A tty-réteg

Az eszközmeghajtók fejlesztéséről szóló sorozatunk első részében azzal foglalkozunk, hogyan kezeli a rendszermag a rendszerkonzolt és a soros kapukat.

Köszöntöm az olvasót a „Driving Me Nuts” című új rovatban. Ebben a rovatban a Linux-rendszermag különböző eszközmeghajtó-alrendszereit vizsgáljuk, megpróbáljuk átlátni az általuk és az eszközmeghajtókon keresztül biztosított különböző felületek széles skáláját. Ha valaki azt szeretné, hogy egy adott alrendszert közelebbről megvizsgáljunk, kérem, küldjön levelet.

Számos nagyon jó referencia létezik a Linux-rendszermag és a Linux-eszközmeghajtók programozásának témakörében (lásd a ***Kapcsolódó címeket***). Ez a rovat feltételezi, hogy legalább futólag ismerjük ezeket, vagy elérhető közelségben vannak olvasóink számára.

Vágjunk neki az útnak: először vizsgáljuk meg a rendszermag tty-rétegét. Ezt a réteget minden Linux-felhasználó igénybe veszi, amikor parancssorba ír, vagy a soros kapun keresztül lép kapcsolatba valamilyen eszközzel.

Minden tty-meghajtó számára létre kell hozni egy `struct tty_driver`-t, amely leírja saját magát, és a tty-réteg segítségével bejegyzi a felépítését. A `struct tty_driver` az ***include/linux/tty_driver.h*** állományban kerül megadásra.

Az *1. lista* mutatja, hogyan épül fel ez a szerkezet a 2.4.18 rendszermagváltozattól kezdve. A szerkezet mérete igen tekintélyes, úgyhogy próbáljuk meg kisebb részekre bontani.

A ***magic*** mezőnek minden esetben a `TTY_DRIVER_MAGIC` értéket kell felvennie. Ezt a tty-réteg annak ellenőrzésére használja, hogy valóban tty-meghajtóról van szó.

A ***driver_name*** és ***name*** mezők az eszközmeghajtó leírására használatosak. A ***driver_name*** mező tartalmát, ami a ***/proc/tty/drivers***c fájlban is megjelenik, valamilyen kifejező értékre kell beállítani. A ***name*** mező annak megadására szolgál, hogy milyen ***/dev*** vagy ***devfs*** név szolgál alapul a meghajtónkhoz. Például a rendszermag soros meghajtója a ***driver_name*** mező értékét `serial`-ra, a ***name*** mezőét pedig `ttyS`-re állítja, ha a ***devfs*** nincs engedélyezve és `tts/%d`-re, ha engedélyezve van. Amennyiben a ***devfs*** engedélyezett, a mezőt használja az eszközmeghajtó számára történő új eszközcsomópont létrehozására. A ***name*** mező `%d` része az eszköz mellékszámával (minor number) kerül feltöltésre, amikor a tty-alrendszer számára megtörténik a bejegyzése.

A ***name_base*** mezőre csak akkor van szükség, ha az eszköz számozása nem a 0 mellékszámmal kezdődik. Csaknem minden eszközmeghajtónál a 0 értéket kell beállítani.

A ***major***, ***minor_start*** és ***num*** mezők írják le, hogy a tty-rétegben eszközmeghajtónkhoz milyen fő-, illetve mellékszámok vannak rendelve. A major mező tartalmának az eszközmeghajtóhoz rendelt főszámnak (major number) kell lennie. Amennyiben új eszközmeghajtót hozunk létre, olvassuk el a ***Documentation/devices.txt*** leírást arra vonatkozóan, hogyan biztosíthatunk eszközmeghajtónk számára új főszámot. Annak is hasznos a fájl elolvasása, aki arra kíváncsi, hogy az egyes eszközmeghajtók milyen fő- és mellékszámpárokat használnak. A ***minor_start*** mező az eszköz által használt első mellékszám megadására szolgál. Ha eszközmeghajtónkhoz egy teljes főszám hozzá van rendelve, annak értéke 0 kell legyen. A ***num*** mező értéke mutatja meg, hogy hány különböző mellékszámot rendeltünk az eszközmeghajtóhoz.

