väylälle komennon SLEEP_IN eikä jää odottamaan näytöltä vastausta.

```
static int lcd_spi_exec(unsigned int command)
{
     DPRINTK("SPI exec 0x%x\n", command);
     omap_spi_writetochannel(LCD_SPI, command);

     return 0;
}
```

Näytön saatua SLEEP_IN komennon siirtyy se välittömästi virransäästötilaan. Näyttö palaa normaaliin toimintatilaan SLEEP_OUT komennolla.

5. YHTEENVETO

Työssä toteutettiin Linux-käyttöjärjestelmään yksinkertainen LCD-näytön laiteajuri ja esiteltiin yleisiä rajapintoja Linux-käyttöjärjestelmässä sekä kuvattiin erään LCD-näytön rajapintoja ja toimintoja.

Usein laiteajureilla on nopeusvaatimuksia, mutta tällä ajurilla ei niitä ole, koska näytön tilaa ei tarvitse kovinkaan usein vaihtaa sen toimintakuntoon saattamisen jälkeen. Tämän nopeutti ja helpotti ajurin tekemistä ja sen ansiosta ajuri on rakenteeltaan yksinkertainen ja selkeä.

Laiteajureiden kirjoittamisesta Linux-käyttöjärjestelmälle on saatavilla runsaasti englanninkielisiä ohjeita ja dokumentteja. Ytimen lähdekoodin Documentation-hakemistossa on hyvät ohjeet alkuun pääsemiseksi ja internetistä löytyy useita esimerkkiajureita hyvin kommentoituna ja selostettuna. Postituslistoilta ja internetin keskustelupalstoilta saa myös runsaasti apua vaikeampiinkin ongelmiin.

Tutkintotyössä on hyvää pohjatietoa Linux ajureiden kirjoittamista suunnitteleville. Vaikka kaikkien laitteiden käyttäminen on laitekohtaisia, on jokaisella ajurilla samantyyppinen perusrakenne ja rajapinnat ovat samankaltaisia.

