# Vežba 9 : Stablo uređaja (*device tree*) i drajveri za platformske uređaje (*platform device drivers*)

## 1. Platformski uređaji

Od samog početka razvoja drajvera za Linux uređaje, često se dešavalo da su korisnici morali eksplicitno naznačiti kojim sve uređajima kernel ima pristup kako bi sistem kao celina radio na korektan način. U nedostatku ovih informacija, kernel ne bi znao koje I/O portove i prekidne linije neki uređaj koristi, a samim tim ne bi znao kako da rukuje njime.

Danas živimo u dobu magistrala kao što su PCIe (Peripheral Component Interconnect Express) koje imaju ugrađenu mogućnost automatskog detektovanja uređaja. To znači da će bilo koji uređaj koji je povezan preko ovakve magistrale moći da kaže ostatku sistemu o kojem se tipu uređaja radi i koje resurse zahteva. Sada kernel može da enumeriše sve uređaje na magistrali pri podizanju sistema i preuzme sve potrebne informacije od njih. PCIe magistrala se danas koristi na personalnim računarima za povezivanje perifernih uređaja koji zahtevaju visoke performance (grafičke karte, mrežne karte, akceleratori pristupa memoriji, itd). Sve moderne magistrale podržavaju enumeracuju uređaja kao i priključivanje uređaja tokom rada sistema (*hot swapping, hot plugging*). Najpoznatiji su: USB - za ulazno izlazne uređaje, ATA/SATA - za uređaje masovnog skladištenja podataka kao i HDMI, VGA, DVI, DisplayPort - za komunikaciju sa monitorima.

Ipak, i dalje postoji mnoštvo uređaja koji nisu povezani preko ovog tipa magistrala, pa ih procesor ne može sam otkriti (npr. AXI interfejs na ARM arhitekturama). Oni se zovu platformski uređaji i najviše su zastupljeni u *embedded* i *SoC* (*System on Chip*) svetu. Kernelu je i dalje potrebno da zna pojedinosti ovih uređaja, pa postoje različiti načini da se to obezbedi.

Jedan od načina je da dizajner hardvera sam detaljno opiše uređaj pomoću C koda. U tu svrhu bi se koristile strukture kao što su ***struct platform_device*** čija se definicija može naći u biblioteci <linux/platform_device.h>. Kada bi se uređaj opisivao na ovakav način, svi resursi koje uređaj zahteva (broj prekidne linije, adresni prostor registara, itd) bi bili opisani u nizu struktura ***struct resource***. Na taj niz bi pokazivao pokazivač *resource* unutar strukture *platform_device* (pogledati sledeći kodni segment).

```
struct platform_device
{
   const char *name;
   int id;
   struct device dev;
   u32 num_resources;
   struct resource *resource;
   const struct platform_device_id *id_entry;
   /* Ostala polja ovde nece biti prikazana*/
};
```

Kada bi se napravila struktura platform_device, ona bi se prosledila kernelu pomoću poziva funkcije ***int platform_device_register (struct platform_device *pdev)***.

U narednom kodnom segmentu je prikazano kako bi izgledao opis jednostavnog uređaja my_device sa adresnim prostorom koji počinje na lokaciji 0x10000000 i završava na 0x10001000, i koji zauzima prekidnu liniju sa rednim brojem 20.

```
static struct resource my_resources[] =
{
      {
            .start = 0x10000000,
            .end  = 0x10001000,
            .flags = IORESOURCE_MEM,
            .name = "io-memory"
      },
      {
            .start = 20,
            .end = 20,
            .flags = IORESOURCE_IRQ,
            .name = "irq",
      }
};

static struct platform_device my_device =
{
      .name = "my_device",
      .resource = my_resources,
      .num_resources = ARRAY_SIZE(my_resources),
};
```

Prethodno opisan sistem platformskih uređaja ima dugu istoriju i intenzivno se koristio, ali je imao veliku manu - da se ovi uređaji moraju opisati i instancirati unutar koda kernela. Nastajali su takozvani "board fajlovi" koji su sadržali detaljan opis svih uređaja koji se nalaze u sistemu, a zatim bi se kernel pomoću tih fajlova kompajlirao za tačno određenu platformu. Kada bi kernel bio kompajliran za neku određenu ploču, on ne bi radio ni na jednoj drugoj. Ovaj sistem board fajlova je funkcionisao, ali samo ukoliko nije bilo čestih promena u hardveru kao što je slučaj kod registara x86 procesora.