Vagyis amennyiben eszközmeghajtónk a 188-as főszámmal rendelkezik, az alábbi értékeket kell beállítani:

```
major: 188
       minor_start: 0
       num: 255
```

A ***type*** és ***subtype*** mezők írják le, hogy eszközmeghajtónk milyen típusú tty-meghajtóként szerepel a tty-réteg számára. A ***type*** mező az alábbi értékeket veheti fel:

- TTY_DRIVER_TYPE_SYSTEM: a tty-alrendszer belső használatára van fenntartva annak közlésére, hogy belső tty-eszközmeghajtóról van szó. Ha ezt az értéket veszi fel, akkor a ***subtype*** értéke SYSTEM_TYPE_TTY, SYSTEM_TYPE_CONSOLE, SYSTEM_TYPE_SYSCONS vagy SYSTEM_TYPE_SYSPTMX lehet. Ezt a típust normál tty-eszközmeghajtó nem használhatja.
  TTY_DRIVER_TYPE_CONSOLE: csak a konzolmeghajtó használhatja, másra ne használjuk!
- TTY_DRIVER_TYPE_SERIAL: bármilyen soros eszköz meghajtója használhatja. Ez esetben a ***subtype*** értéke SERIAL_TYPE_NORMAL vagy SERIAL_TYPE_CALLOUT, a meghajtó típusától függően. A ***type*** mezőnek ez az egyik leggyakrabban használt beállítása.
- TTY_DRIVER_TYPE_PTY: a pseudo terminal felület (pty) által használt beállítás. Ha ezt az értéket használjuk, a subtype a PTY_TYPE_MASTER vagy PTY_TYPE_SLAVE értéket veheti fel.

  Az ***init_termios*** mező a kezdeti időértékek (sorbeállítások, sebességek) beállítására szolgál az eszköz első létrehozásakor.

  A ***flags*** mező az eszközmeghajtó igényeitől függően az alábbi bitértékek eredőjét veszi fel:
- TTY_DRIVER_INSTALLED: a bit beállításakor az eszközmeghajtó nem tudja saját magát bejegyezni a tty-réteggel, úgyhogy ne használjuk.
- TTY_DRIVER_RESET_TERMIOS: a bitet magasra állítva a tty-réteg nullázza az időértékeket, amikor az utolsó folyamat lezárja az eszközt. Konzol- és pty-meghajtók esetén hasznos.
- TTY_DRIVER_REAL_RAW: e bit bekapcsolása azt jelzi, hogy az eszközmeghajtó biztosítja a párosságadat (parity) vagy a megszakításjelző karakterek jelentését a sorvezérlő felé, amennyiben a sorvezérlőt erre nem utasítjuk. Általában minden eszközmeghajtónál beállítjuk, ez lehetővé teszi a sorvezérlő hatékonyságának növelését.
- TTY_DRIVER_NO_DEVFS: a bit magas értékénél a `tty_register_driver()` hívása nem hoz létre ***devfs*** bejegyzést. Bármilyen eszközmeghajtó esetén hasznos lehet, ha az eszköz fizikai jelenlététől függően a fő eszközöket dinamikusan akarjuk létrehozni, illetve megsemmisíteni. Példák olyan meghajtókra, amelyek használják a bitet: USB soros meghajtók, USB modemmeghajtó és az USB Bluetooth tty-meghajtó.