LÄHTEET

1 Corbet, Jonathan – Rubini, Alessandro – Kroah-Hartman, Greg, Linux Device Drivers, Third Edition. O'Reilly.

2 Linux ytimen lähdekoodit ja dokumentaatio. http://kernel.org/

3 Linux – Wikipedia [www-sivu]. [viitattu 19.05.2006] Saatavissa: http://fi.wikipedia.org/wiki/Linux-käyttöjärjestelmä

LIITTEET

Liite 1: lcd.c

Liite 2: lcd.h

```
/******************************** HEADERS
*********************************/
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/seq_file.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/miscdevice.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/spinlock.h>
#include <asm/uaccess.h>
#include <linux/poll.h>
#include <linux/omap-mcspi.h>
#include <linux/touch_ad78.h>
#include <linux/io_exp_max.h>
#include <linux/delay.h>
#include <linux/fb.h>
#include <linux/spud_pwrctrl.h>

#include "lcd.h"
#include "omapfb.h"

#include "lcd_revision.h"     /* automatically generated */

/****************************** DEFINITIONS
*********************************/

#define LCD_IS_OPEN           0x01  /* /dev/lcd is in use */

#define LCD_SPI         2

#define LCD_GAMMA_CURVE_1     0
#define LCD_GAMMA_CURVE_2     1
#define LCD_GAMMA_CURVE_3     2
#define LCD_GAMMA_CURVE_4     3

/* Debugging on/off */
//#define DEBUG
#undef DEBUG
#ifdef DEBUG
#undef DPRINTK
#define DPRINTK(fmt,arg...) \
      printk (KERN_INFO "%s->%s(): " fmt, THIS_MODULE->name,
__FUNCTION__, ##arg)
#else
#define DPRINTK(fmt,arg...) \
      do { } while (0)
#endif                        /* DEBUG */

/* Error debugging on/off */
//#define ERROR_DEBUG
#undef ERROR_DEBUG
#ifdef ERROR_DEBUG
#define DERROR(fmt,arg...) \
      printk (KERN_ERR "%s->%s(): " fmt, THIS_MODULE->name,
__FUNCTION__, ##arg)
```

```
#else
#define DERROR(fmt,arg...) \
      do { } while (0)
#endif                          /* ERROR_DEBUG */

/************************ LOCAL FUNCTION DECLARATIONS
************************/
static int lcd_open(struct inode *inode, struct file *filp);
static int lcd_ioctl(struct inode *inode, struct file *filp,
                  unsigned int cmd, unsigned long arg);
static int lcd_release(struct inode *inode, struct file *filp);
static int lcd_read_signal_mode(struct lcd_signal_mode *signal);
static int lcd_set_gamma(unsigned char gamma);
static int lcd_set_mad(struct lcd_mad mad);
static int lcd_set_pixel_format(unsigned char format);
static int lcd_set_backlight(unsigned char backlight);

static int lcd_spi_read(unsigned int command, u32* arg, int
len);
static int lcd_spi_write(unsigned int command, u32 arg, int
len);
static int lcd_spi_exec(unsigned int command);
static int lcd_hwreset(void);
static int lcd_wakeup(void);

/***************************** LOCAL VARIABLES
******************************/

/* Device file operations */
static struct file_operations lcd_device_fops = {
      .owner = THIS_MODULE,
      .open = lcd_open,
      .ioctl = lcd_ioctl,
      .release = lcd_release
};

/* Misc device */
static struct miscdevice lcd_dev = {
      LCD_MINOR,
      LCD_NAME,
      &lcd_device_fops
};

spinlock_t lcd_lock = SPIN_LOCK_UNLOCKED;

extern spinlock_t lcd_lock;
static unsigned long lcd_status = 0;      /* open/closed */

/***************************** LOCAL FUNCTIONS
******************************/
/*
 *  lcd_open
 *
 *  Device open.
 *
 *  @param  inode device inode structure
 *  @param  filp  device file structure
 *
```

```
 *  @return 0 always
 *
 */
static int lcd_open(struct inode *inode, struct file *filp)
{
      spin_lock(&lcd_lock);
      if (lcd_status & LCD_IS_OPEN) {
            spin_unlock(&lcd_lock);
            DERROR("LCD already open\n");
            return -EBUSY;
      }
      lcd_status |= LCD_IS_OPEN;
      spin_unlock(&lcd_lock);

      return 0;
}

/*
 *  lcd_ioctl
 *
 *  Device ioctls.
 *
 *  @param  inode device inode structure
 *  @param  filp  device file structure
 *  @param  cmd   ioctl command
 *  @param  arg   ioctl argument
