Hogyan írjunk linuxos USB-eszközvezérlőt?

Greg USB-vezérlővázlatát osztja meg velünk és azt is megmutatja, miképpen formálhatjuk azt a saját igényeinknek megfelelően.

A Linux USB-alrendszer a 2.2.7 rendszermagban megjelent mindössze két különböző eszköz (az egér és a billentyűzet) támogatásából nőtte ki magát. Mára a 2.4-es rendszermag több mint húsz különféle eszköztípust támogat. A Linux jelenleg csaknem minden USB-osztályú eszközt támogat (például a billentyűzetet, az egeret, a modemet, a nyomtatót és a hangszórót), illetve folyamatosan növekszik a gyártófüggő eszközök (mint az USB – soros átalakítók, digitális kamerák, etherneteszközök és MP3-lejátszók) támogatottsága. A jelenleg támogatott különféle USB-eszközök teljes listáját a ***Kapcsolódó címek*** között találhatjuk meg.

A fennmaradó USB-s eszközök, amelyekhez nincs Linux-támogatás, csaknem mind gyártótól függő eszközök. Minden gyártó maga határozhatja meg saját eszköze protokollját, amelyen keresztül az eszközzel kapcsolatot lehet tartani – emiatt többnyire saját vezérlőt kell írnunk. Néhány gyártó USB-protokollját nyílttá tette és segíti a Linux-vezérlők készítését, mások nem teszik közzé, ezért a fejlesztők kénytelenek visszafejteni azt. Ez pedig komoly jogi kérdéseket vonhat maga után, de szerencsére egyre több gyártó látja be, hogy a Nyílt Forrás Közössége is értékes célréteg számára.

Mivel minden eltérő protokollhoz új vezérlőt kell készíteni, egy általános USB-vázat hoztam létre, amelynek alapjául a ***pci-skeleton.c*** fájl szolgált. Ez az a fájl a rendszermag-forrásfában, amelyen igen sok PCI hálózati kártyavezérlő alapul. Egy USB-váz található még a ***drivers/usb/usb-skeleton.c*** helyen a rendszermag-forrásfában. Ebben a cikkben végigsétálunk a vázvezérlő alapjainak lépésein, elmagyarázom az egyes részeket, illetve elmondom, mit kell tennünk, hogy egy adott eszközhöz igazítsuk.

Ha linuxos USB-vezérlőt akarunk írni, elsőként nem árt, ha megismerkedünk az USB protokollal. Ezt számos más hasznos leírással együtt az USB honlapon találjuk (lásd a ***Kapcsolódó címeket***). A linuxos USB-alrendszerrel kapcsolatos kitűnő bemutatkozó írást találhatunk az „USB Working Devices List” helyen. Ez bemutatja, miképpen épül fel az USB-alrendszer, és az olvasót megismerteti az USB ***urbs***-ok fogalmával, amelyek létfontosságúak az USB-eszközökhöz.

Az első dolog, amit a linuxos USB-vezérlőnek meg kell tennie, hogy bejegyzi magát a Linux USB-alrendszeren. Ilyenkor néhány adatot ad át arról, hogy a vezérlő mely eszközöket támogatja, és milyen eljárásokat kell meghívnia, amikor a vezérlő által támogatott eszközt csatlakoztatnak vagy választanak le a rendszerről. Mindezen adatok az USB-alrendszerhez kerülnek, és az `usb_driver` szerkezetben tárolódnak. A vezérlőváz az `usb_driver`-t a következőképpen határozza meg:

```
static struct usb_driver skel_driver = {
      name:   "skeleton",
      probe:  skel_probe,
      disconnect:    skel_disconnect,
      fops:   &skel_fops,
      minor:  USB_SKEL_MINOR_BASE,
      id_table:      skel_table,
};
```

A `name` változó egy karaktersorozat, amely a vezérlőt azonosítja. A rendszer ezt használja a rendszernaplókba írt üzenetekhez. A `probe` és `disconnect` függvények mutatóit akkor kell meghívnunk, amikor az `id_table` változóban megadott adatoknak megfelelő eszköz megjelenik vagy eltávolítják.