1. lista A struct tty_driver

```
struct tty_driver {
   int magic;
   const char *driver_name;
   const char *name;
   int name_base;
   short      major;
   short      minor_start;
   short      num;
   short      type;
   short      subtype;
   struct termios   init_termios;
   int flags;
   int *refcount;
   struct proc_dir_entry   *proc_entry;
   struct tty_driver*other;

   /*
   * Pointer to the tty data structures
   */
   struct tty_struct **table;
   struct termios **termios;
   struct termios **termios_locked;
   void *driver_state;

   /*
   * Interface routines from the upper tty
   ➥layer to the tty driver.
   */
   int   (*open)(struct tty_struct *
   ➥tty, struct file * filp);
   void (*close)(struct tty_struct *
   ➥tty, struct file * filp);
   int   (*write)(struct tty_struct * tty,
   ➥int from_user, const unsigned char
   ➥*buf, int count);
   void (*put_char)(struct tty_struct *tty,
   ➥unsigned char ch);
   void (*flush_chars)(struct tty_struct *tty);
   int   (*write_room)(struct tty_struct *tty);
   int   (*chars_in_buffer)(struct tty_struct
   ➥*tty);
   int   (*ioctl)(struct tty_struct *tty,
   ➥struct file * file, unsigned int cmd,
   ➥unsigned long arg);
   void (*set_termios)(struct tty_struct
   ➥*tty, struct termios * old);
   void (*throttle)(struct tty_struct * tty);
   void (*unthrottle)(struct tty_struct * tty);
   void (*stop)(struct tty_struct *tty);
   void (*start)(struct tty_struct *tty);
   void (*hangup)(struct tty_struct *tty);
   void (*break_ctl)(struct tty_struct *tty,
   ➥int state);
   void (*flush_buffer)(struct tty_struct *tty);
   void (*set_ldisc)(struct tty_struct *tty);
   void (*wait_until_sent)(struct tty_struct
   ➥*tty, int timeout);
   void (*send_xchar)(struct tty_struct *tty,
   ➥char ch);
   int   (*read_proc)(char *page, char **start,
   ➥off_t off, int count, int *eof, void *data);
   int   (*write_proc)(struct file *file,
   ➥const char *buffer, unsigned long count,
   ➥void *data);

   /*
   * linked list pointers
   */
   struct tty_driver *next;
   struct tty_driver *prev;
};
```

A ***refcount*** mező egy a tty-meghajtóban lévő egészre mutató pointer típusú változó; a tty-réteg használja a meghajtó hivatkozásainak kezelésére, a tty-meghajtónak nem szabad használnia.
A ***proc_entry*** mező beállítását nem közvetlenül a tty-meghajtó végzi. Ha a tty-meghajtó `write_proc` vagy `read_proc` függvényt valósít meg, ez a mező tartalmazza a létrejövő meghajtó ***proc_entry*** mezőjét. A másik mezőt csak a pty-meghajtó használja, tty-meghajtók nem alkalmazhatják.
Van még néhány különböző tty-szerkezetre mutató pointer típusú változónk. Az egyik a ***table*** mező, ami egy `tty_struct`-mutatókból álló tömbre mutat. A ***termios*** és a termios_locked mezők egy `struct termios` mutatókból álló tömbre mutatnak. E tömbök mindegyikének annyi eleme van, amennyit a feljebb említett ***minor*** mezőben beállítottunk. Ezeket a tty-réteg használja a különböző mellékszámú eszközök megfelelő kezeléséhez, a tty-meghajtónak közvetlenül nem szabad módosítania.
A ***driver_state*** mezőt csak a pty-meghajtó használja, más tty-meghajtónak nem szabad módosítania.
A `tty_driver` szerkezet a különböző függvénymutatók hosszú listáját tartalmazza. Ezeket a tty-réteg használja a tty-meghajtó meghívására, amikor valamilyen műveletet akar végrehajtani. Egy tty-meghajtónak nem kell az összes függvénymutatót megadni, ám amennyiben ezt a függvényt a tty-meghajtó végrehajtja, néhányat közülük kötelező jelleggel meg kell adnia.
Az `open` függvényt a tty-réteg akkor hívja, amikor az `open(2)` azon az eszközcsomóponton meghívása kerül, amihez tty-meghajtónk rendelve van. A tty-réteg egy fájlmutatóval, valamint egy olyan mutatóval végzi a függvényhívást, amely az eszközhöz rendelt `tty_struct` szerkezetre mutat. Ezt a mezőt a tty-meghajtónak a helyes működéshez mindenképpen be kell állítania a helyes működéshez (ellenkező esetben a felhasználó az `open(2)` hívásakor `-ENODEV` üzenetet kap).
A `close` függvényt a tty-réteg akkor hívja, amikor `release(2)` hívás érkezik a korábban az `open(2)` által létrehozott fájlmutatóra. Ez azt jelenti, hogy az eszközt le kell zárni.
A `write` függvény hívását a tty-réteg akkor kezdeményezi, amikor a tty-eszköz felé adatot kell továbbítani. Az adat érkezhet a felhasználó, illetve a rendszermag felől (a `from_user` bit beállításra kerül, ha a felhasználó felől érkezik). A függvénynek az eszközre kivitt karakterek számával kell visszatérnie. Ezt a függvényt a tty-meghajtónak kötelezően meg kell adnia.
A `put_char` függvényt a tty-réteg akkor hívja, amikor az eszközre egyetlen karaktert kell továbbítani. Amennyiben az eszközön nincs hely a karakter számára, figyelmen kívül hagyható. Amennyiben egy tty-meghajtó nem adja meg ezt a függvényt, a tty-réteg a `write` függvényt hívja, ha egyetlen karaktert akar küldeni.