 *
 *  @return 0 OK, < 0 otherwise
 *
 */
static int lcd_ioctl(struct inode *inode, struct file *filp,
                  unsigned int cmd, unsigned long arg)
{
      int ret;

      struct lcd_mad mad;
      struct lcd_signal_mode signal_mode;
      unsigned char temp;

      DPRINTK("\n");

      switch (cmd) {
      case LCD_SOFTWARE_RESET:
            DPRINTK("LCD_SOFTWARE_RESET\n");
            ret = lcd_spi_exec(RESET);

            if (ret) {
                  DERROR("Unable to software reset\n");
            }
            break;
      case LCD_READ_RED_COLOUR:
            DPRINTK("LCD_READ_RED_COLOUR\n");
            lcd_spi_read(READ_RED, (u32*) &temp, 8);

            ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
            if (ret) {
                  DERROR
```

```
                        ("Unable to copy red colour to user
space\n");
                }
                break;

        case LCD_READ_GREEN_COLOUR:
                DPRINTK("LCD_READ_GREEN_COLOUR\n");
                lcd_spi_read(READ_GREEN, (u32*) &temp, 8);

                ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
                if (ret) {
                      DERROR
                          ("Unable to copy green colour to user
space\n");
                }
                break;

        case LCD_READ_BLUE_COLOUR:
                DPRINTK("LCD_READ_BLUE_COLOUR\n");
                lcd_spi_read(READ_BLUE, (u32*) &temp, 8);

                ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
                if (ret) {
                      DERROR
                          ("Unable to copy blue colour to user
space\n");
                }
                break;

        case LCD_READ_SIGNAL_MODE:
                DPRINTK("LCD_READ_SIGNAL_MODE\n");

                ret = lcd_read_signal_mode(&signal_mode);
                if (ret) {
                      DERROR("Unable to read signal mode\n");
                }

                ret = copy_to_user((struct lcd_signal_mode *) arg,
                                   &signal_mode,
                                   sizeof(struct lcd_signal_mode))
                    ? -EFAULT : 0;

                if (ret) {
                      DERROR("Unable to copy signal mode to user
space\n");
                }
                break;

        case LCD_SLEEP_IN:
                DPRINTK("LCD_SLEEP_IN\n");
                ret = lcd_spi_exec(SLEEP_IN);

                if (ret) {
                      DERROR("Unable to sleep in\n");
                }
                break;
```

```
		case LCD_SLEEP_OUT:
			DPRINTK("LCD_SLEEP_OUT\n");
			ret = lcd_spi_exec(SLEEP_OUT);

			if (ret) {
				DERROR("Unable to sleep out\n");
			}
			break;

		case LCD_SET_GAMMA:
			DPRINTK("LCD_SET_GAMMA\n");

			temp = (unsigned char)arg;
			DPRINTK("temp %d\n", (int)temp);

			ret = lcd_set_gamma(temp);
			if (ret) {
				DERROR("Unable to set gamma\n");
			}

			break;

		case LCD_DISPLAY_OFF:
			DPRINTK("LCD_DISPLAY_OFF\n");

			ret = lcd_spi_exec(DISPLAY_OFF);

			if (ret) {
				DERROR("Unable to set display off\n");
			}

			if (lcd_set_backlight(0x0) !=0) {
				DERROR("Could not set backlight\n");
			}

			break;

		case LCD_DISPLAY_ON:
			DPRINTK("LCD_DISPLAY_ON\n");
			ret = lcd_spi_exec(DISPLAY_ON);

			if (ret) {
				DERROR("Unable to set display on\n");
			}

			if (lcd_set_backlight(0xA0) !=0) {
				DERROR("Could not set backlight\n");
			}
			break;

		case LCD_SET_MADCTL:
			DPRINTK("LCD_MEMORY_ACCESS_CONTROL\n");

			ret = copy_from_user(&mad, (struct lcd_mad *) arg,
					sizeof(struct lcd_mad)) ? -EFAULT :
0;
```

```
            if (ret) {
                  DERROR("Unable to copy MAD from user
space\n");
            }

            ret = lcd_set_mad(mad);
            if (ret) {
                  DERROR("Unable to set MAD\n");
            }
            break;

      case LCD_SET_PIXEL_FORMAT:
            DPRINTK("LCD_SET_PIXEL_FORMAT\n");