ARM arhitekture su postale veliki problem u ovom načinu opisivanja hardvera. Iako svi ARM procesori dele isti kompajler i slične funkcionalnosti, svaki čip baziran na ARM arhitekturi je imao jedinstvene adrese registara i malo drugačiju konfiguraciju. Sem toga, javilo se mnoštvo različitih razvojnih ploča sa ARM procesorima, gde je svaka imala svoj specifičan niz uređaja što je dovelo do nekontrolisanog povećanja board fajlova. Razvojne ploče sa FPGA čipovima su predstavljale dodatan problem, zato što bi dizajneri IP-jeva bili prisiljeni da menjaju trenutne i kreiraju nove board fajlove. Ovi board fajlovi bi postali deo izvornog koda Linux kernela. Pregršt board fajlova su uređivali različiti autori i svaka promena u nekom od njih bi rezultovala u konfliktu sa drugim. Ovakav kernel je bio jako težak za održavanje pa je Linus Tovalds preuzeo inicijativu da se pređe na sistem koji će omogućiti kompajliranje kernela za sve ARM platforme. Frustracija Linusa putem ovog pitanja je očigledna u arhiviranom mejlu: https://lkml.org/lkml/2011/3/17/492

# 2. Stablo uređaja (device tree)

Izabrano rešenje je **stablo uređaja** (*FDT - Flattened Device Tree, OF - Open Firmware*). Ovo je struktura podataka u bajt-kod formatu koja sadrži informacije o hardveru koji se nalazi na ploči. Bootloader je zadužen da kopira stablo uređaja na poznatu adresu u RAM-u, kako bi kernel imao na raspolaganju ove informacije. Iz ovoga se vidi da je glavna prednost stabla uređaja to, da je ono nezavisano od kernela, te više nije potrebno da se kernel rekompajlira kada se nešto od hardvera na ploči promeni. Ovo je od velikog značaja za rad sa FPGA uređajima gde se programabilna logika često rekonfiguriše. Na osnovu informacija koje su mu prosleđene pomoću stabla uređaja, kernel će znati da poveže uređaje sa odgovarajućim drajverima, a drajveri će pomoću specijalizovanih funkcija moći da preuzmu sve neophodne podatke iz stabla uređaja. Više o tome kasnije.

Stablo podataka se može naći u tri forme:

- Tekstualni (izvorni) fajl *.dts (*device tree source*)
- Binarni (objektni) fajl *.dtb (*device tree blob)*
- Fajl sistem u okviru Linux kernela, na putanji */proc/device-tree*

U izvornim fajlovima se nalazi opis svih uređaja koji sačinjavaju sistem. Najčešće postoji više ovakvih fajlova koji opisuju jedan sistem, gde je jedan na najvšem nivou i uključuje ostale.

Kompajler stabla uređaja (*device tree compiler*) se može koristiti za dobijanje binarnog fajla iz izvornog pomoću sledeće komande, gde je *my_tree.dts* izvorni fajl na najvišem nivou hijerarhije.

***.dts** → ***.dtb***

*dtc -I dts -O dtb -o /path/to/my_tree.dtb /path/to/my_tree.dts*

Pri podizanju sistema bootloader kopira sadržaj objektnog *.dtb fajla u RAM a zatim pokrene kernel koji na osnovu tih vrednosti napravi fajl sistem na putnji /proc/device-tree. Ukoliko je za svrhe debagovanja potrebno da se iz objektnog fajla ili fajl sistema napravi izvorni fajl koji korisnik može da pročita, to se može uraditi pomoću sledećih komandi:

***.dtb** → ***.dts***

*dtc -I dtb -O dts -o /path/to/ my_tree.dts /path/to/ my_tree.dtb*

***file_system** → ***.dts***

*dtc -I fs -O dts -o ~/my_tree.dts /proc/device-tree/*

Za svrhe demonstracije stabla uređaja kao i struktura karakterističnih za platformske drajvere, koristiće se primer **gpio** IP-ja koji kontroliše ugrađene **LED** na Zybo razvojnoj ploči. U sledećem kodnom listingu se može videti opis ovog IP-ja unutar izvornog koda stabla uređaja. Iz koda se može primetiti da je uređaj priključen na virtuelnu magistralu *amba_pl* specijalno namenjenu za uređaje u programabilnoj logici sa kojima se komunicira preko *AXI Interconnect-a*.

```
...
amba_pl
{
	compatible = "simple-bus";
	ranges;
	#adress-cells = <0x1>;
	#size-cells = <0x1>;

	gpio@41200000 {
		compatible = "led_gpio";
		xlnx,gpio2-width = <0x20>;
		xlnx,interrupt-present = <0x0>;
		xlnx,all-inputs = <0x0>;
		xlnx,dout-default = <0x0>;
		gpio-controller;
		xlnx,tri-default-2 = <0xffffffff>;
		xlnx,all-outputs = <0x1>;
		xlnx,all-inputs-2 = <0x0>;
		xlnx,dout-default-2 = <0x0>;
		xlnx,is-dual = <0x0>;
		xlnx,all-outputs-2 = <0x0>;
		xlnx,tri-default = <0xffffffff>;
		reg = <0x41200000 0x10000>;
		#gpio-cells = <0x2>;
		xlnx,gpio-width = <0x4>;
	};
}
...
```

Takođe se može primetiti da uređaj u svom opisu ima mnoštvo polja koja su automatski generisana. Nama će biti od značaja polja *compatible* i *reg,* koja su u kodu istaknuta podebljanim fontom. Polje reg govori da je adresni prostor širine 0x10000 i da počinje na adresi 0x41200000. Budući da su registri 32-bitni, oni se adresiraju sa ofsetom 4 u odnosu na baznu adresu. Dakle, prvi bi bio na adresi *0x*41*200000*, drugi na *0x*41*200004*, treći na *0x*41*200008,* itd. Polje *compatible* je string specifičan za uređaj, drugim rečima - ime uređaja. Ovo polje koristi kernel kako bi povezao uređaj sa kompatabilnim drajverom, onim koji ima identičan string u svom compatible polju.

## 3. Drajveri za platformske uređaje

Drajveri za platformske uređaje se od običnih, softverskih drajvera razlikuju samo po nekolicini struktura koje služe za registrovanje drajvera i inicijalizaciju samog uređaja.

Najvažnija struktura koja opisuje drajver za platformski uređaj je platform_driver. Njen prototip se može videti u sledećem kodu:

```
struct platform_driver
{
	struct device_driver driver;
	int (*probe)(struct platform_device *);
	int (*remove)(struct platform_device *);
	void (*shutdown)(struct platform_device *);
	int (*suspend)(struct platform_device *, pm_message_t state);
	int (*suspend_late)(struct platform_device *, pm_message_t state);
	int (*resume_early)(struct platform_device *);
	int (*resume)(struct platform_device *);
};
```

Ova struktura se sastoji od strukture device_driver i pokazivača na funkcije koje se mogu, a i ne moraju definisati. Struktura device_driver ima tri polja: *name*, *owner* i *of_match_table*. U polju name se prosledjuje string koji predstavlja ime drajvera. Polje owner predstavlja vlasnika drajvera i najčešće se postavlja na makro *THIS_MODULE*. Treće polje, *of_match_table*, je niz struktura *of_device_id* koje imaju *compatible* polje. U *compatible* poljima se navode imena svih uređaja za koje je namenjen ovaj drajver.

Ako pogledamo opis uređaja *gpio* u stablu uređaja, možemo videti da u polju *compatible* stoji string "*led_gpio*". Dakle, u drajveru za ovaj uređaj, u jednom od *compatible* polja mora da stoji isti string, kako bi kernel mogao da poveže *gpio* uređaj sa ovim drajverom. U sledećem delu koda se može videti kako izgleda definicija strukture *platform_driver* u drajveru za *gpio* uređaj. U kodu se može primetiti da struktura *device_driver* nije posebno instancirana, već su njena polja direktno dodeljena strukturi *platform_driver*.

```
static struct of_device_id led_of_match[] =
{
	{ .compatible = "led_gpio", },
	{ /* kraj niza struktura of_device_id */ },
};
```

```
static struct platform_driver led_driver =
{
	.driver =
	{
		.name = "led",
		.owner = THIS_MODULE,
		.of_match_table = led_of_match,
	},
	.probe = led_probe,
	.remove = led_remove,
};
```

