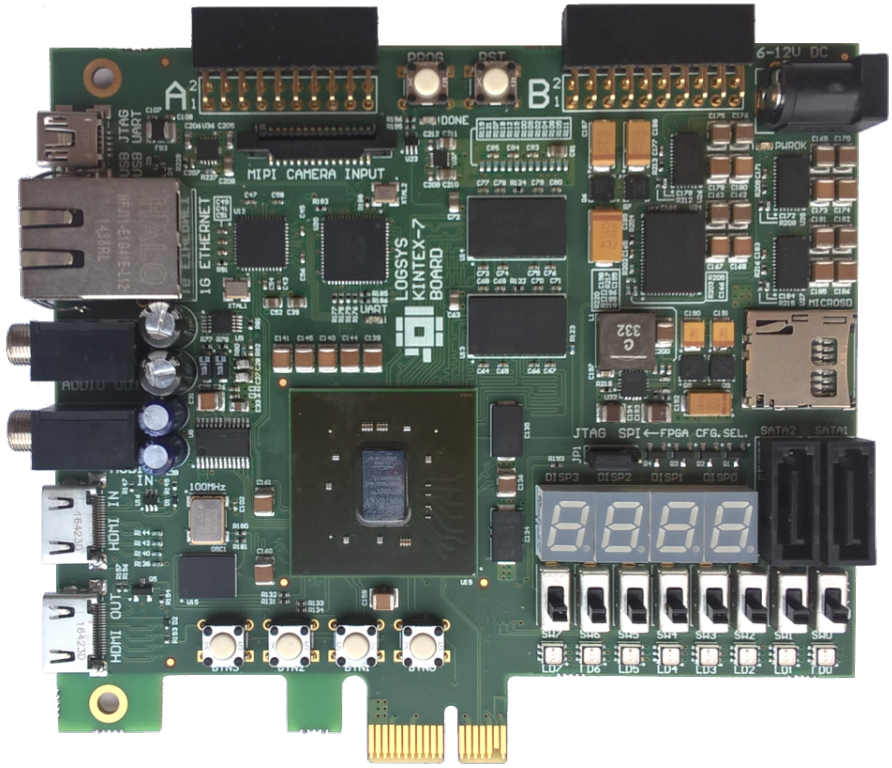
LOGSYS

LOGSYS KINTEX-7 FPGA KÁRTYA

ALKALMAZÁSI ÚTMUTATÓ PCI EXPRESS ALAPÚ RENDSZER ÉPÍTÉSÉHEZ



2017. június 19.

Verzió 1.0

http://logsys.mit.bme.hu

Tartalomjegyzék

1 Bevezetés

A LOGSYS Kintex-7 FPGA kártya az egyszerűbb (kapcsolók, gombok, LED-ek) perifériák mellett nagysebességű számítógépes összeköttetést is biztosít PCI Express interfészen keresztül.

Ez a dokumentum egy egyszerű PCI Express mintarendszer elkészítéséhez nyújt segítséget, valamint a rendszerhez való Linux driver fejlesztését mutatja be.

2 A hardver felépítése

2.1 A hardver működése

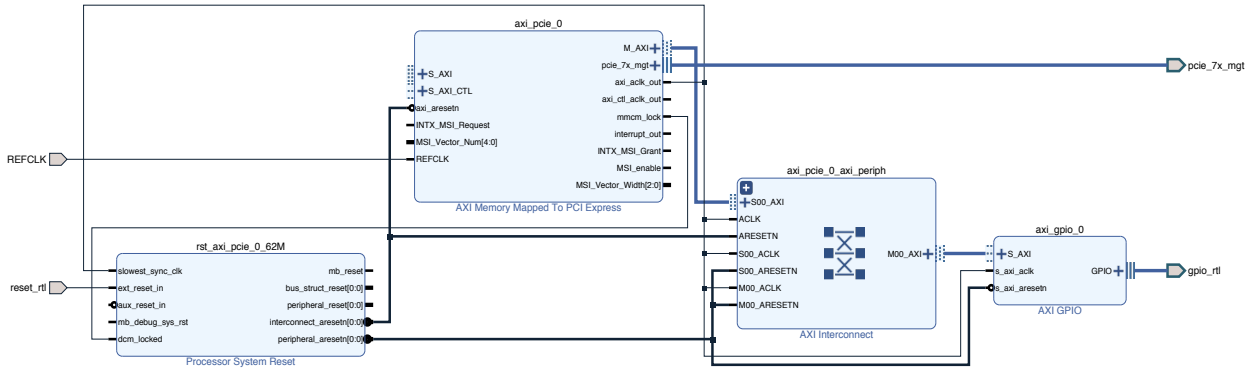
A hardver a LOGSYS Kintex-7 FPGA kártyán található LED-eket képzi le a PCI Express címtartományába, így azok a számítógépről vezérelhetőek lesznek.