2. lista Egy minimál tty-meghajtó

```
#define TINY_TTY_MAJOR 240 /* experimental range */
#define TINY_TTY_MINORS 255 /* use the whole
➥major up */
static int tty_refcount;
static struct tty_struct
➥*tiny_tty[TINY_TTY_MINORS];
static struct termios
➥*tiny_termios[TINY_TTY_MINORS];
static struct termios
➥*tiny_termios_locked[TINY_TTY_MINORS];

static struct tty_driver tiny_tty_driver {
    magic:   TTY_DRIVER_MAGIC,
    driver_name:      "tiny_tty",
#ifdef CONFIG_DEVFS_FS
    name:    "tts/ttty%d",
#else
    name:    "ttty",
#endif
    major:   TINY_TTY_MAJOR,
    num:     TINY_TTY_MINORS,
    type:    TTY_DRIVER_TYPE_SERIAL,
    subtype: SERIAL_TYPE_NORMAL,
    flags:   TTY_DRIVER_REAL_RAW |
             TTY_DRIVER_NO_DEVFS,

    refcount:         &tiny_refcount,
    table:   tiny_tty,
    termios: tiny_termios,
    termios_locked:   tiny_termios_locked,
    open:    tiny_open,
    close:   tiny_close,
    write:   tiny_write,
    write_room:       tiny_write_room,
};

static int __init tiny_init (void)
{
       /* register the tty driver */
       tiny_tty_driver.init_termios =
       ➥tty_std_termios;
       tiny_tty_driver.init_termios.c_cflag =
       ➥B9600 | CS8 | CREAD | HUPCL | CLOCAL;
       if (tty_register_driver (&tiny_tty_driver)) {
         printk (KERN_ERR "failed to register
                ➥tiny tty driver");
            return -1;
        return 0;
}

static void __exit tiny_exit (void)
{
     tty_unregister_driver (&tiny_tty_driver);
}

module_init (tiny_init);
module_exit (tiny_exit);
```

A `flush_chars` függvényt a tty-réteg bizonyos számú karakter `put_char` függvénnyel történt kiírása után hívja. A tty-meghajtónak az eszközt a visszamaradt adatok azonnali továbbítására kell felszólítania.