U strukturi *platform_driver* su pokazivačima ***probe*** i ***remove*** dodeljene funkcije **static int led_probe (struct platform_device *pdev)** i **static int led_remove (struct platform_device *pdev)**. Probe funkcija služi da se izvrši inicijalizacija (početna podešavanja) uređaja kao i alociranje potrebnih resursa za rad sa njime. Remove funkcija služi za zaustavljanje rada uređaja kao i oslobađanje resursa zauzetih u probe funkciji.

Nakon što se u *module_init* funkciji registruje drajver pomoću funkcije ***platform_driver_register*** (kojoj se kao parametar prosleđuje prethodno opisana struktura *platform_driver*), kernel pokušava da u stablu uređaja pronađe uređaj čije se compatible polje poklapa sa jednim od navedenih u nizu *of_match_table*. Ako uspe da pronađe isti string u opisu uređaja i drajveru, to znači da je drajver pisan za uređaj koji postoji u sistemu, i poziva se probe funkcija kako bi inicijalizovala dati uređaj. Na sličan način, nakon što se u *module_exit* funkciji pozove funkcija ***platform_driver_unregister***, kernel poziva remove funkciju drajvera.

Jedini parametar *probe* i *remove* funkcija je pokazivač na strukturu *platform_device* čiji je prototip naveden u prvom poglavlju. Kada kernel pozove neku od ovih funkcija, on kao parametar prosledi *platform_device* strukturu uređaja koji će biti kontrolisan tim drajverom. Kao što je ranije pomenuto, u polju resource ove strukture se nalaze svi podaci o uređaju koji su neophodni drajveru, pa postoje različite pomoćne funkcije za izdvajanje podataka iz ove strukture kao što su:

- Preuzimanje resursa na osnovu rednog broja - ***platform_get_resource***

Kao prvi parametar se prosleđuje struktura *platform_device*, ista koja je prosleđena probe funkciji od strane kernela. Drugi parametar predstavlja tip resursa, najčešće se prosleđuje jedan od niza makroa definisanih u biblioteci <linux/ioport.h>. Treći parametar je redni broj zahtevanog resursa.

```
struct resource *platform_get_resource(struct platform_device *pdev,
unsigned int type, unsigned int n);
```

- Preuzimanje resursa na osnovu imena

Prvi i drugi parametar su isti kao i u prethodnoj funkciji dok treći predstavlja ime resursa. Retko se koristi i nije preporučeno korišćenje ove funkcije u sistemima koji se oslanjaju na stabla uređaja.

```
struct resource *platform_get_resource_byname(struct platform_device
*pdev, unsigned int type, const char *name);
```

Preuzimanje rednog broj prekida, dobijeni broj se koristi pri pozivu funkcije *request_irq*. Više o prekidima na sledećim vežbama.

```
int platform_get_irq(struct platform_device *pdev, unsigned int n);
```

### 3.1. Probe i remove funkcije LED drajvera

Kako bi se demonstrirala upotreba pomenutih funkcija, u nastavku će biti opisana probe funkcija drajvera za LED. Za kontrolu LED se koristi samo jedan registar, kome se pristupa preko bazne adrese uređaja 0x41200000. Ukoliko je u nekom od četiri najniža bita tog registra upisana logička jedinica, dioda koja odgovara tom bitu će biti upaljena. Ukoliko je vrednost bita jednaka nuli, dioda će biti ugašena. Kako bi pristupili registru koji kontroliše diode, moramo da izdvojimo njegovu fizičku adresu iz strukture platform_device. Za početak deklarišemo pomoćnu strukturu led_info koja ima tri polja:

- *mem_start* - početna fizička adresa uređaja, adresa prvog registra
- *mem_end* - krajnja fizička adresa uređaja
- *base_addr* - početna virtuelna adresa uređaja, adresa prvog registra. Ukoliko bi se koristili ostali registi, njihove adrese bi se dobijale sa inkrementom od 4 u odnosu na početnu. Makro *__iomem* služi kao pomoćni alat debagerima kernela kao što je *Sparse*, te će pri regularnom kompajliranju biti ignorisan.

```
struct led_info
{
      unsigned long mem_start;
      unsigned long mem_end;
      void __iomem *base_addr;
};
static struct led_info *lp = NULL;
```

Na početku probe funkcije se pravi pokazivač na strukturu *resource* i poziva se funkcija *platform_get_resource* sa tipom *IORESOURCE_MEM* kao parametrom. Ovaj poziv vraća strukturu koja će kao polja imati podatke o adresnom prostoru uređaja. Odmah nakon poziva funkcije se proverava da li su dobijeni podaci validni. Zatim se pomoću *kmalloc* funkcije zauzima prostor za prethodno deklarisanu pomoćnu strukturu tipa *led_info* i proverava se da li je zahtevani prostor dobijen. Funkcije kmalloc i kfree su definisane u hederu <linux/slab.h>. U *r_mem* strukturi se nalaze početna i krajnja fizička adresa uređaja, koje se sada smeštaju u pomoćnu strukturu *lp*.

```
static int led_probe(struct platform_device *pdev)
{
	struct resource *r_mem;
	int rc = 0;
	r_mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
	if (!r_mem) {
		printk(KERN_ALERT "Failed to get resource\n");
		return -ENODEV;
	}
	lp = (struct led_info *)
		kmalloc(sizeof(struct led_info), GFP_KERNEL);
	if (!lp) {
		printk(KERN_ALERT "Cound not allocate memory\n");
		return -ENOMEM;
	}
	lp->mem_start = r_mem->start;
	lp->mem_end = r_mem->end;
...
```

Nakon što su fizičke adrese smeštene u pomoćnu strukturu, potrebno je da se rezerviše dati opseg adresa, kako ih kernel ne bi dodelio u neku drugu svrhu. To se radi preko poziva funkcije ***request_mem_region*** gde se kao prvi parametar prosleđuje početna adresa, kao drugi parametar zahtevani opseg i kao treći vlasnik memorije. Ukoliko je poziv funkcije neuspešan, u pomoćnu promenljivu rc se smešta error -EBUSY , koji simbolizira da je memorijski opseg već zauzet. U tom slučaju se skače na labelu error1 koja pomoću return funkcije kornisniku vraća broj errora radi lakšeg debagovanja.

Ukoliko je zauzimanje memorije prošlo uspešno, fizičke adrese se moraju mapirati u virtuelne kako bi se moglo čitati i upisivati u te memorijske lokacije. Ovo se radi pomoću poziva funkcije ***ioremap*** koja kao povratnu vrednost vraća virtuelnu adresu koju ćemo smestiti u polje *base_addr* pomoćne strukture *lp*. Ukoliko je mapiranje bilo neuspešno error -EIO se smešta u pomoćnu promenljivu rc, a zatim

se skače na labelu *error2* koja prvo oslobađa prethodno zauzeti memorijski prostor a zatim vraća broj errora korisniku. Ukoliko je poziv ove funkcije prošao bez problema, sada drajver poseduje u strukturi *lp* virtuelnu adresu uređaja, preko koje može da pomoću funkcija **iowrite32** i **ioread32** upisuje u proizvoljne registre uređaja, a samim tim da obezbedi željenu funkcionalnost drajvera.

```
...
if (!request_mem_region (lp->mem_start,
      lp->mem_end - lp->mem_start+1, DRIVER_NAME))
{
      printk(KERN_ALERT "Couldn't lock memory region");
      rc = -EBUSY;
      goto error1;
}
lp->base_addr = ioremap(lp->mem_start,
                        lp->mem_end-lp->mem_start+1);
if (!lp->base_addr)
{
      printk(KERN_ALERT "led: Could not allocate iomem\n");
      rc = -EIO;
      goto error2;
}
return 0;
error2:
      release_mem_region(lp->mem_start, (lp->mem_end -
                                         lp->mem_start + 1));
error1:
      return rc;
```