            ret = get_user(temp, (unsigned char *) arg) ?
-EFAULT : 0;
            if (ret) {
                  DERROR
                      ("Unable to copy pixel format from user
space\n");
            }

            ret = lcd_set_pixel_format(temp);
            if (ret) {
                  DERROR("Unable to set pixel format\n");
            }
            break;

      case LCD_READ_ID1:
            DPRINTK("LCD_READ_ID1\n");
            lcd_spi_read(READ_ID1, (u32*) &temp, 8);

            ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
            if (ret) {
                  DERROR
                      ("Unable to copy display id1 to user
space\n");
            }
            break;

      case LCD_READ_ID2:
            DPRINTK("LCD_READ_ID2\n");
            lcd_spi_read(READ_ID2, (u32*) &temp, 8);

            ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
            if (ret) {
                  DERROR
                      ("Unable to copy display id2 to user
space\n");
            }
            break;

      case LCD_READ_ID3:
            DPRINTK("LCD_READ_ID3\n");
            lcd_spi_read(READ_ID3, (u32 *) &temp, 8);
```

```
            ret = put_user(temp, (unsigned long *) arg) ?
-EFAULT : 0;
            if (ret) {
                  DERROR
                      ("Unable to copy display id3 to user
space\n");
            }
            break;

      case LCD_SET_BACKLIGHT:
            DPRINTK("LCD_SET_BACKLIGHT\n");

            temp = (unsigned char)arg;

            DPRINTK("temp %d\n", (int)temp);

            ret = lcd_set_backlight((unsigned char)temp);
            if (ret) {
                  DERROR("Unable to set backlight\n");
            }
            break;

      case LCD_HARDWARE_RESET:
            DPRINTK("LCD_HARDWARE_RESET\n");

            lcd_hwreset();

            break;

      default:
            DERROR("Unknown ioctl command\n");
            return -ENOTTY;
      }

      return 0;
}

/*
 *  lcd_hwreset
 *
 *  Hardware reset display.
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_hwreset(void)
{
      int ret;

      ret = io_exp_max_resx(CLEAR);
      if (ret) {
            DERROR("Unable to hardware reset\n");
      }

      udelay(500);
      ret = io_exp_max_resx(SET);
      if (ret) {
            DERROR("Unable to hardware reset\n");
```

```
	}

	udelay(500);

	return ret;
}

/*
 *  mcspi_write_channel
 *
 *  Writes data to SPI channel
 *
 *  @param  channel           channel number to write
 *  @param  word_len    length of word to write
 *  @param  buffer            pointer to buffer
 *  @param  len         length of buffer
 *
 *  @return 0 always
 *
 */
static int mcspi_write_channel(int channel, int word_len, u32
*buffer, int len)
{
	int i;

	omap_spi_wordl(MCSPI_CHCONF_WL9);

	for (i = 0; i < len; i++)
	{
		omap_spi_writetochannel(channel, buffer[i]);
	}

	return 0;
}

/*
 *  mcspi_read_channel
 *
 *  Reads data from SPI channel
 *
 *  @param  channel           channel number to read from
 *  @param  word_len    length of word to read
 *  @param  buffer            pointer to buffer
 *  @param  len         length of buffer
 *
 *  @return 0 always
 *
 */
static int mcspi_read_channel(int channel, int word_len, u32
*buffer, int len)
{
	int i;

	omap_spi_wordl(MCSPI_CHCONF_WL9);

	for (i = 0; i < len; i++)
	{
		buffer[i] = omap_spi_readfromchannel(channel);
```

```
	}

	return 0;
}

/*
 *  lcd_wakeup
 *
 *  LCD wakeup sequence
 *
 *  @return 0 always
 *
 */
static int lcd_wakeup(void)
{
	u32 buffer[6];
	u32 version;

	DPRINTK("Sending Init...\n");

	udelay (1000);

	// reset
	buffer[0] = 0x01;
	mcspi_write_channel  (LCD_SPI, 9, buffer, 1);

	udelay (1000);

	// sleep out
	buffer[0] = 0x11;
	mcspi_write_channel  (LCD_SPI, 9, buffer, 1);

	udelay (1000);

	/* read id2 */
	lcd_spi_read(READ_ID2, &version, 8);
	version &= 0xff;
	DPRINTK("Version 0x%x\n", version);

	/* Send these init commands if we are using Lcd version