A `fops` és `minor` változók elhagyhatók. A legtöbb USB-vezérlő más rendszermag-alrendszerekre is csatlakozik, mint például SCSI-, hálózati vagy TTY-alrendszer. Ezek a típusú vezérlők más rendszermag-alrendszerekben is bejegyzik magukat, és minden felhasználói beavatkozás e csatolófelületeken keresztül megy végbe. Azon vezérlők esetében, amelyekhez nem tartozik rendszermag-alrendszer, mint például az MP3-lejátszóknál vagy a lapolvasóknál, valamilyen módszert kell találnunk a felhasználóval való kapcsolattartásra. Az USB-alrendszer lehetőséget ad kisebb eszközszám (minor device number) megadására és olyan `file_operations` függvénykészlet-mutatók megadására, amelyek megengedik ezt a felhasználói beavatkozást. A vázvezérlőnek ilyen csatolófelületre van szüksége, ezért a `file_operations` függvényekre egy kisebb kezdőszámot és egy mutatót szolgáltat.

Az USB-vezérlőt ezután az `usb_register` hívással bejegyezzük, általában a vezérlő `init` függvényében, ahogyan azt az *1. lista* is mutatja.

Amikor a vezérlő törlődik a rendszerből, az USB-alrendszerből ki kell jelentenie magát. Ezt az `usb_unregister` függvénnyel tehetjük meg:

```
static void __exit usb_skel_exit(void)
{
   /* a vezØrlıt kijelentj k az
   USB-alrendszerbıl*/
   usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

Ha a linux-hotplug rendszert engedélyezni szeretnénk, hogy a megfelelő vezérlő önműködően betöltődjék, amikor az eszközt csatlakoztatják, egy `MODULE_DEVICE_TABLE` táblát kell készítenünk. A következő kód a `hotplug` parancsfájlnak azt mutatja meg, hogy ez a modul egyetlen eszközt támogat – egy adott gyártó- és termékazonosítóval:

```
/* a vezØrlıvel mßk dı eszk z k listÆja */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID,
      USB_SKEL_PRODUCT_ID) },
    { }            /* Megsz ntetı bejegyzØs */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

1. lista Az USB-vezérlő bejegyzése

```
static int __init usb_skel_init(void)
{
   int result;

   /* a vezØrlıt bejegyezz k az USB-
      alrendszerben */
   result = usb_register(&skel_driver);
   if (result < 0) {
      err("usb_register failed for the"
           ➥__FILE__
           ➥" driver. Error number %d",
           ➥result);
      return -1;
   }

   return 0;
}
module_init(usb_skel_init);
```

2. lista A skel_write függvény

```
/* mind ssze egyetlen egy urb-ot tudunk  rni */
bytes_written = (count > skel->bulk_out_size)
  ? skel->bulk_out_size : count;

/* az adatot a felhasznÆl t l mÆsoljuk az urb-ba */
copy_from_user(skel->write_urb->transfer_buffer,
  ➥buffer, bytes_written);

/* az urb beÆll tÆsa */
FILL_BULK_URB(skel->write_urb, skel->dev,
            usb_sndbulkpipe(skel->dev,
            skel->bulk_out_endpointAddr),
            skel->write_urb->transfer_buffer,
            bytes_written,
            skel_write_bulk_callback,
            skel);

/* az adatot Ætk ldj k bulk-kapun */
result = usb_submit_urb(skel->write_urb);
if (result) {
      err(__FUNCTION__ " - failed submitting write
            ➥urb, error %d", result);
}
```

