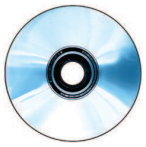

A tty-réteg (2. rész)

Az ioctl rendszerhívás azt jelenti, hogy a felhasználó az eszközt újra be szeretné állítani. Most megmutatjuk, hogyan kezeli mindezt az eszközmeghajtó.

Cikksorozatom előző részében (Linuxvilág 2002. szeptember, 34–37. oldal) a tty-réteg alapjairól volt szó, valamint megbeszéltük, hogyan készíthetünk kicsi, de működő tty-meghajtót. Most néhány nehezebb rész magyarázatával folytatjuk a tty-réteg feltárását.
Emlékszünk még az első részből a `struct tty_driver`-re, amelyet minden tty-meghajtónak meg kell valósítania? Vizsgáljuk meg most néhány olyan vonását, amit az elmúlt alkalommal nem tárgyaltunk meg minden részletében.

101 ioctls

A `struct tty_driver` `ioctl` függvény visszahívását a tty-réteg indítja, amikor az `ioctl(2)` hívása megtörténik az eszközcsomóponton. Amennyiben a meghajtó nem tudja az átadott `ioctl`-értéket kezelni, egy `-ENOIOCTLCMD` értéket kell visszaadnia annak érdekében, hogy a tty-réteg megpróbálkozhasson egy általános hívással. De milyen `ioctl`-értékek kezelésére érdemes meghajtónkat felkészítenünk?
A 2.4.19-es rendszermag körülbelül hatvan különböző lehetséges `tty-ioctl`-t határoz meg. A tty-meghajtónknak nem kell mindegyiket megvalósítania, de az alábbi általánosan használtakat mindenképpen érdemes:

- `TIOCMGET`: erre a hívásra akkor kerül sor, amikor a felhasználó a soros kapu (például DTR- vagy RTS-vonalak) állapotát akarja lekérdezni. Amennyiben soros kapunk MSR- vagy MCR-regisztereit közvetlenül tudjuk olvasni, vagy ezek másolatait helyben tároljuk (mint ahogyan néhány USB-soros eszköz számára ez szükséges), az `ioctl` megvalósítása például az alábbi módon történhet:

```
int tiny_ioctl (struct tty_struct *tty,

                struct file *file, unsigned
                ➥int cmd, unsigned long arg)
{
    struct tiny_private *tp = tty->private;

    if (cmd == TIOCMGET) {
        unsigned int result = 0;
        unsigned int msr = tp->msr;
        unsigned int mcr = tp->mcr;

        result = ((mcr & MCR_DTR)
                  ➥? TIOCM_DTR: 0)
                  /* DTR is set */

                  | ((mcr & MCR_RTS)
                      ➥? TIOCM_RTS: 0)
                  /* RTS is set */

                  | ((msr & MSR_CTS)
                      ➥? TIOCM_CTS: 0)
                  /* CTS is set */

                  | ((msr & MSR_CD)
                      ➥? TIOCM_CAR: 0)
                  /* Carrier detect is set*/
                  | ((msr & MSR_RI)
                      ➥? TIOCM_RI:  0)
                  /* Ring Indicator is set */

                  | ((msr & MSR_DSR)
                      ➥? TIOCM_DSR: 0);
                  /* DSR is set */

        if (copy_to_user((unsigned int *)arg,
                         &result,
                         ➥sizeof(unsigned int)))
            return -EFAULT;
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

- `TIOCMBIS`, `TIOCMBIC` és `TIOCMSET`: a tty-eszköz különböző modemvezérlő regisztereinek beállítására használatos. A `TIOCMBIS` hívás bekapcsolja az RTS, DTR, illetve hurok-eszköz (loopback) regisztereket, míg a `TIOCMBIC` hívás kikapcsolja őket. A `TIOCMSET` hívás mindhármat kikapcsolja, és csak a meghatározott értékeket állítja be. Lássunk egy példát ennek kezelésére:

```
int tiny_ioctl (struct tty_struct *tty,
                struct file *file,
                unsigned int cmd,
                unsigned long arg)
{
    struct tiny_private *tp = tty->private;

    if ((cmd == TIOCMBIS) ||
        (cmd == TIOCMBIC) ||
        (cmd == TIOCMSET)) {
        unsigned int value;
        unsigned int mcr = tp->mcr;

        if (copy_from_user(&value,
                           ➥(unsigned int *)arg,
                           ➥sizeof(unsigned int)))
            return -EFAULT;

        switch (cmd) {
        case TIOCMBIS:
            if (value & TIOCM_RTS)
                ➥mcr |= MCR_RTS;
            if (value & TIOCM_DTR)
                ➥mcr |= MCR_RTS;
            if (value & TIOCM_LOOP)
                ➥mcr |= MCR_LOOPBACK;
            break;
```