2.2 A szükséges szoftverek

A rendszer létrehozásához a dokumentum írásakor legfrissebb Xilinx Vivado 2017.1-et ajánljuk. Újabb verziókban adódhatnak eltérések.

2.3 A hardver blokkvázlata

A hardver felépítése a következő:



A rendszer az egyszerű bővíthetőség érdekében PCI Express – AXI híd interfészt használ, így a Xilinx által biztosított AXI IP core-ok használhatóak.
Az elhelyezett „AXI Memory Mapped To PCI Express” IP core a megszokott AXI buszrendszer alapú fejlesztést teszi lehetővé, az órajelet a külső „REFCLK” jelből állítva elő. A reset jelet generáló core számára leglassabb órajelnek az AXI busz órajelét szükséges megadni, „dcm_locked” jelnek (mely azt jelzi, hogy a kívánt frekvencia előállt) az mmcm_lock jelet szükséges megadni [1].
A „REFCLK” jel a kártya 100MHz-es órajele, a „reset_rtl” a kártya reset jele, a „gpio_rtl” az RGB LED-ekhez tartozó 24 láb, a „pcie_7x_mgt” pedig a PCI Express kommunikációhoz tartozó két differenciális érpár (RX és TX).

2.4 A PCI Express-AXI híd interfész konfigurációja [1]

Az „AXI Memory Mapped To PCI Express” IP core alapbeállításai megfelelőek, a „PCIE:ID” beállításoknál változtathatóak az eszköz azonosítói, melyeket a driverfejlesztésnél az eszköz

felismeréséhez használunk. A fülön beállíthatóak még az eszköz osztályazonosítói, mely alapján funkcionalitásbeli besorolást adhatunk a hardvernek.

3 A szoftver felépítése

3.1 Előkészületek

A driverfejlesztéshez Debian GNU/Linux alapú operációs rendszer ajánlott (Debian, Ubuntu, Linux Mint). Ezen rendszereken szükséges a `build-essential` és a `linux-headers` csomagok feltelepítése. A PCI Express buszon található eszközök listázásához a `pciutils` csomag telepítése is ajánlott.
A PCI Express slotba helyezett Logsys Kintex-7 kártya minden felprogramozása után – a helyes detektálás és hardverinicializálás érdekében – az operációs rendszer újraindítása szükséges.
A helyes hardverműködésről az `lspci` parancs kiadásával győződhetünk meg, ez a számítógéphez kapcsolt PCI eszközökről ad információt.
Egy lehetséges kimenet részlete:

```
04:00.0 Communication controller: Xilinx Corporation Device 7011
```

Ebben az esetben az „AXI Memory Mapped To PCI Express” IP core beállításainál vendor azonosítónak a Xilinx-et adtuk meg (0x10EE), device azonosítónak pedig 0x7011-et. Az osztályazonosító 0x7, így kommunikációs eszközként jelenik meg.

3.2 A Linux driver felépítése

A Linux driver két részből áll: a C forrásfileból, mely a működést írja le, illetve a driver fordítását leíró Makefile-ból.

3.3 A driver fordításához szükséges Makefile[2]

A Makefile felel azért, hogy a forrásunk a megfelelően forduljon:

1. Adott verziójú kernelhez,
2. Adott architektúrára,
3. Adott forrásfileokból,
4. És „szokásos” futtatható helyett Linux kernel modul formátumúvá.