Vannak más makrók is, amelyeket olyan `usb_device_id`-k leírásához használhatunk fel, amelyek egy egész USB-vezérlő-osztályt támogatnak. További adatok az ***usb.h*** fájlban találhatók. Amikor az USB-sínre olyan eszközt csatlakoztatunk, amelynek device ID-mintája megegyezik a USB core-ban bejegyzett vezérlőével, a `probe` függvény hívódik meg. Az `usb_device` szerkezet, a csatolószám és a csatolóazonosító értékként átadódik a függvénynek:

```
static void * skel_probe(struct usb_device
              *dev, unsigned int ifnum, const
              struct usb_device_id *id)
```

A vezérlőnek most ellenőriznie kell, hogy az eszköz valóban olyan, mint amilyet vár. Ha az az alaphelyzetbe állítás közben nem ilyennek bizonyul, vagy bármilyen hiba lép fel, a `probe` függvényből `NULL` értékkel tér vissza. Máskülönben a vezérlő visszatérési értéke egy olyan mutató, amely az ehhez az eszközhöz tartozó állapotot leíró belső adatszerkezetre mutat. Ez a mutató az `usb_device` szerkezetben tárolódik, és a vezérlő összes meghívása (`callback`) majd ezt az értéket adja át.
A vezérlővázban megadjuk, hogy milyen végpontok vannak adattömeg-bemenetként (bulk data input) és -kimenetként megjelölve. Készítünk egy átmeneti tárat, ami tárolja az eszközre kiküldésre szánt, illetve onnan beolvasott adatot, illetve egy USB urb-ot, ahová az adatot az eszköz alaphelyzetbe állításához írjuk. Ugyanakkor az eszközt a devfs alrendszeren bejegyezzük, hogy a devfs felhasználói elérhessék. Ez a bejegyzés valahogy így néz ki:

```
/* az eszk zh z alaphelyzetbe Æll tunk egy
   devfs csom pontot Øs bejegyezz k */
sprintf(name, "skel%d", skel->minor);

skel->devfs = devfs_register
      (usb_devfs_handle, name,
      DEVFS_FL_DEFAULT, USB_MAJOR,
      USB_SKEL_MINOR_BASE + skel->minor,
      S_IFCHR | S_IRUSR | S_IWUSR |
      S_IRGRP | S_IWGRP | S_IROTH,
      &skel_fops, NULL);
```

Ha a `devfs_register` függvény kudarcot vall, nem törődünk vele, hiszen a devfs alrendszer úgyis jelenteni fogja a felhasználónak.
Ugyanígy, amikor az eszközt eltávolítják az USB-sínről, a `disconnect` függvény hívódik meg az eszköz mutatójával. A vezérlőnek minden saját adatot, amit csak lefoglalt, ki kell ürítenie, és minden függőben lévő ***urb***-ot le kell zárnia az USB-rendszeren. A vezérlő a devfs alrendszerről a következő hívással egyúttal ki is jelenti magát:

```
/* eltÆvol tjuk a devfs csom pontot */
devfs_unregister(skel->devfs);
```

Most, amikor az eszközt már csatlakoztattuk a rendszerhez és a vezérlőt az eszközhöz rendeltük, az USB-alrendszernek átadott `file_operations` szerkezetben található bármely függvény meghívható egy felhasználói programból, amely megpróbál kapcsolatot tartani az eszközzel. Az első meghívott függvény az `open` lesz, hiszen a program az eszközt megpróbálja megnyitni ki- és bemenetre (I/O-ra). Ha a vezérlőváz `open` függvényében modulról van szó, a vezérlő használtság-számlálóját megnöveljük a `MODULE_INC_USE_COUNT` hívással. Ezzel a makróhívással, ha a vezérlőt modulként fordítottuk, a vezérlőt addig nem törölhető, amíg a megfelelő `MODULE_DEC_USE_COUNT` makró le nem fut. Növeljük saját felhasználás-számlálónkat és a mutatót mentsük a fájlrendszerben található belső szerkezetbe. Ezt azért tesszük, hogy a későbbi fájlműveletek engedélyezzék a vezérlőnek, hogy az meghatározhassa, melyik eszközt címzi meg a felhasználó. Mindezeket a következő kód hajtja végre:

```
/* n velj k meg a modulhasznÆlat-szÆmlÆl t*/
MOD_INC_USE_COUNT;
```

```
++skel->open_count;

/* objektumunkat ments k a fÆjl sajÆt
   szerkezetØbe */
file->private_data = skel;
```