```
        case TIOCMBIC:
            if (value & TIOCM_RTS)
                ➥mcr &= ~MCR_RTS;
            if (value & TIOCM_DTR)
                ➥mcr &= ~MCR_RTS;
            if (value & TIOCM_LOOP)
                ➥mcr &= ~MCR_LOOPBACK;
            break;

        case TIOCMSET:
            /* turn off the RTS and DTR and
             * LOOPBACK, and then only turn on
             * what was asked for */
            mcr &=  ~(MCR_RTS | MCR_DTR
                    ➥ | MCR_LOOPBACK);
            mcr |= ((value & TIOCM_RTS)
                    ➥ ?  : 0);
            mcr |= ((value & TIOCM_DTR)
                    ➥ ? CR_DTR : 0);
            mcr |= ((value & TIOCM_LOOP)
                    ➥ ? MCR_LOOPBACK : 0);
            break;
        }

        /* set the new MCR value in the
         ➥device */
        tp->mcr = mcr;
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

Figyeljünk arra, hogy a hurokeszköz hívása (`TIOCM_LOOP`) a 2.2-es rendszermagból még hiányzik, de a 2.4-esben és az annál újabbakban már használható.

Ha tty-meghajtónk kezelni képes ezt a négy `ioctl`-t, a felhasználói programok nagy részével már együtt tud működni, bár mindig akad olyan program, amely valamilyen más `ioctl`-t igényel. Emiatt úgy is dönthetünk, hogy közülük is kezelni fogunk néhányat:

- `TIOCSERGETLSR`: a tty-eszközünkhöz tartozó vonalállapot-regiszter (Line Status Register – LSR) értékének visszakeresésére használatos.
- `TIOCGSERIAL`: hívásával eszközünk soros vonalának adatait egyszerre nyerhetjük vissza. Ehhez egy mutató kerül átadásra, ami `serial_struct` szerkezetű, és amelyet a meghajtónknak kell a megfelelő értékekkel feltöltenie. Néhány program (mint a `setserial` vagy a `dip`) a függvényt arra használja, hogy megállapítsa, jól lett-e beállítva az átviteli sebesség, valamint hogy tty-eszközünk típusáról általános adatokat gyűjtsön be. Íme egy példa ennek lehetséges megvalósítására:

```
int tiny_ioctl (struct tty_struct *tty,
                struct file *file,
                unsigned int cmd,
                unsigned long arg)
{
    struct tiny_private *tp = tty->private;

    if (cmd == TIOCGSERIAL) {
        struct serial_struct tmp;

        if (!arg)
            return -EFAULT;

        memset(&tmp, 0, sizeof(tmp));

        tmp.type    = tp->type;
        tmp.line    = tp->line;
        tmp.port    = tp->port;
        tmp.irq     = tp->irq;
        tmp.flags   = ASYNC_SKIP_TEST
                    ➥ | ASYNC_AUTO_IRQ;
        tmp.xmit_fifo_size = tp->xmit_
                           ➥fifo_size;
        tmp.baud_base      = tp->baud_base;
        tmp.close_delay    = 5*HZ;
        tmp.closing_wait   = 30*HZ;
        tmp.custom_divisor = tp->custom
                           ➥_divisor;
        tmp.hub6           = tp->hub6;
        tmp.io_type        = tp->io_type;

        if (copy_to_user(arg, &tmp,
            ➥sizeof(struct serial_struct)))
            return -EFAULT;
        return 0;
        }
    return -ENOIOCTLCMD;
}
```