ES1-4 (HW1) */
	if (version == 0x0 || version == 0xff) {
		printk(KERN_INFO LCD_MISC_NAME ": ES1-4 display\n");

		buffer[0] = 0xb0;
		buffer[1] = 0x08 | (1 << 8);
		mcspi_write_channel  (LCD_SPI, 9, buffer, 2);

		buffer[0] = 0xb1;
		buffer[1] = 0x0b | (1 << 8);
		buffer[2] = 0x1c | (1 << 8);
		mcspi_write_channel  (LCD_SPI, 9, buffer, 3);

		buffer[0] = 0xb2;
		buffer[1] = 0x00 | (1 << 8);
		buffer[2] = 0x00 | (1 << 8);
		buffer[3] = 0x00 | (1 << 8);
		buffer[4] = 0x00 | (1 << 8);
```

```
                mcspi_write_channel  (LCD_SPI, 9, buffer, 5);

                buffer[0] = 0xb3;
                buffer[1] = 0x00 | (1 << 8);
                buffer[2] = 0x00 | (1 << 8);
                buffer[3] = 0x00 | (1 << 8);
                buffer[4] = 0x00 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 5);

                buffer[0] = 0xb4;
                buffer[1] = 0x87 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 2);

                buffer[0] = 0xb5;
                buffer[1] = 0x37 | (1 << 8);
                buffer[2] = 0x07 | (1 << 8);
                buffer[3] = 0x37 | (1 << 8);
                buffer[4] = 0x06 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 5);

                buffer[0] = 0xb6;
                buffer[1] = 0x64 | (1 << 8);
                buffer[2] = 0x24 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 3);

                buffer[0] = 0xb7;
                buffer[1] = 0x90 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 2);

                buffer[0] = 0xb8;
                buffer[1] = 0x10 | (1 << 8);
                buffer[2] = 0x11 | (1 << 8);
                buffer[3] = 0x20 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 4);

                buffer[0] = 0xb9;
                buffer[1] = 0x31 | (1 << 8);
                buffer[2] = 0x02 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 3);

                buffer[0] = 0xba;
                buffer[1] = 0x04 | (1 << 8);
                buffer[2] = 0xa3 | (1 << 8);
                buffer[3] = 0x9d | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 4);

                buffer[0] = 0xbb;
                buffer[1] = 0x15 | (1 << 8);
                buffer[2] = 0xb2 | (1 << 8);
                buffer[3] = 0x8c | (1 << 8);
                buffer[4] = 0x00 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 5);
        } else if (version >= 0x90) {
                /* Send these init commands if we are using Lcd
version ES2-3 and above (HW2) */
                printk(KERN_INFO LCD_MISC_NAME ": ES2-3 or newer
display\n");
                buffer[0] = 0xc2;
```

```
                buffer[1] = 0x02 | (1 << 8);
                buffer[2] = 0x00 | (1 << 8);
                buffer[3] = 0x00 | (1 << 8);
                mcspi_write_channel  (LCD_SPI, 9, buffer, 4);
        } else {
                printk(KERN_INFO LCD_MISC_NAME ": Unknown
display\n");
        }

        buffer[0] = 0x3a;
        buffer[1] = 0x60 | (1 << 8); /*18 bits / pixel*/
        mcspi_write_channel  (LCD_SPI, 9, buffer, 2);

        udelay (1000);
        DPRINTK("Enable LCD...\n");

        memset (buffer, sizeof(buffer), 0);
        buffer[0] = 0x29;
        mcspi_write_channel  (LCD_SPI, 9, buffer, 1);

        udelay (1000);

        return 0;
}

/*
 *  lcd_spi_read
 *
 *  Reads data from SPI channel
 *
 *  @param  command     ID of command to execute, defined in
lcd.h
 *  @param  arg   argument
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_spi_read(unsigned int command, u32* arg, int len)
{
        u32 temp = 0;

        omap_spi_wordl(MCSPI_CHCONF_WL18);
        temp = (command & 0xff) << 9;

        omap_spi_readwritechannel(LCD_SPI, &temp, arg, 1);
        *arg = (*arg >> 1) & 0xff;
        DPRINTK("SPI read 0x%x 0x%x\n", command, *arg);

        return 0;
}


/*
 *  lcd_spi_write
 *
 *  Writes data to SPI channel
 *
 *  @param  command     ID of command to execute, defined in
```