A Makefile tartalma a következő:

```
obj-m += logsysled.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Az első sor alapján a driverünk egy fileból fog állni, mely a `logsysled.o` tárgykódra fordul: a forrásfile neve `logsysled.c` lesz.
Az *all* és *clean* receptek pedig a jelenleg futó kernel fordítási keretrendszerét hívják meg, hogy a jelenleg futó kernellel kompatibilis modult állítson elő.

3.4 A driver felépítése

Ellentétben egy szokásos C programmal, melynek egy belépési pontja van, a *main()*, egy Linux driver felépítése eltér, ugyanis eseményvezérelt.

3.4.1 Belépési és kilépési pontok

Egy Linux drivernek két eseménykezelőt kell biztosítania a Linux kernel felé: inicializálás és kilépés. Az inicializálás a driver betöltésekor, a kilépés pedig a driver leállításakor hívódik meg. A hardverhez tartozó *logsysled* driverben ez a következőképpen néz ki:

```
int logsysled_init(void){
  return pci_register_driver(&logsysled_driver);
}

void logsysled_exit(void){
  pci_unregister_driver(&logsysled_driver);
}

module_init(logsysled_init);
module_exit(logsysled_exit);
MODULE_LICENSE("GPL");
```

A driver belépési pontja a *logsysled_init* függvény lesz, ezt jelzi a *module_init(logsysled_init)* makróhívás. Kilépési pont a *logsysled_exit*.

A driver belépéskor egy PCI eszközkezelőt regisztrál a Linux kernel számára, mely a *logsysled* hardvert fogja kezelni. Erről a 3.4.2 alfejezet ír bővebben.

3.4.2 A PCI driver leírása

A PCI és PCI Express hardvereszközök azonosítása vendor és device ID alapján történik. Linux PCI Express driver fejlesztése során regisztrálnunk kell a kernel számára, hogy a driver milyen azonosítójú eszközöket kezel, illetve mit tesz abban az esetben, ha egy ilyen eszközt hozzáadnak a rendszerhez (*probe*), illetve ha eltávolítanak (*remove*)

A 3.4.1 alfejezetben található *logsysled_driver* ezek alapján a következőképpen néz ki:

```
static const struct pci_device_id logsysled_id_table[] = {
  { PCI_DEVICE(0x10ee, 0x7021), },
  { PCI_DEVICE(0x10ee, 0x7011), },
  {}
};

static struct pci_driver logsysled_driver = {
  .name = "logsysled",
  .id_table = logsysled_id_table,
  .probe = logsysled_probe,
  .remove = logsysled_remove
};
```

A driver a *logsysled_id_table* tömbben leírt PCI eszközöket fogja kezelni. A vendor azonosítója 0x10ee, tehát a 2.4 fejezetben megadott Xilinx, az eszközazonosító pedig többféle is lehet, esetünkben 0x7011 és 0x7021, ha ez a driver kétféle PCI eszközhöz is nyújt támigatást. A driver neve „logsysled”, a kernel ezen a néven fogja azonosítani a felhasználó felé.

A driverrel kompatibilis hardvereszköz észlelésekor mintegy konstruktorként a *logsysled_probe*, eltávolításakor a destruktorhoz hasonló *logsysled_remove* függvény hívódik meg.

3.4.3 A *probe()* és *remove()* függvények

Eszköz észlelésekor a regisztrált *probe()* függvény hívódik meg. A *probe()* függvény feladata az eszközhöz tartozó erőforrások lefoglalása, melyek a következők lehetnek:

1. IO erőforrások: az eszköz memóriaterületében megtalálható regiszterek
2. Megszakítások: az eszköz által generált megszakításokhoz tartozó vonalak
3. Az eszköz adminisztrációjához szükséges driverspecifikus struktúrák
4. Az eszköz és a felhasználó közötti kommunikációhoz szükséges driverspecifikus interfész

Ezen erőforrások lefoglalása azért szükséges, mert a Linux kernel több drivert is tartalmazhat, így lefoglalás nélkül esetleg egy hardveregységhez párhuzamosan több driver is hozzá tudna férni.

A *logsysdev_probe* függvény hibakezelés nélküli verziója a következő:

```
static int logsysled_probe(struct pci_dev * pdev, const struct
pci_device_id * ent){
  unsigned int * mem;
  pci_enable_device(pdev);
  pci_request_region(pdev,0,"logsysled");
  mem=pci_iomap(pdev, 0, 0);
  pci_set_drvdata(pdev,mem);
  sysfs_create_group(&pdev->dev.kobj, &logsysled_attribute_group);
  return 0;
}
```

A PCI eszközt azonosító és leíró struktúra a *pdev* argumentumban található, a hozzá tartozó azonosítóra – melyre az eszköz illeszkedett az *ent* mutat. Jelen esetben a támogatott eszközök működése ugyanolyan, így nem szükséges az azonosító vizsgálata.

A *pci_enable_device()* a hardver alacsonyszintű inicializálását végzi el, ennek meghívása PCI driverek esetén szükséges. Ezek után a 0. PCI erőforrást (BAR0) kérjük le a *pci_request_region()* függvénnyel.

Virtuális címzéssel rendelkező platform esetén – ilyen az x86 architektúra is – szükséges az adott fizikai cím virtuális címtartományba képzése, ezt a *pci_iomap()* függvény végzi. A *pdev* által azonosított eszköz 0. erőforrását (BAR0) képezzük le a virtuális címtartományba, a későbbiekben a visszatérési értékben kapott mutatót kezelhetjük mintegy báziscímként. A *pdev* által mutatott struktúra – mely az eszközhöz tartozik – tartalmazhat driverspecifikus adatot, melybe a driver saját belső adminisztrációjához szükséges értékeket menthetünk. Mivel az eszköz címét mind a felszabadításkor, mind a felhasználói térből való hozzáférésnél használni fogjuk, érdemes elmenteni a *pci_set_drvdata()* függvénnyel.

Legvégül a *probe()* függvény létrehoz a felhasználói térrel való kommunikációra a */sys* filerendszerben megtalálható fileokat, ezekről a 3.4.4 alfejezet szól.

A *remove()* függvény hasonló a *probe()*-hoz, a felhasznált erőforrásokat fordított sorrendben felszabadítja:

```
static void logsysled_remove(struct pci_dev * pdev){
  unsigned int * mem=pci_get_drvdata(pdev);
  sysfs_remove_group(&pdev->dev.kobj, &logsysled_attribute_group);
  pci_iounmap(pdev, mem);
  pci_release_region(pdev,0);
  pci_disable_device(pdev);
  return;
}
```

Érdemes észrevenni, hogy a *probe()* függvényben megkapott eszközcímet nem globális változóként, hanem az eszközhöz csatolt driver adatként tároltuk, így a driverünk több példányt is támogat ugyanabból az eszközből, lebontáskor mindig csak az adott hardvereszközhöz tartozó erőforrás szabadul fel.

3.4.4 Sysfs attribútumok

A 2. fejezetben ismertetett hardverplatform tartalmaz egy GPIO perifériát, mely a Logsys Kintex-7 kártyán található LED-ekre van kikötve. Egy drivernél általában biztosítani kell a felhasználóval – a felhasználói térben futó alkalmazásokkal – való kommunikációt, ennek demonstrálására a LED perifériát alkalmazzuk. A felhasználó képes lesz saját maga állítani a LED-ek állapotát anélkül, hogy a driver kódjához hozzá kelljen írnia.

A Linux, és más Unix és Unix-szerű operációs rendszer filozófiája szerint a kernel és a felhasználói programok közötti kommunikáció fileokon keresztül zajlik, ahol filenak nevezünk minden olyan objektumot, melyen a megnyitás, lezárás, írás, olvasás műveletek értelmezhetőek. Jelen esetben a driver egy filet fog létrehozni a filerendszerben, melybe íráskor a fileba írt szám közvetlenül a LED perifériára lesz kiírva.

A Linux driverek által használt általános megvalósítások a következők lehetnek:

1. Karakteres eszköz. A karakteres eszköz lényegében egy olyan file, mely megnyitás, bezárás, írás és olvasás esetén a kernelbe regisztrált függvényeinket hívja meg, ennek kezelése a driverfejlesztő feladata. Tipikus felhasználása nagyobb méretű adatok átvitele.
2. Procfs bejegyzés. A procfs tipikusan processzekről való információk megjelenítésére szolgál, általában a */proc/* könyvtár alatt található.
3. Sysfs bejegyzés. A */sys/* könyvtárban a Linux kernel által látott hardverstruktúra jelenik meg, hierarchiába rendezve. Az egyes hardvereszközökhöz – a PCI eszközökhöz is – egy-egy könyvtár tartozik a hierarchiában. A könyvtáron belül létrehozhatunk saját fileokat – attribútumokat –, melyek megváltoztatásával befolyásolhatjuk a driver működését.

A LED-ek vezérlésére legkézenfekvőbb a Sysfs attribútum, egyfelől egyszerűsége révén, másfelől a LED-ek állapota logikailag is az eszközállapothoz tartozik.

Sysfs attribútum hozzáadásakor lényegében a kernel által nyilvántartott PCI eszköz egyik tulajdonságához – esetünkben a LED-ek állapotához – írunk getter és setter függvényeket.

A getter és a setter (show és store) függvények a következőképpen néznek ki:

```
ssize_t show_led(struct device *dev, struct device_attribute * attr,
           char *buf) {
  unsigned int * mem;
  unsigned int led;
  mem = dev_get_drvdata(dev);
  led=ioread32(mem);
  return scnprintf(buf, PAGE_SIZE, "%u\n", led);
}
```

A *show_led()* függvény a paraméterül megkapott eszközből, *dev*-ből (mely lényegében azonos a *probe()* függvényben megkapott *pdev* paraméterrel) kinyeri az eltárolt báziscímet, majd az *ioread32* függvénnyel kiolvassa az azon a címen található értéket – esetünkben a PCI-AXI hídon keresztül a GPIO periféria első regiszterét, a LED-ek állapotát. Ezt az integer értéket az *scnprintf()* függvény segítségével a *buf* által mutatott stringbe írjuk. Ennek hatására az attribútum fileból való kiolvasásakor a LED-ek állapotát kapjuk vissza olvasható formában.
A setter függvény, a *store_led()* ehhez hasonló, csak fordítottan:

```
ssize_t store_led(struct device *dev, struct device_attribute *attr,
           const char *buf, size_t count) {
  unsigned int * mem;
  unsigned int led;
  dd = dev_get_drvdata(dev);
  if(sscanf(buf,"%d",&led)==1){
    iowrite32(led, mem);
    return count;
  }
  else
    return -EINVAL;
}
```

A különbség, hogy ebben az esetben *buf* bejövő érték, egy string, melyet az *sscanf()* függvénnyel értelmezünk integer típusúvá. Amennyiben ez sikerült, *count* értékkel való visszatéréssel jelezzük, hogy a pufferben kapott teljes stringet feldolgoztuk, hiba esetén viszont *-EINVAL*, azaz „Invalid argument” hibával térünk vissza.
Az *iowrite32()* makró felel a *led* változóba eltárolt érték kiírására a GPIO perifériába.
A Sysfs attribútumot még regisztrálnunk kell a kernel számára:

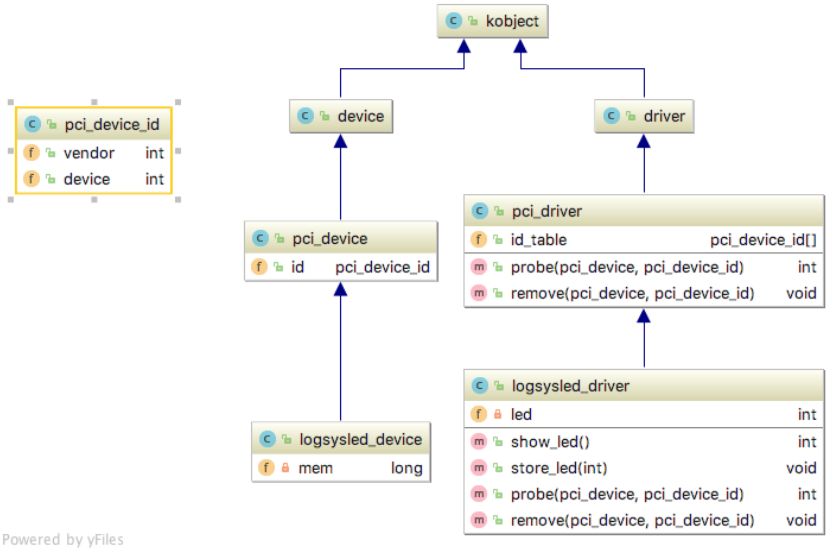
```
DEVICE_ATTR(led, 0644, show_led, store_led);

static struct attribute * logsysled_attributes[] = {
  &dev_attr_led.attr,
  NULL
};