A `write_room` függvényt a tty-réteg akkor használja, ha azt szeretné lekérdezni, hogy a tty-meghajtó még hány felhasználható hellyel rendelkezik az írási tárban. Ez a szám asszerint változhat, ahogy a karakterek kiürülnek az átmeneti tárból.

A `chars_in_buffer` függvény annak lekérdezésére szolgál, hogy hány karakter várakozik még mindig az írási átmeneti tárból való továbbításra.

Az `ioctl` függvényt a tty-réteg akkor hívja, amikor az eszközcsomópontra `ioctl(2)` hívás érkezik. Ez a tty-meghajtó számára eszközfüggő `ioctl` függvények végrehajtását teszi lehetővé. Amennyiben az `ioctl`-kérést a meghajtó nem támogatja, egy `ENOIOCTLCMD` üzenettel kell visszatérnie. Ennek köszönhetően a tty-réteg képes arra, hogy ha lehet, az általános `ioctl`-változatot valósítsa meg.

A `set_termios` függvényt a tty-réteg akkor hívja meg, amikor az eszköz termiosbeállításai megváltoznak. A tty-meghajtónak ekkor meg kell változtatnia az eszköz fizikai beállításait a termios-szerkezet különböző mezőinek értékeitől függően. A tty-meghajtónak képesnek kell lennie kezelni azt az esetet, amikor függvény hívásakor egy korábbi változó esetleg NULL értékkel bír.

A `throttle` és `unthrottle` függvények a tty-réteg bemeneti tárolójának túlterhelését segítenek kezelni. A throttle függvény akkor használatos, amikor a tty-réteg bemeneti tárolója kezd megtelni. A tty-meghajtónak meg kell próbálnia jeleznie az eszköznek, hogy ne küldjön neki több karaktert. Az `unthrottle` függvény akkor hívódik, amikor a bemeneti tár kiürült, és a tty-réteg újabb karakterek elfogadására kész. A tty-meghajtónak jeleznie kell az eszköznek, hogy az adatok fogadásának nincs akadálya.

A `stop` és `start` függvények sok tekintetben hasonlítanak a `throttle` és `unthrottle` függvényhez, de azt is jelzik, hogy a tty-meghajtónak az eszközhöz történő adatküldést félbe kell szakítania, és később folytatja a műveletet.

A `hangup` függvény azt jelzi a tty-meghajtónak, hogy a tty-eszköz tevékenységét fel kell függesztenie.

A `break_ctrl` függvény hívására akkor kerül sor, amikor a tty-meghajtónak az RS-232 kapu BREAK állapotát kell ki-bekapcsolnia. Ha az állapot `-1`, be kell kapcsolnia, ha `0`, akkor a BREAK állapot kikapcsolása szükséges. Amennyiben ezt a függvényt a tty-meghajtó végrehajtja, a tty-réteg kezeli a `TCSBRK`, `TCSBRKP`, `TIOCSBRK` és `TIOCCBRK` `ioctl`-eket. Ellenkező esetben ezek a tty-meghajtó ioctl függvénye számára továbbítódnak.

A `flush_buffer` függvény akkor kerül hívásra, ha a tty-meghajtónak az írási átmeneti tárból minden adatot ki kell ürítenie. Ez azt is jelenti, hogy a visszamaradó adatok elvesznek, és nem továbbítja őket az eszközhöz.

A `set_ldisc` függvényt a tty-réteg akkor hívja, amikor megváltoztatta a tty-meghajtó sorszerkezetét. Ez a függvény általában többször nem is használatos, ezért nem is kell beállítani.

A `wait_until_sent` függvényt a tty-réteg akkor hívja, amikor a tty-meghajtó írási tárában lévő összes adatot ki akarja küldeni az eszközre. A függvény a befejezésig nem térhet vissza, ennek eléréséhez az alvó üzemmód is megengedett.