lcd.h

```
 *  @param  arg   argument
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_spi_write(unsigned int command, u32 arg, int len)
{
      u32 buffer[6];

      omap_spi_wordl(MCSPI_CHCONF_WL9);
      buffer[0] = command;
      buffer[1] = arg | (1 << 8);
      mcspi_write_channel  (LCD_SPI, 9, buffer, 2);
      DPRINTK("SPI write 0x%x 0x%x\n", command, arg);

      return 0;
}

/*
 *  lcd_spi_exec
 *
 *  Writes data to SPI channel
 *
 *  @param  command     ID of command to execute, defined in
lcd.h
 *
 *  @return 0 always
 *
 */
static int lcd_spi_exec(unsigned int command)
{
      DPRINTK("SPI exec 0x%x\n", command);
      omap_spi_writetochannel(LCD_SPI, command);

      return 0;
}

/*
 *  lcd_read_signal_mode
 *
 *  Reads displays signal mode
 *
 *  @param  signal      signal mode struct
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_read_signal_mode(struct lcd_signal_mode *signal)
{
      int ret;
      u32 param = 0;

      ret = lcd_spi_read(READ_SIGNAL_MODE, &param, 8);
      if (ret) {
            DERROR("Unable to execute SPI command\n");
            return ret;
      }
```

```
	signal->tearing_effect_line = param >> 7 & 1;
	signal->tearing_effect_output_line = param >> 6 & 1;
	signal->h_sync = param >> 5 & 1;
	signal->v_sync = param >> 4 & 1;
	signal->pix_clock = param >> 3 & 1;
	signal->data_enable = param >> 2 & 1;

	return ret;
}

/*
 *  lcd_set_gamma
 *
 *  Set displays gamma curve
 *
 *  @param  gamma gamma curve
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_set_gamma(unsigned char gamma)
{

	int ret;
	u32 param = 0;

	param = gamma;

	ret = lcd_spi_write(SET_GAMMA, param, 8);
	if (ret) {
		DERROR("Unable to execute SPI command\n");
		return ret;
	}

	return ret;
}

/*
 *  lcd_set_mad
 *
 *  Set displays Memory data access control
 *
 *  @param  mad   Memory data access control
 *
 *  @return 0 on success <0 on error
 *
 *  @see    lcd_read_mad()
 */
static int lcd_set_mad(struct lcd_mad mad)
{
	int ret;
	u32 param = 0;

	param = mad.page_address_order << 7 |
mad.column_address_order << 6
		| mad.rgb_bgr_order << 3;
```

```
	ret = lcd_spi_write(SET_MADCTL, param, 8);
	if (ret) {
		DERROR("Unable to execute SPI command\n");
		return ret;
	}

	return ret;
}

/*
 *  lcd_set_pixel_format
 *
 *  Set displays pixel format
 *
 *  @param  format      pixel format
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_set_pixel_format(unsigned char format)
{
	int ret;
	u32 param = 0;

	param = format;

	ret = lcd_spi_write(SET_PIXEL_FORMAT, param, 8);
	if (ret) {
		DERROR("Unable to execute SPI command\n");
		return ret;
	}

	return ret;
}

/*
 *  lcd_set_backlight
 *
 *  Set displays backlight
 *
 *  @param  backlight   intensity of backlight
 *
 *  @return 0 on success <0 on error
 *
 */
static int lcd_set_backlight(unsigned char backlight)
{
	int ret;

	DPRINTK("Setting backlight to %d\n", (int)backlight);

	if (machine_is_omap_spud_hw1() ||
machine_is_omap_spud_hw2()){
		if (backlight) {
			touch_ad78_gpio4(1);
		} else {
			touch_ad78_gpio4(0);
		}
```

```
	} else if (machine_is_omap_spud_hw2_1()) {
		if (backlight == 0) {
			touch_ad78_gpio4(1);
		} else {
			touch_ad78_gpio4(0);
		}
	} else {
		DERROR("Unrecognized HW\n");
		return 1;
	}

	ret = touch_ad78_backlight(backlight);
	if (ret) {
		DERROR("Unable to set backlight\n");
		return ret;
	}

	return ret;
}

/*
 *  lcd_release
 *
 *  Device release.
 *
 *  @param  inode device inode structure
 *  @param  filp  device file structure
 *
 *  @return 0 always
 *
 */
static int lcd_release(struct inode *inode, struct file *filp)
{
	spin_lock(&lcd_lock);
	lcd_status &= ~LCD_IS_OPEN;
	spin_unlock(&lcd_lock);

	return 0;
}

/*
 *  initialise_lcd
 *
 *  Standard module initialisation.
 *
 *  @param  None
 *
 *  @return 0 OK, < 0 otherwise
 *
 */
static int __init initialise_lcd(void)
{
	int ret;
	struct fb_info *info = registered_fb[0];
	struct spi_channel_config spi_config;

	printk(KERN_INFO LCD_MISC_NAME ": Lcd LCD driver v %s\n",
LCD_REVISION);
```

```
	/* config SPI*/
	spi_config.mode = MASTER;                        /*
MAster/Slave */
	  spi_config.endianess = MCSPI_MODULCTRL_LITTLEEND;  /*
little endian */
	  spi_config.transmitreceive = MCSPI_CHCONF_TRANSRECEIVE;
	/* Receive/transmit/both */
	  spi_config.wordlength = MCSPI_CHCONF_WL18;         /*