Az `open` függvény meghívása után a `read` és `write` következik, hogy az eszköznek adatot fogadjanak, illetve küldjenek. A `skel_write` függvényben olyan adatra fogadunk mutatót, amit a felhasználó az eszközre szeretne küldeni. A függvény meghatározza, hogy az elkészített írási ***urb*** alapján mennyi adatot tud az eszközre küldeni (ez a méret függ a tömeges kimenetnek az eszközben meghatározott végpontjától). Ezután az adatot a felhasználói területről a rendszermagterületre másolja, az ***urb***-ot rávezeti az adatra, és kiküldi az az USB-alrendszernek (lásd a *2. listát*).
Amikor a kiíró ***urb*** a megfelelő adatokkal a `FILL_BULK_URB` függvény segítségével fel lett töltve, az ***urb*** befejezésjelző visszahívóját (`completion callback`) úgy állítjuk be, hogy a saját `skel_write_bulk_callback` függvényünket hívja meg. Ez a függvény fog ugyanis meghívódni, amikor az USB-alrendszer az ***urb***-ot befejezi. A visszahívási függvény a megszakításkor hívódik meg, ezért figyelnünk kell, hogy ilyenkor ne túl sok számítást használjunk. A mi `skel_write_bulk_callback` megvalósításunk egyszerűen csak jelzi, hogy az ***urb*** befejezése sikeres volt-e vagy sem, majd kilép.
Az olvasás kicsit másképpen működik, mint az írás, ugyanis az eszközről a vezérlőhöz történő adatátvitelhez nincs szükségünk ***urb***-okra. Ehelyett meghívjuk az `usb_bulk_msg` függvényt, amelyet arra használhatunk, hogy az eszközről ***urb***-ok készítése nélkül küldjünk vagy fogadjunk adatot, és az ***urb*** befejezésjelző visszahívásait kezeljük. Meghívjuk az `usb_bulk_msg` függvényt egy átmeneti tárat adva meg neki, ahová az eszközről érkező adatokat helyezheti, valamint egy időtúllépési határértéket. Ha az időtúllépési érték anélkül jár le, hogy az eszközről adatot kapnánk, a függvény sikertelen és hibaüzenetet ad vissza (lásd a 3. listát; 23. CD Magazin/USB könyvtár).
Az `usb_bulk_msg` függvény nagyon hasznos lehet, ha az eszközzel egyszeri olvasásokat vagy írásokat akarunk végezni; ugyanakkor ha az eszközről folyamatosan kell írni vagy olvasni, ajánlott, hogy saját ***urb***-jainkat állítsuk föl és küldjük el őket az USB-alrendszernek.
Amikor a felhasználói program elereszti a fájlkezelőt, amit eddig az eszközzel való kapcsolattartásra használt, a vezérlő `release` függvénye hívódik meg. Ebben a függvényben a modul használatszámlálót a `MOD_DEC_USE_COUNT` használatával (a korábbi `MOD_INC_USE_COUNT` hívásunk párjaként) csökkentjük. Azt is meghatározzuk, hogy jelenleg van-e valamilyen más program, amely éppen az eszközhöz beszél (az eszközt egy időben több program is megnyithatta). Ha a felhasználó az eszköz utolsó használója, az összes esetleg még futó írást leállítjuk. Ezt a következőképpen tehetjük meg:

```
/* cs kkents k az eszk zh z tartoz
   hasznÆlatszÆmlÆl t */
--skel->open_count;
if (skel->open_count <= 0) {
   /* `ll tsunk le minden  rÆst, ami esetleg
      mØg megy */
   usb_unlink_urb (skel->write_urb);
   skel->open_count = 0;
}
```

```
/* cs kkents k a modulhoz tartoz
   hasznÆlatszÆmlÆl t */
MOD_DEC_USE_COUNT;
```