static struct attribute_group logsysled_attribute_group = {
  .name = NULL,
  .attrs = logsysled_attributes
};
```

A *DEVICE_ATTR()* makró definiálja a *led* nevű attribútumot, melynek filerendszerbeli jogosultsága 0644 lesz: tulajdonos írhatja és olvashatja, csoportba tartozó felhasználók és mindenki más csak olvashatja. Ugyanez a makró rendeli hozzá az attribútumhoz a getter és setter függvényeket.
Ha több attribútumunk lenne, a *logsysled_attributes* tömbbe a *dev_attr_led.attr*-hoz hasonló módon hozzáadhatjuk, a tömb utolsó eleme a lezáró NULL.

A definiált attribútumokat a 3.4.3 alfejezetben megvalósított *probe()* adja hozzá a kernelhez a függvényben található
`sysfs_create_group(&pdev->dev.kobj, &logsysled_attribute_group);`
sorban. Ez az eszközünkhöz tartozó sysfs könyvtár alatt létrehozza az összes – jelen esetben egy – definiált attribútumhoz tartozó filet, ebben az esetben a *led* nevűt.

3.5 A driver felépítésének szemléltetése

A Linux kernelben megvalósított driverek erősen objektum orientáltak. A driver felépítése és megvalósítása szemléletesebb lehet, ha az osztályok – eszközök és driverek – közötti kapcsolatot látjuk. A PCI driver struktúrája OOP szemlélettel a következő osztálydiagramot eredményezi:



Minden Linux kernelben létező objektum a *kobject* ősből származik. Az eszközöket és drivereket az absztrakt *device* és *driver* képviselik. Amikor a PCI drivert hozunk létre, a *pci_driver* osztályt (melyet C *struct* alapon valósítanak meg) bővítjük ki saját attribútumokkal (esetünkben a *led*-del), és definiáljuk felül a *probe* és *remove* metódusokkal.
Az eszközhöz tartozó – ám a driver által használt – adattaggal a *pci_device* eszközt egészítjük ki (ebben az esetben az eszközhöz tartozó báziscímmel).

3.6 A driver tesztelése

3.6.1 Fordítás és betöltés

A driver fordítását a könyvtárában kiadott *make* paranccsal tehetjük meg. Hibátlan lefutás esetén ez a *logsysled.ko* (*.ko*, mint Kernel Object) filet állítja elő, ez tartalmazza a lefordított drivert, melyet a kernelhez adhatunk.
A kész modult az *insmod logsysled.ko* paranccsal tölthetjük be, mint minden, a rendszer biztonságát érintő, műveletet, ezt is csak rendszergazdaként futtathatjuk. Ilyenkor meghívódik a driver általunk regisztrált *init()* függvénye, illetve minden, a driverrel kompatibilis PCI eszközre a beregisztrált *probe()*.
A modult a rendszergazda jogosultsággal futtatott *rmmod logsysled* paranccsal távolíthatjuk el, ekkor hívódik meg az összes kezelt eszközre a felszabadítást végző *remove()*, majd a driver *exit()* függvénye.

3.6.2 A LED-ek vezérlése

A beregisztrált sysfs attribútumok az eszközhöz tartozó */sys* alatti könyvtárban találhatóak. PCI eszközök esetén ezeket a */sys/bus/pci/devices* könyvtárban kell keresni.
Amennyiben az *lspci* parancs kimenete a következő:

```
04:00.0 Communication controller: Xilinx Corporation Device 7011
```

Akkor az eszközünk könyvtára a */sys/bus/pci/devices/**04:00.00*** lesz, ezen belül találjuk meg a *led* filet.
Ebbe a fileba különböző értékeket írva az *echo* paranccsal (például *echo 9999 > led*) meghívódik az attribútumhoz beregisztrált *store()* függvény, mely az *iowrite32()* függvénnyel a perifériára juttatja az értéket.
Ehhez hasonlóan kiolvasni a *cat led* paranccsal lehet.

4 Tipikus felhasználás és továbbfejlesztés

A mintarendszer kiválóan alkalmas olyan fejlesztések kiindulási alapjaként, ahol PCI Express buszon keresztül kis adatátvitelekkel – tehát DMA igénybevétele nélkül – szeretnénk az FPGA-n található perifériát vezérelni.
Kisebb módosítással támogatja a megszakításokat is, mind *legacy* mind MSI módon.
Egyszerű periféria esetén, ahol a nagy teljesítmény és a sebesség/késleltetés nem számít, lehetséges az eszköz elérése kernel driver fejlesztése nélkül is [3].

5 Forráskódok

5.1 A Makefile