Length of the word to be transmitted */
	  spi_config.spipolarity = MCSPI_CHCONF_EPOL_LOW ;   /*
Polaroty of SPIEN */
	  spi_config.clkphase = MCSPI_CHCONF_PHA_ODD;        /*
clock phase */
	  spi_config.clkpolarity = MCSPI_CHCONF_POL_HIGH;
	/* clock polarity */
	  spi_config.clkdivisor = MCSPI_CHCONF_CLKD_4;       /*
12MHz */

	omap_spi_channelconfig(LCD_SPI, &spi_config);

	io_exp_max_resx(SET);
	udelay(500);

	ret = lcd_wakeup();
	if (ret) {
		printk(KERN_ERR "Lcd wakeup rituals didn't
succeed\n");
		return ENODEV;
	}

	/* Register misc device */
	DPRINTK("Registering miscdevice %s\n", lcd_dev.name);
	ret = misc_register(&lcd_dev);
	if (ret) {
		DERROR("Could not register %s\n", LCD_NAME);
		return ret;
	}
	DPRINTK("Created device as minor: %d\n", lcd_dev.minor);

	if (lcd_set_backlight(0xA0) !=0) {
		DERROR("Could not set backlight\n");
	}

	if (info->fbops->fb_set_par)
		info->fbops->fb_set_par(info);

	return 0;
}

/*
 *  exit_lcd
 *
 *  Standard module exit.
 *
 *  @param  None
 *
 *  @return None
```

```
 *
 */
static void __exit exit_lcd(void)
{
      DPRINTK("module exit\n");

      if (lcd_set_backlight(0) !=0) {
            DERROR("Could not set backlight\n");
      }

      omap_spi_channelfree(LCD_SPI);

      /* Unregister misc device */
      DPRINTK("Misc deregister %s\n", lcd_dev.name);
      misc_deregister(&lcd_dev);

      return;
}

module_init(initialise_lcd);
module_exit(exit_lcd);

MODULE_AUTHOR("Timo Launonen / Elektobit Oy");
MODULE_DESCRIPTION("LCD driver");

/* EOF */
```

```
#ifndef __LCD_H
#define __LCD_H

/******************************** HEADERS
***********************************/

#include <linux/ioctl.h>

/***************************** DEFINITIONS
***********************************/
#define LCD_NAME        "lcd" /* Device name in /dev */
#define LCD_MINOR       156   /* Device minor number */
#define LCD_MINORS            1     /* Number of devices */
#define LCD_MISC_NAME   "lcd"

#define LCD_IOC_MAGIC   91

#define LCD_SOFTWARE_RESET          _IO   (LCD_IOC_MAGIC, 0)
#define LCD_READ_RED_COLOUR         _IOR  (LCD_IOC_MAGIC, 2,
unsigned char*)
#define LCD_READ_GREEN_COLOUR _IOR  (LCD_IOC_MAGIC, 3, unsigned
char*)
#define LCD_READ_BLUE_COLOUR        _IOR  (LCD_IOC_MAGIC, 4,
unsigned char*)
#define LCD_READ_SIGNAL_MODE        _IOR  (LCD_IOC_MAGIC, 10,
struct lcd_signal_mode)
#define LCD_SLEEP_IN                _IO   (LCD_IOC_MAGIC, 12)
#define LCD_SLEEP_OUT         _IO   (LCD_IOC_MAGIC, 13)
#define LCD_SET_GAMMA         _IOW  (LCD_IOC_MAGIC, 14, unsigned
char)
#define LCD_DISPLAY_OFF       _IO   (LCD_IOC_MAGIC, 15)
#define LCD_DISPLAY_ON        _IO   (LCD_IOC_MAGIC, 16)
#define LCD_SET_MADCTL        _IOW  (LCD_IOC_MAGIC, 17, struct
lcd_mad)
#define LCD_SET_PIXEL_FORMAT        _IOW  (LCD_IOC_MAGIC, 18,
unsigned char)
#define LCD_READ_ID1                _IOR  (LCD_IOC_MAGIC, 19,
unsigned char*)
#define LCD_READ_ID2                _IOR  (LCD_IOC_MAGIC, 20,
unsigned char*)
#define LCD_READ_ID3                _IOR  (LCD_IOC_MAGIC, 21,
unsigned char*)
#define LCD_SET_BACKLIGHT           _IOW  (LCD_IOC_MAGIC, 22,
unsigned char*)
#define LCD_HARDWARE_RESET          _IO   (LCD_IOC_MAGIC, 23)

/* commands */
#define RESET                 0x01
#define READ_RED        0x06
#define READ_GREEN            0x07
#define READ_BLUE       0x08
#define READ_SIGNAL_MODE      0x0E
#define SLEEP_IN        0x10
#define SLEEP_OUT       0x11
#define SET_GAMMA       0x26
#define DISPLAY_OFF           0x28
```

```
#define DISPLAY_ON            0x29
#define SET_MADCTL            0x36
#define SET_PIXEL_FORMAT      0x3A
#define READ_ID1        0xDA
#define READ_ID2        0xDB
#define READ_ID3        0xDC

struct lcd_mad{
      unsigned char page_address_order;
      unsigned char column_address_order;
      unsigned char page_column_order;
      unsigned char line_address_order;
      unsigned char rgb_bgr_order;
      unsigned char data_latch_order;
      unsigned char switch_segment_ram;
      unsigned char switch_common_ram;
};

struct lcd_signal_mode{
      unsigned char tearing_effect_line;
      unsigned char tearing_effect_output_line;
      unsigned char h_sync;
      unsigned char v_sync;
      unsigned char pix_clock;
      unsigned char data_enable;
};

#endif                        /* __LCD_H */
 /* EOF */
```