A `send_xchar` függvény a nagyobb fontosságú XON és XOFF

karakterek tty-eszközre való küldését végzi.
A `read_proc` és `write_proc` függvények akkor használatosak, amikor a meghajtó egy `proc/tty/driver/<name>` bejegyzést akar megvalósítani, ekkor a `<name>` a feljebb leírt ***name*** mező nevét veszi fel. Ha ezek közül valamelyik függvény be van állítva, akkor a bejegyzés létrejön és bármilyen `read(2)` vagy `write(2)` hívás az alkalmas függvényhez továbbítódik. Végül a ***next*** és ***prev*** mezőket a tty-réteg az összes különböző tty-meghajtó összefűzésére használja, a tty-meghajtónak nem szabad hozzányúlnia.

Nincs read függvény?

Az egyik dolog, ami szemet szúrhat a fenti függvénylistában, a `read` függvény megvalósításának hiánya. A tty-réteg egy átmeneti tárral rendelkezik, amely adatokat küld a felhasználó számára, amikor `read(2)`-hívás érkezik valamelyik tty-eszköz csomópontja felől. Azt a tárat a tty-meghajtónak kell feltöltenie, ha adat érkezik az eszközről. Mivel a tty-adat nem mindig akkor áll rendelkezésre, amikor a felhasználó akarja, ennek a modellnek a használata szükséges. Ily módon a tty-réteg minden kapott adatot átmenetileg tárol, az egyedi tty-meghajtóknak pedig nem kell aggódniuk a forgalom elakadása miatt, amíg az adat feltűnik a tty-vonalon.

A Tiny tty Driver

Most hogy túl vagyunk a különböző mezőkön, valójában mik azok, amikre egy alapvető tty-meghajtó felélesztéséhez és futtatásához szükségünk van? A *2. listán* látható példa a lehető legkisebb tty-meghajtót tartalmazza. Ha a bemutatott lépéseket elvégeztük, hozzuk létre a `tiny_open`, `tiny_close`, `tiny_write` és `tiny_write_room` függvényeket, és már készen is vagyunk egy parányi tty-meghajtó létrehozásával. Ennek megvalósítására látható egy példa a 3. listán (39. CD Magazin /tty könyvtár). Ez a tty-meghajtó egy időzítőt hoz létre, ami – egy igazi eszközt utánozva – minden második másodpercben adatot helyez el a tty-rétegen. Továbbá megfelelően kezeli az eszköz szerkezetének zárolását, amennyiben SMP-gépen fut.

Az adatok áramlása

Amikor a tty-meghajtó open függvénye hívásra kerül, a meghajtótól azt várjuk, hogy adatokat mentsen a `tty_struct` változóba, amit átadunk neki. Ennek célja, hogy az eszköz-meghajtó tisztában legyen vele, melyik eszközre hivatkozunk, amikor a későbbi `close`, `write` vagy más függvényeket hívjuk. Ha ezt nem tesszük, a meghajtó a `MINOR(tty->device)` függvény segítségével adja át az eszköz mellékszámát.
Ha rápillantunk a `tiny_open` függvényre, látható, hogy a tty-meghajtóban a `tiny_serial` szerkezet kerül mentésre. Ez lehetővé teszi a `tiny_write`, `tiny_write_room` és `tiny_close` függvények számára lehetővé teszi, hogy visszakeressék a `tiny_serial` szerkezetet, és megfelelően változtassák meg.
A tty-meghajtó `open` és `close` függvényei ugyanarra az eszközre többször is hívhatók – ahogyan a különböző felhasználók és programok kapcsolódnak az eszközhöz. Ez megengedné, hogy az egyik folyamat adatokat írjon, míg egy másik olvas. Hogy minden helyesen folyjék le, meg kell számolnunk, hogy a kapu hányszor volt kinyitva és becsukva. Amikor először nyitjuk meg a kaput, elvégezhetjük a szükséges eszköz-előkészítést és memóriafoglalást. Az utolsó kapuzáráskor pedig megfelelően állíthatjuk le az eszközt és szabadíthatjuk fel a memóriát. Példaként figyeljük meg a `tiny_open()` és `tiny_close()` függvényeket, ahol látható, hogyan lehet megszámolni a kapu nyitásait. Amikor az eszköz felől adat érkezik, szükség van rá, hogy a tty-eszköz átmeneti fliptárjában helyezzük el. Ezt a következő kóddal is megvalósíthatjuk:

```
for (i = 0; i < data_size; ++i) {
        if (tty->flip.count >=
                ➥TTY_FLIPBUF_SIZE)
                tty_flip_buffer_push(tty);
        tty_insert_flip_char(tty, data[i], 0);
}
tty_flip_buffer_push(tty);
```