```
obj-m += logsysled.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

5.2 A hibakezeléssel is rendelkező logsysled.c

```
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/io.h>

struct logsysled_data {
  unsigned int * mem;
  /* ide opcionálisan egyéb adatokat is tehetünk */
};

ssize_t show_led(struct device *dev, struct device_attribute * attr,
           char *buf) {
  struct logsysled_data * dd;
  unsigned int led;
  dd = dev_get_drvdata(dev);
  led=ioread32(&dd->mem[0]);
  return scnprintf(buf, PAGE_SIZE, "%u\n", led);
}

ssize_t store_led(struct device *dev, struct device_attribute *attr,
           const char *buf, size_t count) {
  struct logsysled_data * dd;
  unsigned int led;
  dd = dev_get_drvdata(dev);
  if(sscanf(buf,"%d",&led)==1){
    iowrite32(led,&dd->mem[0]);
    return count;
  }
  else
    return -EINVAL;
}

DEVICE_ATTR(led, 0644, show_led, store_led);

static struct attribute * logsysled_attributes[] = {
  &dev_attr_led.attr,
  NULL
```

```
};

static struct attribute_group logsysled_attribute_group = {
  .name = NULL,
  .attrs = logsysled_attributes
};


static int logsysled_probe(struct pci_dev * pdev, const struct
pci_device_id * ent){
  int err;
  unsigned int * mem;
  struct logsysled_data * priv;
  priv=kzalloc(sizeof(struct logsysled_data),GFP_KERNEL);
  if(!priv){
    err= -ENOMEM;
    goto err_alloc;
  }
  if((err=pci_enable_device(pdev))){
    goto err_enable;
  }
  if((err=pci_request_region(pdev,0,"logsysled"))){
    goto err_request;
  }
  mem=pci_iomap(pdev, 0, 0);
  if(!mem){
    err=-ENOMEM;
    goto err_map;
  }
  priv->mem=mem;
  pci_set_drvdata(pdev,priv);
  if ((err=sysfs_create_group(&pdev->dev.kobj,
logsysled_attribute_group))) {
    goto err_sysfs;
  }
  goto err_success;
/* visszagörgetés hiba esetén */
 err_sysfs:
  pci_iounmap(pdev, mem);
 err_map:
  pci_release_region(pdev,0);
 err_request:
  pci_disable_device(pdev);
 err_enable:
  kfree(priv);
 err_alloc:
 err_success:
  return err;
}
```

```
static void logsysled_remove(struct pci_dev * pdev){
  struct logsysled_data * priv=pci_get_drvdata(pdev);
  void * mem=priv->mem;
  sysfs_remove_group(&pdev->dev.kobj, &logsysled_attribute_group);
  pci_iounmap(pdev, mem);
  pci_release_region(pdev,0);
  pci_disable_device(pdev);
  kfree(priv);
  return;
}

static const struct pci_device_id logsysled_id_table[] = {
  { PCI_DEVICE(0x10ee, 0x7021), },
  { PCI_DEVICE(0x10ee, 0x7011), },
  {}
};

static struct pci_driver logsysled_driver = {
  .name = "logsysled",
  .id_table = logsysled_id_table,
  .probe = logsysled_probe,
  .remove = logsysled_remove
};

int logsysled_init(void){
  return pci_register_driver(&logsysled_driver);
}

void logsysled_exit(void){
  pci_unregister_driver(&logsysled_driver);
}

module_init(logsysled_init);
module_exit(logsysled_exit);
MODULE_LICENSE("GPL");
```

6 Irodalomjegyzék

[1] Xilinx, „AXI Memory Mapped to PCI Express (PCIE) Gen2 v2.8 LogiCORE IP Product Guide,” 4 Április 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_8/pg055-axi-bridge-pcie.pdf. [Hozzáférés dátuma: 19 Június 2017].

[2] J. Corbet, A. Rubini és G. Kroah-Hartman, Linux Device Drivers, 3rd Edition szerk., O'Really Media, 2005.

[3] F. Bill, „Simple program to read & write to a pci device from userspace,” 25 július 2016. [Online]. Available: https://github.com/billfarrow/pcimem. [Hozzáférés dátuma: 19 június 2017].

7 Változások a dokumentumban

Dátum	Verzió	Megjegyzés
2017. június 19.	1.0	Az első kiadás.