U remove funkciji LED drajvera se jednostavno oslobađaju svi resursi koji su bili zauzeti u probe funkciji. *Remove* funkciju poziva kernel nakon što se u ***module_exit*** funckciji pozove funkcija ***platform_driver_unregister()***.

```
static int led_remove(struct platform_device *pdev)
{
 printk(KERN_ALERT "led platform driver removed\n");
 iowrite32(0, lp->base_addr);
 iounmap(lp->base_addr);
 release_mem_region(lp->mem_start,(lp->mem_end-lp->mem_start+1));
 kfree(lp);
 return 0;
}
```

### *3.2. Read* i *write* funkcije LED drajvera

Pri pozivu insmod komande će biti pozvana *module_init* funkcija, na čijem se kraju registruje drajver pozivom *platform_driver_register(&led_driver)*. Ukoliko u stablu uređaja postoji uređaj za koji je naš drajver namenjen (poklapaju se compatible polja), kernel će pozvati *probe* funkciju. Kao što je opisano u prethodnom poglavlju, *probe* funkcija će rezervisati virtuelni memorijski opseg pomoću kojega možemo pristupiti registrima našeg uređaja. Ukoliko se *probe* funkcija uspešno izvršila, početna virtuelna adresa se nalazi u polju *base_adress* pomoćne strukture *lp* (*lp -> base_adress*).

Sa registrima uređaja se komunicira pomoću funkcija ***iowrite32*** i ***ioread32*** koje su deklarisane u <linux/io.h>:

- void **iowrite32**(u32 val, void *__iomem* *addr);
- unsigned int **ioread32**(void *__iomem* *addr);

Upis u drajver (*led_write* funkcija) podržava nekoliko različitih fomata:

- heksadecmalni: 0xa ili 0Xa
- binarni: 0b1010 ili 0B1010
- dekadni: 10

Stoga se prvo vrši provera formata u kojem je poslat podatak, a zatim se poziva funkcija *kstrtol* koja konvertuje string u *long int*. Funkcija *kstrol* prima string koji je potrebno konvertovati, brojevni sistem i promenljivu u koju će biti smešten rezultat. Nakon što je string parsiran, poziva se *iowrite32* funkcija koja upisuje dobijenu vrednost u registar.

```
ssize_t led_write(struct file *pfile, const char __user *buffer,
size_t length, loff_t *offset)
{
	char buff[BUFF_SIZE];
	int ret = 0;
	long int led_val=0;

	ret = copy_from_user(buff, buffer, length);
	if(ret)
		return -EFAULT;
	buff[length] = '\0';

	// HEX  INPUT
	if(buff[0]=='0' && (buff[1]=='x' || buff[1]=='X'))
	{
		ret = kstrtol(buff+2,16,&led_val);
```

```
	}
	// BINARY INPUT
	else if(buff[0]=='0'  && (buff[1]=='b' || buff[1]=='B'))
	{
		ret = kstrtol(buff+2,2,&led_val);
	}
	// DECIMAL INPUT
	else
	{
		ret = kstrtol(buff,10,&led_val);
	}


	if (!ret)
		iowrite32((u32)led_val, lp->base_addr);
	else
		printk(KERN_INFO "Wrong command format\n");

	return length;
}
```

Čitanje iz drajvera (*led_read* funkcija) se uvek vrši u binarnom formatu. Budući da ne postoji funkcija koja prebacuje *integer* u string u binarnom brojevnom sistemu, ova konverzija se vrši ručno. Od interesa su samo najniža četiri bita registra, tako da možemo u *for* petlji ispitati te bite koristeći pomeranje u desno (shift) i masku (0x01).

```
ssize_t led_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
	int ret;
	int len = 0;
	u32 led_val = 0;
	int i = 0;
	char buff[BUFF_SIZE];
	if (endRead){
		endRead = 0;
		return 0;
	}

	led_val = ioread32(lp->base_addr);
```

```
	buff[0]= '0';
	buff[1]= 'b';
	for(i=0;i<4;i++)
	{
		if((led_val >> i) & 0x01)
			buff[5-i] = '1';
		else
			buff[5-i] = '0';
	}
	buff[6]= '\n';
	len=7;
	ret = copy_to_user(buffer, buff, len);
	if(ret)
		return -EFAULT;
	//printk(KERN_INFO "Succesfully read\n");
	endRead = 1;

	return len;
}
```