Az egyik legnehezebb feladat, amit az USB-vezérlőnek könnyedén kezelnie kell, hogy az USB-eszközt a rendszerből bármikor eltávolíthatják, még akkor is, ha a program éppen kapcsolatban áll vele. Ehhez az kell, hogy a pillanatnyi olvasási és írási műveleteket le lehessen állítani, és a felhasználói programot értesíthessük, hogy az eszköz már nincs többé jelen (lásd a 4. listát; 23. CD Magazin/USB könyvtár).
Ha a program rendelkezik az eszközhöz nyitott kezelővel, az `usb_device` szerkezetet a helyi szerkezetben egyszerűen csak `null`-ra állítjuk, mivelhogy már nem létezik. Minden írás, olvasás, elereszt és más olyan szolgáltatás, amely az eszköz jelenlétét feltételezi, először ellenőrzi, hogy az `usb_device` szerkezet jelen van-e még. Amennyiben nincs, elereszti az eltűnt eszközt, és a felhasználói programnak egy `-ENODEV` hibaüzenetet ad vissza. Ha az elereszt függvénye hívódik meg, előbb meghatározza, hogy valóban megszűnt-e már az `usb_device` szerkezet, és ha nem, akkor szokásos módon törli a `skel_disconnect` függvényt, éppúgy, mintha az eszközön nem lennének nyitott fájlok (lásd az 5. listát; 23. CD Magazin/USB könyvtár).
Ez az USB-vezérlőváz a megszakításokra vagy az eszközre küldött, illetve onnan fogadott `isochronous` adatokra semmilyen példát nem tartalmaz. A megszakításadatokat csaknem pontosan úgy kell küldeni, mint a tömegadatokat, néhány kisebb eltérés van közöttük. Az `isochronous` adat másképpen működik: az eszközről állandó adatfolyamokat küld vagy fogad. Az audio- és videokamera-vezérlők nagyon jó példák az `isochronous` adatkezelésre, és hasznosak lehetnek, ha nekünk is ilyesmit kell tennünk.
Linuxos USB-eszközvezérlő írása nem nehéz feladat, amint azt az USB-vezérlőváz is jól példázza. Ez a vezérlő más jelenlegi USB-vezérlőkkel kombinálva már elég példát nyújt, hogy egy kezdő alkotó rövid idő alatt működő vezérlőt készíthessen.
A linux-usb-devel levelezőlista levéltára ugyancsak rengeteg hasznos adatot tartalmaz.

Greg Kroah-Hartman
a Linux-rendszermag USB-fejlesztőinek egyike. Ingyenes programját sokkal több ember használja, mint amennyit zárt forrású projektek fejlesztéséért valaha is fizettek neki.

Kapcsolódó címek

Linux Hotplug Project ➲ http://linux-hotplug.sourceforge.net
The Linux USB Project ➲ http://www.linux-usb.org
A Linux alatt használható USB-eszközök listája
➲ http://www.qbik.ch/usb/devices
A linux-usb-devel levelezőlista levéltára
➲ http://marc.theaimsgroup.com/?l=linux-usb-devel
Programming Guide for Linux USB Device Drivers
➲ http://usb.cs.tum.edu/usbdoc
USB honlap ➲ http://www.usb.org
Linux alatt működő USB-csatolós lapolvasók listája
➲ http://www.buzzard.org.uk/jonathan/scanners-usb.html
USBMan-Linux USB Guide
➲ http://www.usbman.com/linuxusb.htm