- `TIOCSSERIAL`: a `TIOCGSERIAL` ellentéte; ennek segítségével eszközünk soros vonalának állapota egy lépésben állítható be. A hívás számára át kell adni az eszköz kívánt beállításának megfelelő adatokkal feltöltött `serial_struct` szerkezetre mutató pointert. Ha eszközünk ezt a hívást nem is kezeli, csaknem minden program megfelelően fog működni.
- `TIOCMIWAIT`: ez egy igen érdekes hívás. Amikor a felhasználó ilyen `ioctl`-hívást küld, az addig a rendszermagban várakozik, amíg a tty-eszköz MSR-regiszterében valamilyen változás nem áll be. Az `arg` kapcsoló tartalmazza annak az eseménynek a típusát, amire a felhasználó várakozik. Ezt az `ioctl`-t gyakran használják az állapotvonalon történő változásra várás közben, jelezve, hogy az adatok készen állnak az eszközre való küldésre.
- `TIOCMIWAIT`: ennek `ioctl`-megvalósításánál körültekintően kell eljárnunk. Szinte minden ezt használó rendszermagmeghajtó használja az `interruptible_sleep_on()` hívást is, ami nem biztonságos. Ehelyett célszerű egy `wait_queue`-t (várakozási sor) használni, amellyel mindezek elkerülhetők.

  A `TIOCMIWAIT` megvalósításának egyik helyes módjára az alábbiakban láthatunk példát:

```
int tiny_ioctl (struct tty_struct *tty,
                struct file *file,
                unsigned int cmd,
                unsigned long arg)
{
    struct tiny_private *tp = tty->private;

    if (cmd == TIOCMIWAIT) {
        DECLARE_WAITQUEUE(wait, current);
        struct async_icount cnow;
        struct async_icount cprev;

        cprev = tp->icount;
```

```
        while (1) {
            add_wait_queue(&tp->wait, &wait);

set_current_state(TASK_INTERRUPTIBLE);
            schedule();
            remove_wait_queue(&tp->wait,
                              ➥&wait);

            /* see if a signal woke us up */
            if (signal_pending(current))
                return -ERESTARTSYS;

            cnow = edge_port->icount;
            if (cnow.rng == cprev.rng &&
                cnow.dsr == cprev.dsr &&
                cnow.dcd == cprev.dcd &&
                cnow.cts == cprev.cts)
                return -EIO;
                /* no change => error */

            if (((arg & TIOCM_RNG) &&
                 (cnow.rng != cprev.rng)) ||
                ((arg & TIOCM_DSR) &&
                 (cnow.dsr != cprev.dsr)) ||
                ((arg & TIOCM_CD)  &&
                 (cnow.dcd != cprev.dcd)) ||
                ((arg & TIOCM_CTS) &&
                 (cnow.cts != cprev.cts)) )
{
                return 0;
            }
            cprev = cnow;
        }
    }
    return -ENOIOCTLCMD;
}
```

Az MSR-regiszter változását a fenti kódrészletben található

```
wake_up_interruptible(&tp->wait);
```

sor érzékeli, ennek hívása biztosítja a kód helyes működését.

- `TIOCGICOUNT`: segítségével a felhasználó megtudhatja az előfordult soros vonali megszakítások számát. A hívás a rendszermagnak egy megfelelően feltöltött `serial_icounter_struct` szerkezetre mutató mutatót (pointert) kell átadjon. Ez a hívás gyakran használatos az imént tárgyalt `TIOCMIWAIT` `ioctl`-hívással együtt. Ha az eszközmeghajtónk működése közben bekövetkező összes megszakítást nyomon követjük, a hívást megvalósító kód egészen egyszerű is lehet. Erre a ***drivers/usb/serial/io_edgeport.c*** fájlban láthatunk példát.

A write() szabályai

A `tty_struct` `write()` hívása történhet megszakítási környezetből és felhasználói környezetből egyaránt. Ezt azért fontos tudni, mert megszakítási környezetből nem szabad olyan függvényt hívni, amelyik ideiglenesen felfüggesztheti a működését. Ide tartoznak azok a függvények, amelyek esetleg a `schedule()` függvényt hívhatják, többek között az olyan gyakran használt függvények, mint a `copy_from_user()`, a `kmalloc()` és a `printk()`. Ha mindenáron szüneteltetni akarjuk a működést, a pillanatnyi állapotot először az `in_interrupt()` függvény hívásával ellenőrizzük.
A `write()` hívása történhet olyankor, amikor maga a tty-alrendszer szeretne adatokat küldeni a tty-eszközön keresztül. Ez akkor fordulhat elő, ha a `tty_struct` szerkezetben nem valósítjuk meg a `put_char()` függvényt. (Emlékezzünk, hogyha nincs `put_char()` függvény, a tty-réteg a `write()` függvényt fogja használni.) Ez akkor jellemző, amikor a tty-réteg egy újsorkaraktert soremelés és újsorkarakterré szeretne átalakítani. A legfontosabb, amit ebben az esetben meg kell jegyeznünk, hogy `write()` függvényünk ilyen hívás esetén nem adhat vissza 0 értéket. Ez annyit tesz, hogy ezt az adatbájtot mindenképpen ki kell küldeni az eszközre, mivel a hívó (a tty-réteg) ***nem*** tárolja az adatot, hogy később próbálkozzon az adatküldéssel. Mivel a `write()` nem tudja megállapítani, vajon a `put_char()` helyett hívták-e – még egy bájt adat átküldése esetén sem –, lehetőleg úgy írjuk meg a `write()` függvényt, hogy legalább egy bájt adatot mindig fogadni tudjon. Számos USB–soroskapu-átalakító tty-meghajtó nem követi ezt a szabályt, ebből kifolyólag néhány végberendezés, ha ezen keresztül csatlakoztatjuk, nem működik megfelelően.