Ebben a példában biztosított, hogy ne történjen túlcsordulás, miközben adatokat helyezünk el a tárban. Azoknak a meghajtóknak az esetében, amelyek nagyon nagy sebességgel képesek az adatok fogadására, a `tty->low_latency` kapcsolót kell beállítani, ennek hatására az utolsó `tty_flip_buffer_push()` hívás azonnal végrehajtódik. A példában a `tty_timer` függvény egy egybájtos adatot helyez el a tty fliptárjában, majd az időzítőt a következő hívásra várva visszaállítja. Ezzel egy eszközről érkező lassú karakterfolyamot utánoz.
Eszközre történő adat küldésekor a `write` függvény hívódik meg. Fontos, hogy a függvény ellenőrzi a `from_user` jelzőt, így elkerüli, hogy a felhasználói adatok véletlenül közvetlenül a rendszermag területére legyenek másolva. A `write` függvény számára engedélyezett a rövid írási visszatérés. Ez annyit tesz, hogy az eszköz nem tudott minden kért adatot elküldeni.
A `write(2)` függvényt hívó felhasználói program feladata – a visszatérési érték alapján – annak eldöntése, hogy az összes adat tényleg el lett-e küldve. Sokkal könnyebb ezt az ellenőrzést a felhasználói területen lefolytatni, mint egy meghajtóprogramnak addig várakoznia, míg az adatok kiküldésre kerülhetnek.

A tty-felület változásai

A tty-réteg a 2.0, 2.2 és 2.4 rendszermagokkal igen megbízható, és ez idő alatt nagyon kevéssé változott. Így a 2.0-s változathoz írt meghajtó a 2.4-es maggal szinte változtatás nélkül működni fog. A 2.5-ös rendszermagsorozatban a tty-réteg átírásra van kijelölve, így ez a cikk is tartalmazhat olyan részeket, amelyek már nem érvényesek. Amennyiben kétség merülne fel, forduljunk annak a rendszermagnak az ***include/tty_driver.h*** fájljához, amelyikre a meghajtót fejleszteni szeretnénk. Továbbá vessünk egy pillantást a ***driver/char*** könyvtárban található bármelyik tty-meghajtóra olyan függvények megvalósításainak példáiért, amelyek e cikkben nem szerepelnek.

Összegzés

Áttekintettük a tty-réteg alapjait, feltárva a 2.4-es rendszermag `tty_driver`-szerkezetének összes mezőjét, kiemelve, hogy, melyekre van szükség egy eszközmeghajtó megvalósításához. A 3. listán látható ***tiny_tty.c*** meghajtó (39. CD Magazin/tty könyvtár) jó példa egy kis méretű, sikeresen működtethető tty-eszközmeghajtóra. A kód a jövőbeli saját tty-eszközmeghajtóink létrehozásához szabadon felhasználható.

Kapcsolódó címek a 39. CD Magazin/tty könyvtárában találhatók.

Linux Journal 2002. augusztus, 100. szám

Greg Kroah-Hartman
(greg@kroah.com) jelenleg a Linux USB és a PCI Hot Plug rendszermagfelelőse. Az IBM-nél dolgozik, ahol számos, a Linux rendszermagjával kapcsolatos kérdéssel foglalkozik.