Stablo uređaja (*device tree*) i drajveri za platformske uređaje (*platform device drivers*)

Od početka razvoja drajvera za linux uređaje, često se dešavalo da su korisnici morali reći kernelu kojim sve uređajima on ima pristup kako bi sistem radio na korektan način. U nedostatku ovih informacija, kernel ne bi znao koje I/O portove i prekidne linije uređaj koristi, a samim tim ne bi znao kako da rukuje uređajem. Danas živimo u dobu magistrala kao što su PCIe (*Peripheral Component Interconnect Express*) koje imaju ugrađenu mogućnost automatskog detektovanja uređaja. Ovaj tip magistrale se danas koristi na personalnim računarima za povezivanje perifernih uređaja koji zahtevaju visoke performance (grafičke karte, mrežne karte, akceleratori pristupa memoriji, ...itd). To znači da će bilo koji uređaj koji je povezan preko ovakve magistrale moći da kaže ostatku sistemu koji je to tip uređaja i koje resurse zahteva. Sada krenel može da pri podizanju sistema enumeriše sve uređaje na magistrali i preuzme sve potrebne informacije od njih. Ovakve magistrale se često zovu *hot-pluggable* zato što se periferije mogu priključiti u toku rada sistema bez ikakvih poteškoća.

Ipak, danas postoji mnoštvo uređaja koji nisu povezani preko ovog tipa magistrale, pa ih procesor ne može sam otkriti. Oni se zovu *platformski uređaji* i najviše su zastupljeni u embeded 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 se korisnik sam detaljno opiše uređaj. 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 naznačeni u nizu struktura tipa *struct resource*, na koji bi pokazivao pokazivač *resource* unutar strukture *platform_device* (pogledati sledeći kodni segment). Kada bi se napravila struktura *platform_device,* ona bi se prosledila kernelu pomoću poziva funkcije *int platform_device_register(struct platform_device *pdev)*.

Listing 1. Prototip strukture za opis platformskog uređaja

```
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 nisu prikazana*/
};
```

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.

Listing 2. Primer upotrebe struktura za definisanje uređaja

```
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. Ovo je funkcionisalo, ali samo ukoliko nije bilo čestih promena u hardveru kao što su npr. registri 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, svaka ploča na kojoj je bio ARM procesor je imala svoj niz perifernih uređaja što je dovelo do nekontrolisanog povećanja board fajlova koji su postali deo izvornog koda linux kernela. Nastao je pregršt fajlova koji 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, osnivač linuxa, preuzeo inicijaivu da se pređe na sistem koji će omogućiti kompajliranje kernela za sve ARM procesore.

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 i 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 IP-ja koji kontroliše ugrađene LED na Zybo razvojnoj ploči. U kodnom listingu br.3 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.

Listing 3. Deo izvornog koda stabla uređaja

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

	myLed@43c00000
	{
		compatible = "dglnt, myled-1.00.a";
		reg = <0x43c00000 0x10000>;
	}
}
...
```

Takođe se može primetiti da uređaj po imenu *myLed* u svom opisu ima samo dva polja. Polje *reg* govori da je adresni prostor širine *0x10000* i da počinje na adresi *0x43c00000*. Budući da su registri 32-bitni, oni se adresiraju sa ofsetom 4 u odnosu na baznu adresu. Dakle, prvi bi bio na adresi 0x43c00000, drugi na 0x43c00004, treci na 0x43c00008 ... itd. Polje *compatible* je string specifičan za uređaj, drugim rečima - ime uređaja. Ovo polje koristi kernel kako bi povezao uređaj sa kompatibilnim drajverom, onim koji ima identičan string u svom *compatible* polju.

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. U ovom dokumentu se neće zalaziti u opisivanje struktura i funkcije koje su tipične za drajvere karakter uređaja (struktura file_operations, module_init i module_exit funkcije ... itd).

Najvažnija struktura koja opisuje drajver za platformski uređaj je *platform_driver*. Njen prototip se može videti u listingu br. 4.

Listing 4. Prototip strukture *platform_driver*

```
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 *myLed* u stablu uređaja (listing br.3), možemo videti da u polju *compatible* stoji string *"dglnt, myled-1.00.a"*. Dakle, u drajveru za ovaj uređaj, u jednom od polja *compatible* mora da stoji isti taj string, kako bi kernel mogao da poveže uređaj *myLed* sa ovim drajverom. U sledećem delu koda se može videti kako izgleda definicija strukture *platform_driver* u drajveru za uređaj *myLed*. U kodu se da primetiti da struktura *device_driver* nije posebno instancirana, već su njena polja direktno dodeljena strukturi *platform_driver*.

Listing 5. Definisanje strukture *platform_driver*

```
static struct of_device_id led_of_match[] =
{
	{ .compatible = "dglnt,myled-1.00.a", },
	{ /* 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 inicijalizaciju (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 prikazan u listingu br.1. Kada kernel pozove neku od ovih funkcija on kao parametar prosledi *platform_device* strukturu uređaja koji će biti kontrolisan tim drajverom (npr. *myLed*). 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
  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*.

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

Probe funkcija myLed drajvera

Kako bi se demonstrirala upotreba pomenutih funkcija, u nastavku će biti opisana probe funkcija drajvera za *myLed* uređaj. Ukoliko je u nekom od četiri registra upisana logička jedinica, LED koja odgovara tom registru će biti upaljena. Ukoliko je vrednost registra jednaka nuli, LED će biti ugašena. Kako bi pristupili registrima koji kontrolišu diode, moramo za početak da izdvojimo njihove fizičke adrese 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. Adrese ostalih registara se dobijaju sa inkrementom od 4 u odnosu na početnu adresu.

```
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. 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 "invalid address\n");
		return -ENODEV;
	}
	lp = (struct led_info *)
		kmalloc(sizeof(struct led_info), GFP_KERNEL);
	if (!lp) {
		printk(KERN_ALERT "Cound not allocate led device\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 eror -EBUSY , koji simbolizira da je memorijski opseg već zauzet. U tom slučaju se skače na labelu *error1* koja pomoću *return* funkcije kornisniku vraća broj erora 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 eror -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 erora 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;
}
```