A set_termios() megvalósítása

A `set_termios()` függvényhívás megfelelően működő megvalósításához meghajtónknak képesnek kell lennie a `termios`-szerkezet összes lehetséges beállításának visszafejtésére (dekódolására). Ez meglehetősen bonyolult feladat, mivel a kapcsolattartó vonal összes beállítása ebbe a szerkezetbe van különféle módokon belesűrítve.
Listánkon (41. CD Magazin/tty könyvrár) a `set_termios()` hívás egy egyszerű megvalósítását mutatja, amely a felhasználó által kért összes különböző vonalbeállítást a rendszermag hibakeresési naplójába rögzíti.
Először is készítsünk egy másolatot a tty-szerkezet `cflags` változójáról, ezt sokszor fogjuk használni a következőkben:

```
            unsigned int cflag;
            cflag = tty->termios->c_cflag;
```

Ezután vizsgáljuk meg, hogy szükségesek-e további lépések. Ellenőrizzük például, hogy a felhasználó nem próbálkozik-e az általunk éppen használt beállításokkal. Ne dolgozzunk feleslegesen, ha nem szükséges.

```
* check that they really want us to change
 * something */
if (old_termios) {
    if ((cflag == old_termios->c_cflag) &&
        (RELEVANT_IFLAG
        ➥(tty->termios->c_iflag) ==
         RELEVANT_IFLAG
         ➥(old_termios->c_iflag))) {
            printk (KERN_DEBUG
            ➥" - nothing to change...\n");
            return;
    }
}
```

A `RELEVANT_IFLAG()` makró meghatározása:

```
#define RELEVANT_IFLAG(iflag)
(iflag & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

Ez a `cflags` változó fontos bitjeinek kimaszkolására használatos. Hasonlítsuk össze a kapott értéket a korábbi értékkel, hogy megállapíthassuk, van-e különbség. Ha nincs, semmit sem kell tennünk, visszatérhetünk. Vegyük észre, hogy mielőtt az `old_termios` változó adatmezőjét megpróbáltuk volna elérni, először azt ellenőriztük, hogy a mutató éppen mutat-e valahova. Ez a vizsgálat feltétlenül szükséges, hiszen előfordulhat, hogy a változó `NULL` értéket tartalmaz. Egy `NULL` értékű mutató mezőjének visszakérdezése csúnya hibát okoz a rendszermagban.

Most, miután már tudjuk, hogy változtatnunk kell a beállításokon, nézzük a kívánt adatméretet:

```
/* get the byte size */
switch (cflag & CSIZE) {
  case CS5:
      printk (KERN_DEBUG " - data bits = 5\n");
     break;
  case CS6:
      printk (KERN_DEBUG " - data bits = 6\n");
      break;
  case CS7:
      printk (KERN_DEBUG " - data bits = 7\n");
      break;
  default:
  case CS8:
      printk (KERN_DEBUG " - data bits = 8\n");
      break;
       }
```

Maszkoljuk a `cflag` változót a `CSIZE` bitmezővel, és vizsgáljuk meg az eredményt. Ha nem tudjuk megállapítani, mely bitek voltak beállítva, az alapértelmezett 8 adatbitet is használhatjuk. Ezután a kívánt párosságot (parity) ellenőrizzük:

```
/* determine the parity */
    if (cflag & PARENB)
        if (cflag & PARODD)
            printk (KERN_DEBUG
            ➥" - parity odd\n");
        else
            printk (KERN_DEBUG
            ➥" - parity even\n");
    else
        printk (KERN_DEBUG
        ➥" - parity none\n");
```

Először annak nézzünk utána, hogy a felhasználó akar-e valamilyen párosság ellenőrzését használni. Ha igen, ellenőriznünk kell, hogy melyik fajtát szeretné (párosat vagy páratlant).
A `stop-bit` beállítás ellenőrzése is egyszerűen végrehajtható:

```
/* figure out the stop bits requested */
if (cflag & CSTOPB)
    printk (KERN_DEBUG " - stop bits = 2\n");
else
    printk (KERN_DEBUG " - stop bits = 1\n");
```

Most már folytathatjuk a megfelelő adatáram-vezérlés beállításaival. Egy egyszerű módszerrel eldönthetjük, hogy használnunk kell-e az RTS/CTS-t:

```
/* figure out the flow control settings */
if (cflag & CRTSCTS)
    printk (KERN_DEBUG
    ➥" - RTS/CTS is enabled\n");
else
printk (KERN_DEBUG " - RTS/CTS is disabled\n");
```

A különböző programból megvalósított adatáram-beállítások, továbbá az indító és leállító karakterek meghatározása egy kicsit bonyolultabb:

```
/* determine software flow control */
/* if we are implementing XON/XOFF, set the
 * start and stop character in the device */
if (I_IXOFF(tty) || I_IXON(tty)) {
   unsigned char stop_char  = STOP_CHAR(tty);
   unsigned char start_char = START_CHAR(tty);

   /* if we are implementing INBOUND XON/XOFF */
   if (I_IXOFF(tty))
       printk (KERN_DEBUG
            " - INBOUND XON/XOFF is enabled, "
            "XON = %2x, XOFF = %2x",
            start_char, stop_char);
   else
            printk (KERN_DEBUG
                  ➥" - INBOUND XON/XOFF "
                  ➥"is disabled");

   /* if we are implementing OUTBOUND XON/XOFF */
   if (I_IXON(tty))
        printk (KERN_DEBUG
               " - OUTBOUND XON/XOFF is
                    ➥enabled, "
               ➥"XON = %2x, XOFF = %2x",
               ➥start_char, stop_char);
   else
        printk (KERN_DEBUG
               ➥" - OUTBOUND XON/XOFF "
               ➥"is disabled");
}
```

Végül szükségünk van az átviteli sebesség megállapítására. Szerencsére a `tty_get_baud_rate` függvény segítségével a `termios`-beállításokból kinyerhető az adat, mégpedig egész típusú értékként:

```
/* get the baud rate wanted */
printk (KERN_DEBUG " - baud rate = %d",
        tty_get_baud_rate(tty));
```

Most, hogy minden szükséges csatornabeállítást meghatároztunk, már csak rajtunk múlik, hogy az eszközt ennek az adatnak a segítségével megfelelően beállítsuk.

Egyéb tty-adatok

Vern Hoxie kitűnő leírásgyűjteményt írt példaprogramokkal arról, hogy a soros kapuk hogyan érhetőek el a felhasználói területről. Az anyag az ➲ ftp://scicom.alphacdc.com/pub/linux címen érhető el. Az adatok nagy része egy rendszermag-programozónak ugyan nem lesz túl hasznos, de az `ioctl(2)` parancsok leírásai közül néhány, és a tty-adatok beállításainak, illetve visszakeresésének különböző módjairól, s a háttérben meghúzódó előzményekről leírtak egészen jók. Aki tty-eszközmeghajtó írására szánja magát, annak melegen ajánlom ezeknek a részeknek az átolvasását, ha másért nem, azért, hogy tisztában legyenek vele, miként próbálják majd a felhasználók használni a meghajtónkat.

Végszó

Szeretnék köszönetet mondani ***Al Borchers***-nek, amiért segített megfejteni a `write()` hívás pontos, minden apró részletre kiterjedő működését. ***Peter Berger***-rel együtt ő a szerzője a Digi AccelePort eszközök számára írt ***drivers/usb/serial/digi_acceleport.c*** nevű USB soros átalakító meghajtónak. Ez a jól működő tty-eszközmehajtók tökéletes mintapéldája.

Linux Journal 2002. október, 102. szám

Greg Kroah-Hartman (greg@kroah.com) jelenleg a Linux-rendszerek USB és gyors csatlakoztatású (PCI Hot Plug) egységeinek rendszermagba épített meghajtóprogramjainak fejlesztője. Az IBM-nél dolgozik.